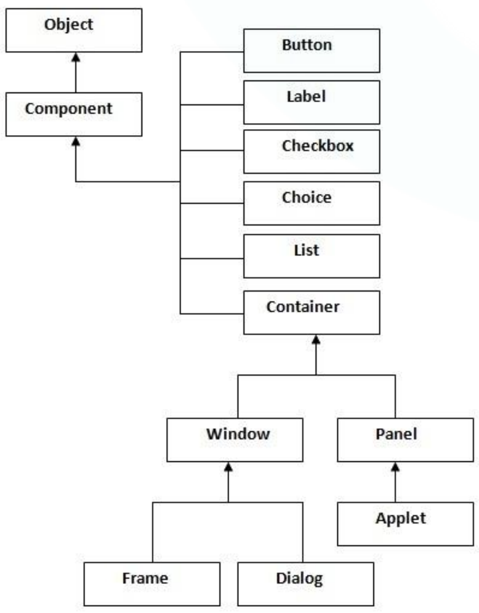
Abstract Windows Toolkit (AWT)

- AWT is the set of java classes that allows us to create GUI (Graphical User Interfaces) component and manipulate them.
- GUI Components are used to take the input from user in a user friendly manner.
- Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.
- Java AWT (Abstract Windowing Toolkit) is an API to develop GUI or window-based application in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system.
- The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Graphical User Interface

- Graphical User Interface (GUI) offers user interaction via some graphical components.
- For example our underlying Operating System also offers GUI via window,frame,Panel, Button, Textfield, TextArea, Listbox, Combobox, Label, Checkbox etc.
- These all are known as components. Using these components we can create an interactive user interface for an application.
- GUI makes it easier for the end user to use an application. It also makes them interesting.
- GUI provides result to end user in response to raised events.
- GUI is entirely based events. For example clicking over a button, closing a window, opening a window, typing something in a textarea etc. These activities are known as events.
- The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

The **Container** class is a subclass of **Component**. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**).

Panel

Panel is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

It can have other components like button, textfield etc.

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**.

Panel is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.

Window

The window is the container that has no borders and menu bars. Generally, you won't create **Window** objects directly; you must use frame, dialog or another window for creating a window.

The **Window** class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas

Canvas component represents a blank rectangular area where application can draw something or can receive inputs created by user.

AWT components

Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.

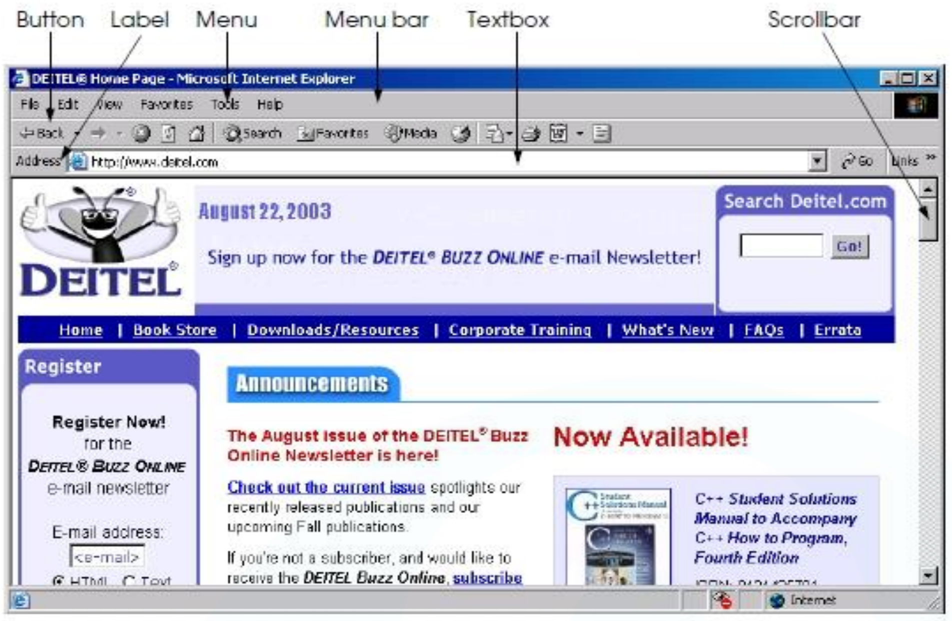

Button
Label
Menu
Menu bar
Textbox
Scrollbar
DEITEL® Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.deitel.com
August 22, 2003
Sign up now for the DEITEL® BUZZ ONLINE e-mail Newsletter!
Search Deitel.com
Go!
Home | Book Store | Downloads/Resources | Corporate Training | What's New | FAQs | Errata
Register
Announcements
Now Available!

Fig. 12.1 Sample Internet Explorer window with GUI components.

<u>working with frames</u>

Here are two of **Frame**'s constructors:

- Frame( ): Creates a standard window that does not contain a title.
- Frame(String title): Creates a window with the title specified by title.

There are several methods you will use when working with **Frame** windows.

setSize( ): to set the dimensions of the window.

void setSize(int newWidth, int newHeight)

setVisible( ): to make a created window visible. After a frame window has been created, it will not be visible until you call **setVisible( )**.

void setVisible(boolean visibleFlag)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

setTitle( ): to change window.

void setTitle(String newTitle)

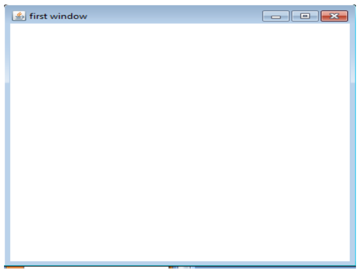
Here, newTitle is the new title for the window.

To create **simple awt example**, you need a frame. There are two ways to create a frame in AWT.

1. By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

1. To create a frame by extending Frame class

Output:



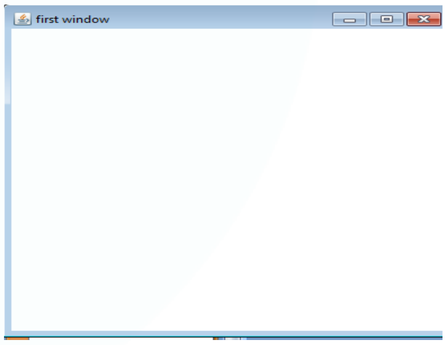
```
import java.awt.*;
class Myframe extends Frame
{
        Myframe ()
        {
                setSize(300,300);//frame size 300 width and 300 height
                setLayout(null);//no layout manager
                setVisible(true);//now frame will be visible, by default not visible
        }
        public static void main(String args[])
        {
                Myframe f=new Myframe();
        }
}
```

Here, we are showing Button component on the Frame.

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

2. To create a frame by creating the object of Frame class

```
import java.awt.*;
public class Myframe
{
        public static void main(String args[])
        {
                Frame f=new Frame("Frame");
                f.setSize(300,300);
                f.setLayout(null);
                f.setVisible(true);
                f.setTitle("Frame example")
        }
}
```



AWT Graphics Classes

Java's graphics class includes methods for drawing many different types of shapes, from simple lines to polygons to text in variety of fonts and colors.

Java provides us an easy way to draw text and graphics using GUI. Graphics class in AWT package allow us to draw primitive geometric types like line and circle. Other than this it can also display text.

Methods of Graphics Class

3. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
4. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
5. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
6. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
7. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
8. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
9. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
10. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
11. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
12. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
13. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;
/* <applet code="GraphicsDemo.class" width="300" height="300">  */
public class GraphicsDemo extends Applet
{
        public void paint(Graphics g)
        {
                g.setColor(Color.red);
                g.drawString("Welcome",50, 50);
                g.drawLine(20,30,20,300);
                g.drawRect(70,100,30,30);
                g.fillRect(170,100,30,30);
                g.drawOval(70,200,30,30);

                g.setColor(Color.pink);
                g.fillOval(170,200,30,30);
                g.drawArc(90,150,30,30,30,270);
                g.fillArc(270,150,30,30,0,180);
        }
}
```

AWT Color and Font

Working with color

- Java supports color in a portable, device independent fashion. Color is encapsulated by the Color class.
- The AWT color system allows you to specify any color you want.

Color Constructors:

1. Color(int red, int green, int blue)

The first constructor takes three integers that specify the color as a mix of red, green, and blue

These values must be between 0 and 255, as in this

Example:

```
new Color(255, 100, 100); // light red
```

2. Color(int rgbValue)

It takes a single integer that contains the mix of red, green, and blue packed into an integer.

The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7

Example:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);

Color darkRed = new Color(newRed);
```

3. Color(float red, float green, float blue)

It takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue

Color Methods:

By default, graphics objects are drawn in the current foreground color

setColor( ): Change the foreground color

void setColor(Color newColor) //newColor specifies the new drawing color

getColor( ): Returns the current color

Color getColor( )

set the background color and foreground color using the following methods:

void setBackground (Color newColor)

void setForeground (Color newColor)

Example of color.

```
import java.awt.*;
import java.applet.*;
/*<applet code="ColorDemo" width=300 height=200></applet>*/
public class ColorDemo extends Applet
{// draw lines
        public void paint(Graphics g)
        {
                Color c1 = new Color(255, 100, 100);
                Color c2 = new Color(100, 255, 100);
                Color c3 = new Color(100, 100, 255);
                g.setColor(c1);
                g.drawLine(0, 0, 100, 100);
                g.drawLine(0, 100, 100, 0);
                g.setColor(c2);
                g.drawLine(40, 25, 250, 180);
                g.drawLine(75, 90, 400, 400);
                g.setColor(c3);
                g.drawLine(20, 150, 400, 40);
                g.drawLine(5, 290, 80, 19);
                g.setColor(Color.red);
                g.drawOval(10, 10, 50, 50);
                g.fillOval(70, 90, 140, 100);
                g.setColor(Color.blue);
                g.drawOval(190, 10, 90, 30);
                g.drawRect(10, 10, 60, 50);
                g.setColor(Color.cyan);
                g.fillRect(100, 10, 60, 50);
                g.drawRoundRect(190, 10, 60, 50, 15, 15);
        }
}
```

Working with Fonts

- The AWT supports multiple type fonts
- Fonts are encapsulated by the Font class

Creating and Selecting a Font

To select a new font, you must first construct a Font object that describes that font.

Font constructor

Font(String fontName, int fontStyle, int pointSize)

Here, fontName specifies the name of the desired font. The style of the font is given by font style. fontStyle consists of one or more of these three constants: FontPLAIN, FontBOLD, FontITALIC

The size of the font is specified by pointSize

Font Methods

- **int getStyle():**get the style of current font
- **Font getFont():**Obtain information of currently selected font.
- **int getsize():**get the size of current font
- **String getName():**get the name of current font
- setFont(Font f):set the new font

  void setFont(Font fontObj) //Here, fontObj is the object that contains the desired font

AWT Control Fundamentals

Controls are components that allow a user to interact with your application in various ways.

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Radio Buttons
- Choice lists
- Lists
- Scroll bars
- Text Field
- Text Area

These controls are subclasses of Component.

Adding Controls

First create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**.

Component add(Component compObj)

Here, compObj is an instance of the control that you want to add.

Removing Controls

remove( ): used to remove a control from a window when the control is no longer needed.

This method is also defined by **Container**. Here is one of its forms:

void remove(Component *obj*)

obj is a reference to the control that we want to remove.

Remove all controls by calling **removeAll( )**.

Responding to Controls

Labels are passive controls. Except for labels, all controls generate events when they are accessed by the user.

For example, when the user clicks on a push button, an event is sent that identifies the push button.

The program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor.

1.Labels

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 constructors:

Label( ): Constructs an empty label.

Label(String text): Constructs a new label with the specified string of text, left justified.

Label(String text, int align): creates a label that contains the string specified by **text** using the alignment specified by **align**. The value of **align** must be one of these three constants: **Label.LEFT, Label.RIGHT**, or **Label.CENTER.**

- **static int CENTER** -- Indicates that the label should be centered.
- **static int LEFT** -- Indicates that the label should be left justified.
- **static int RIGHT** -- Indicates that the label should be right justified

Label Methods:

void setText(String str): To set or change the text in a label.

label1.setText("Label1");

void setAlignment(int align): To set the alignment of the string // Here, ***align*** must be one of the alignment constants.

label2.setAlignment(Label.CENTER);

String getText( ): To obtain the current label.

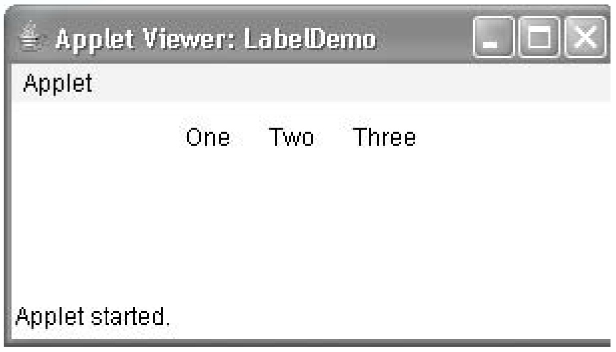
int getAlignment( ): To obtain the current alignment,

Demonstration of Label.

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

```
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);
```

Output:



```
add(three);
}
}
```

2. Button

A 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 constructors:

Button( ): creates an empty button

Button(String str): creates a button that contains str as a label

Button Methods

void setLabel(String str**)** Sets the button's label. Here, str becomes the new label for the button.

String getLabel() Gets the label of this button

void addActionListener(ActionListener l) Adds the specified action listener to receive action events from this button.

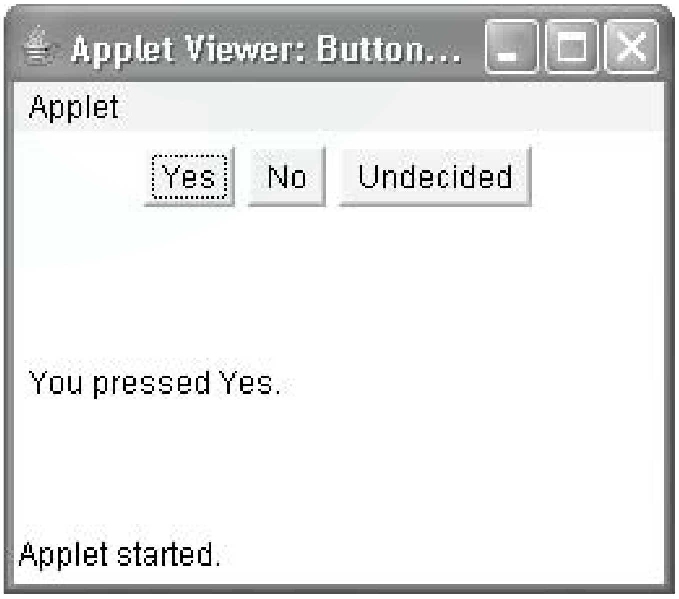
void removeActionListener(ActionListener l) Removes the specified action listener so that it no longer receives action events from this button

void setActionCommand(String command) Sets the command name for the action event fired by this button.

String getActionCommand() Returns the command name of the action event fired by this button.

Demonstration of Button.

```
import java.awt.*;
import java.awt.event.*;
class ButtonDemo extends Frame implements ActionListener
{
String msg = " ";
Button yes= new Button("Yes");
Button no= new Button("No");
Button maybe= new Button("Undecided");
Label l=new Label(msg);
public ButtonDemo(String s)
{
super(s);
setLayout(new FlowLayout());
add(yes);
add(no);
add(maybe);add(l);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
```



```
You pressed Yes.
Applet started.
```

```
{
String str = ae.getActionCommand();
if(str.equals("Yes"))
{
msg = "You pressed Yes.";
}
else if(str.equals("No"))
{
msg = "You pressed No.";
}
else
{
msg = "You pressed Undecided.";
}
l.setText(msg);
}
public static void main(String args[])
{
ButtonDemo b=new ButtonDemo("Demonstration of Buttons");
b.setSize(600, 250);
b.setVisible(true);
}
}
```

3. Checkbox

A check 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.

Each time a check box is selected or deselected, an item event is generated.

This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.

Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method.

An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

Checkbox constructors:

Checkbox( ): creates a check box whose label is initially blank

Checkbox(String str): creates a check box whose label is specified by str, State -unchecked

Checkbox(String str, boolean on): set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared.

Checkbox(String str, boolean on, CheckboxGroup cbGroup): create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null.

Checkbox Methods:

void setLabel(String str): To set the label.

String getLabel(): To obtain the current label.

void setState(boolean state): To set the current state of a check box.

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

void addItemListener(ItemListener l): Adds the specified item listener to receive item events from this check box.

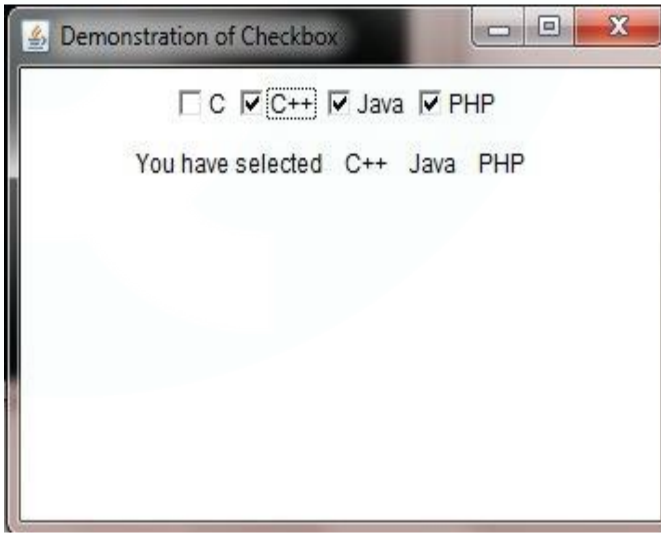
void setCheckboxGroup(CheckboxGroup g): Sets this check box's group to the specified check box group.

CheckboxGroup getCheckboxGroup(): Determines this check box's group.

Demonstration of Checkbox.

```
import java.awt.*;
import java.awt.event.*;
class CheckboxDemo extends Frame implements ItemListener
{
String msg = " ";
Checkbox c1 = new Checkbox("C", null, true);
Checkbox c2 = new Checkbox("C++");
Checkbox c3 = new Checkbox("Java");
Checkbox c4 = new Checkbox("PHP");
Label l=new Label(msg);
public CheckboxDemo(String s)
{
super(s);
setLayout(new FlowLayout());
add(c1);
add(c2);
add(c3);
add(c4);
add(l);
c1.addItemListener(this);
c2.addItemListener(this);
c3.addItemListener(this);
c4.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
msg="You have selected";
if(c1.getState())
msg=msg+" C";
if(c2.getState())
msg=msg+" C++";
if(c3.getState())
msg=msg+" Java";
if(c4.getState())
msg=msg+" PHP";
```

Output:



```
You have selected   C++   Java   PHP
```

```
l.setText(msg);
}
public static void main(String args[])
{
CheckboxDemo b=new CheckboxDemo ("Demonstration of Checkbox");
b.setSize(350, 250);
b.setVisible(true);
}
}
```

4. Radio Buttons/ Checkbox Group

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, because they act like the station selector on a car radio—only one station can be selected at any one time.

Checkbox methods:

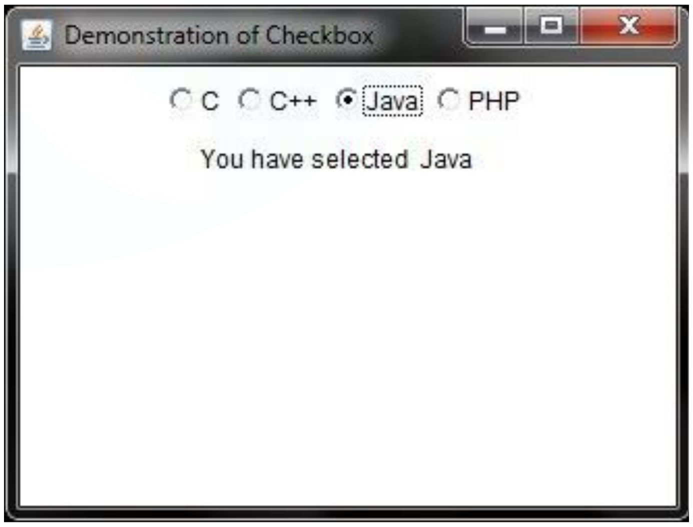
Checkbox getSelectedCheckbox( ): To determine which check box in a group is currently selected

void setSelectedCheckbox(Checkbox which): To set a check box. Here, **which** is the check box that you want to be selected. The previously selected check

box will be turned off.

Demonstration of Radio Buttons.

```
import java.awt.*;
import java.awt.event.*;
class RadioDemo extends Frame implements ItemListener
{
String msg = " ";
CheckboxGroup cbg=new CheckboxGroup();
Checkbox c1 = new Checkbox("C", cbg, true);
Checkbox c2 = new Checkbox("C++",cbg,false);
Checkbox c3 = new Checkbox("Java",cbg,false);
Checkbox c4 = new Checkbox("PHP",cbg,false);
Label l=new Label(msg);
public RadioDemo(String s)
{
super(s);
setLayout(new FlowLayout());
add(c1);
add(c2);
add(c3);
add(c4);
add(l);
c1.addItemListener(this);
c2.addItemListener(this);
c3.addItemListener(this);
c4.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
```



You have selected Java

```
{
msg="You have selected "+cbg.getSelectedCheckbox().getLabel();
l.setText(msg);
}
public static void main(String args[])
{
RadioDemo b=new RadioDemo ("Demonstration of Checkbox");
b.setSize(350, 250);
b.setVisible(true);
}
}
```

5. Choice

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu.

When the user clicks on Choice, the whole list of choices pops up, and a new selection can be made.

Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.

Each time a choice is selected, an item event is generated.

This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.

Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.

Choice constructor:

Choice only defines the default constructor, which creates an empty list.

void add(String name): To add a selection to the list

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add() occur.

Choice methods:

String getSelectedItem( ): returns currently selected string.

int getSelectedIndex( ): returns the index of the currently selected Item. The first item is at index 0. By default, the first item added to the list is selected.

These methods are shown here:

int getItemCount( ): To obtain the number of items in the list.

void select(int index): To set the currently selected item using the select( ) method with a zero-based integer index.

void select(String name): set the currently selected item using the select( ) method with a string that will match a name in the list.

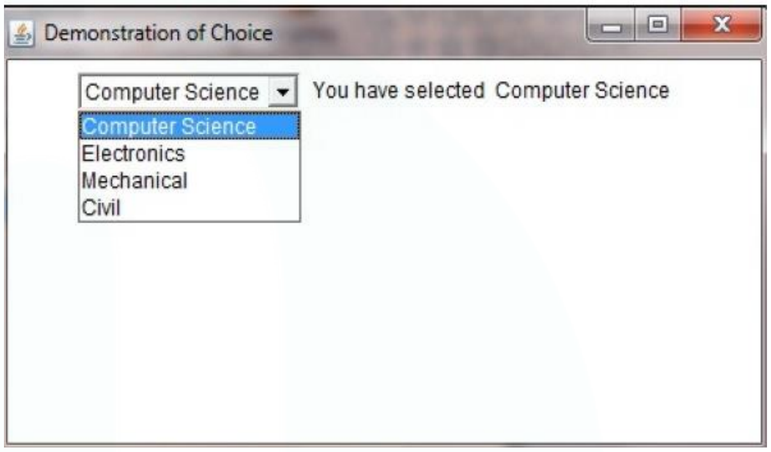
String getItem(int index): obtain the name associated with the item at specified index**.**

Here, index specifies the index of the desired item.

Demonstration of Choice.

```
import java.awt.*;
import java.awt.event.*;
class ChoiceDemo extends Frame implements ItemListener
{
String msg = " ";
Choice dept=new Choice();
Label l=new Label(msg);
public ChoiceDemo(String s)
{
super(s);
setLayout(new FlowLayout());
dept.add("Computer Science");
dept.add("Electronics");
dept.add("Mechanical");
dept.add("Civil");
add(dept);
add(l);
dept.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
msg="You have selected "+dept.getSelectedItem();
l.setText(msg);
}
public static void main(String args[])
{
ChoiceDemo b=new ChoiceDemo("Demonstration of Choice");
b.setSize(450, 250);
b.setVisible(true);
}
}
```

Output:



You have selected Computer Science

6. List

The **List** class provides a compact, multiple-choice, scrolling selection list.

Unlike the **Choice** object, this shows only the single selected item in the menu. It can also be created to allow multiple selections.

To process list events, you will need to implement the **ActionListener** interface.

Each time a **List** item is double-clicked, an **ActionEvent** object is generated.

Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item.

Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated.

Its **getStateChange( )** method can be used to determine whether a selection or deselection triggered this event.

getItemSelectable() returns a reference to the object that triggered this event.

<u>List constructors:</u>

List( ): Allows to select only one time at a time.

List(int numRows): the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed)

List(int numRows, boolean multipleSelect): 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.

To add a selection to the list, call **add( )**. It has the following two forms:

void add(String name): adds items to the end of the list.

void add(String name, int index): adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list. Here, name is the name of the item added to the list

<u>For lists that allow only single selection:</u>

String getSelectedItem( ): returns a string containing the name of the item If more than one item is selected or if no selection has yet been made, **null** is returned.

int getSelectedIndex( ): returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

<u>For lists that allow multiple selection:</u>

String[ ] getSelectedItems( ): returns an array containing the names of the currently selected items.

int[ ] getSelectedIndexes( ): To obtain the number of items in the list, call **getItemCount( )**.

int getItemCount( ): To obtain the number of item in the list.

void select(int index): To set the currently selected item.

String getItem(int index): To obtain the name associated with the item at that index.

Here, index specifies the index of the desired item.

<u>Demonstration of List box.</u>

```
import java.awt.*;
import java.awt.event.*;
class ListDemo extends Frame implements ItemListener
{
String msg = " ";
List dept=new List(4, true);
Label l=new Label(msg);
public ListDemo(String s)
{
```

```
super(s);
setLayout(new FlowLayout());
dept.add("Computer Science");
dept.add("Electronics");
dept.add("Mechanical");
dept.add("Civil");
add(dept);
add(l);
dept.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
int idx[];
msg="You have selected ";
idx = dept.getSelectedIndexes();
for(int i=0; i<idx.length; i++)
msg += dept.getItem(idx[i]) + " ";
l.setText(msg);}
public static void main(String args[])
{
ListDemo b=new ListDemo("Demonstration of List");
b.setSize(450, 250);
b.setVisible(true);
}
}
```

7. TextField

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 respond when the user presses enter. When this occurs, an action event is generated.

TextField constructors:

TextField( ): creates a default text field

TextField(int numChars): creates a text field that is numChars characters wide

TextField(String str: initializes the text field with the string contained in str

TextField(String str, int numChars): initializes a text field and sets its width.

TextField (and its superclass TextComponent): provides several methods that allow you to utilize a text field.

TextField Methods:

String getText( ): To obtain the current string in the text field.

void setText(String str): To set the text. // Here, str is the new string.

String getSelectedText( ): To obtain the currently selected text.

void select(int startIndex, int endIndex): select a portion of text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1.

getSelectedText( ): returns the selected text.

boolean isEditable( ): determine editability of the text field. isEditable( ) returns true if the text may be changed and false if not.

void setEditable(boolean canEdit): To set the editability. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered.

void setEchoChar(char ch): You can disable the echoing of the characters as they are typed by calling setEchoChar( ). Here, ch specifies the character to be echoed. This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown).

boolean echoCharIsSet( ): To check the echo set mode is activated or not.

char getEchoChar( ): To retrieve the echo character.

8. 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 called TextArea.

TextArea constructors:

TextArea( )

TextArea(int numLines, int numChars)

TextArea(String str)

TextArea(String str, int numLines, int numChars)

TextArea(String str, int numLines)

Layout Managers

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) Java has several predefined **LayoutManager** classes, several of which are: FlowLayout, BorderLayout, GridLayout.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, 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.

<u>**FlowLayout** constructors:</u>

FlowLayout( ): creates the default layout, which centers components and leaves five pixels of space between each component

FlowLayout(int how): specify how each line is aligned. Valid values for how are as follows:

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHTs

FlowLayout(int how, int horz, int vert): specify the horizontal and vertical space left between components in horz and vert, respectively.

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.

BorderLayout constructors:

BorderLayout( ): creates a default border layout

BorderLayout(int horz, int vert): 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

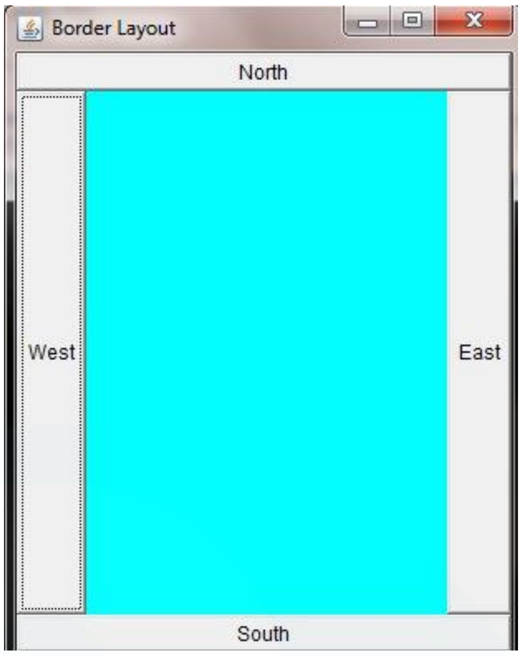
When adding components, you will use these constants with the following form of add( ), which is defined by Container:

void add(Component compObj, Object region);

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

Demonstration of Border Layout.

```
import java.awt.*;
class Border extends Frame
{
Button b1=new Button("East");
Button b2=new Button("West");
Button b3=new Button("North");
Button b4=new Button("South");
public Border(String s1)
{
super(s1);
setBackground(Color.cyan);
setLayout(new BorderLayout());
add(b1, BorderLayout.EAST);
add(b2, BorderLayout.WEST);
add(b3, BorderLayout.NORTH);
add(b4, BorderLayout.SOUTH);
}
public static void main(String args[])
{
Border c=new Border("Border Layout");
c.setSize(300,400);
c.show();
}
}
```



GridLayout

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

GridLayout constructors:

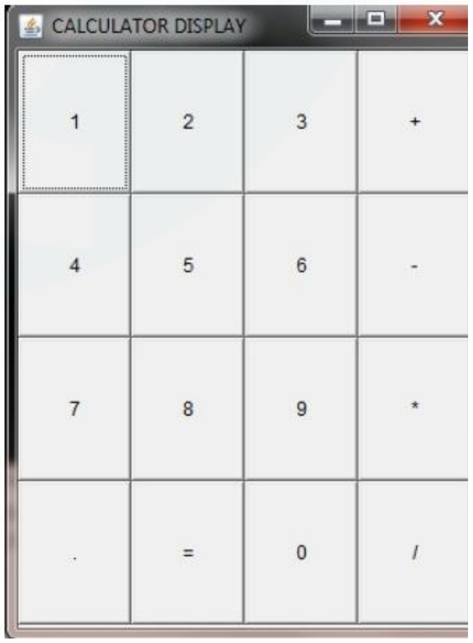
GridLayout( ): creates a single-column grid layout

GridLayout(int numRows, int numColumns): creates a grid layout with the specified number of rows and columns

GridLayout(int numRows, int numColumns, int horz, int vert): specify the horizontal and vertical space left between components in horz and vert, respectively.

Demonstartion of GridLayout.

```
import java.awt.*;
class Cal extends Frame
{
String s=new String("123+456-789*.=0/");
int i;
Button b[]=new Button[16];
public Cal(String s1)
{
super(s1);
for(i=0;i<16;i++)
b[i]=new Button(String.valueOf(s.charAt(i)));
setLayout(new GridLayout(4,4));
for(i=0;i<16;i++)
add(b[i]);
}
public static void main(String args[])
{
Cal c=new Cal("CALCULATOR DISPLAY");
```



```
c.setSize(300,400);
c.show();
}
}
```

Swing

Swing, which is an extension library to the AWT, includes new and improved components that enhance the look and functionality of GUIs.

Java Swing is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Swing can be used to build Standalone swing GUI Applications as well as Servlets and Applets.

Swing is a set of classes that provides more powerful and flexible GUI components than provided by the AWT.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The Swing-related classes are contained in javax.swing.

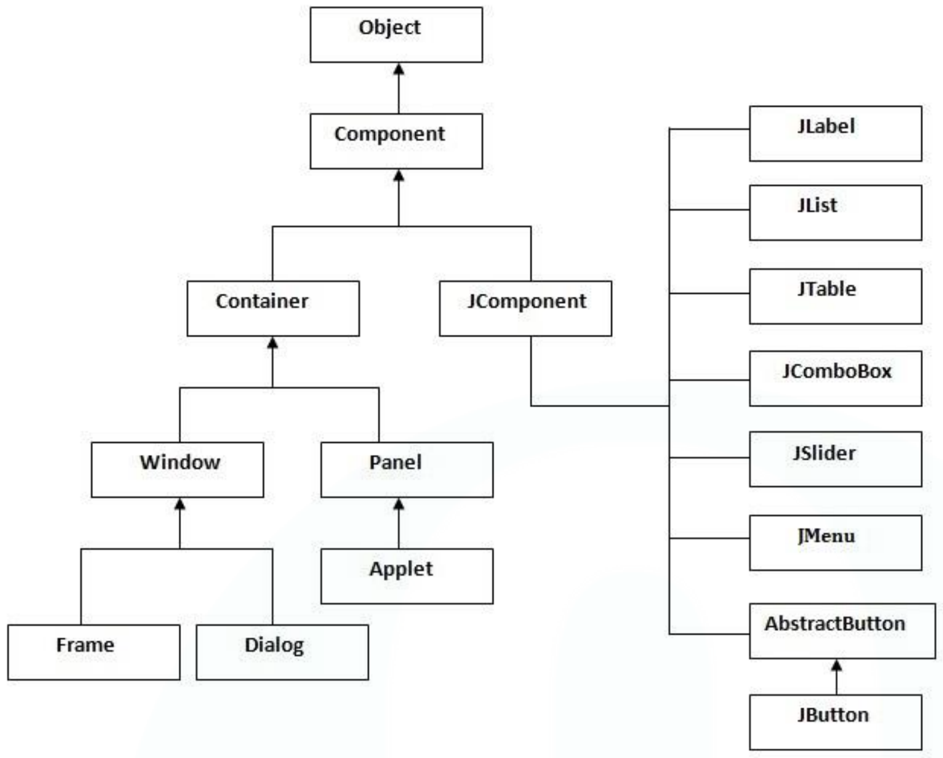
The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Swing supplies several exciting additions including tabbed panes, scroll panes, trees, and tables

Difference between AWT and Swing

Java AWT	Java Swing
AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
AWT components are **heavyweight**.	Swing components are **lightweight**.
AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Hierarchy of Java Swing classes



Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Java Swing Examples

There are two ways to create a frame:

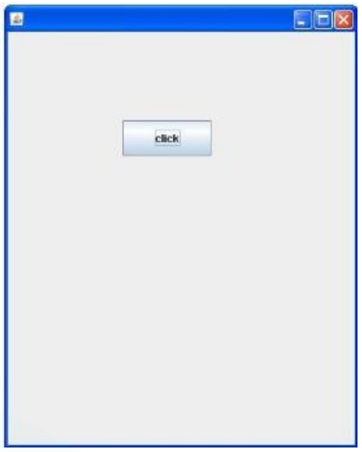
- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

//***FirstSwingExample.java***

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
```

```
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}
```



Swing Component Classes

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

JLabel

JLabel is the Swing component that creates a label, which is a component that displays information.

Label does not respond to user input. It just displays output.

JLabel extends JComponent.

JLabel constructors:

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:

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.

JLabel Example

JFrame: JFrame is the top-level container that is commonly used for Swing applications.

```
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);
   }
   }
```

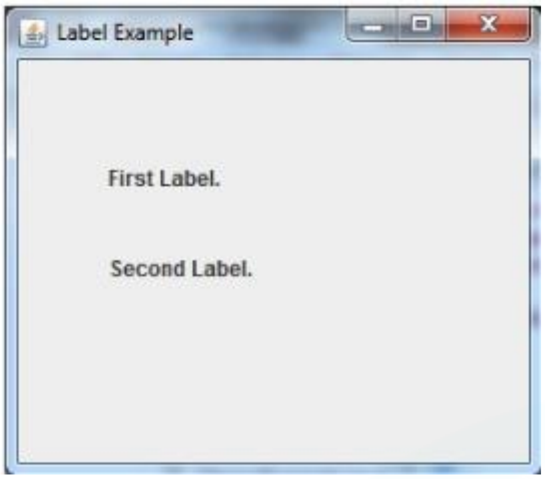
Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac LabelExample.java
```

To run the program, use this command line:

```
java LabelExample
```

Output:



JTextField

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

Commonly used Constructors:

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:

void addActionListener(ActionListener l) It is used to add the specified action listener to receive action events from this textfield.

Action getAction() It returns the currently set Action for this ActionEvent source, or null if no Action is set.

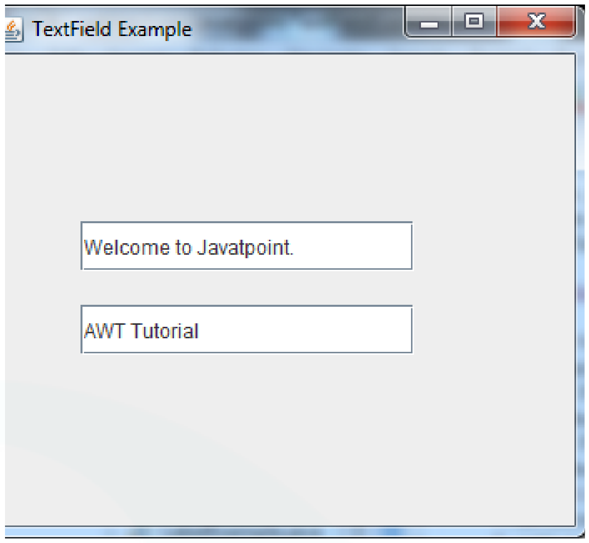
void setFont(Font f) It is used to set the current font.

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

JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
   {
```

```
JFrame f= new JFrame("TextField Example");
JTextField t1,t2;
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);
}
}
```



JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

Swing buttons provide features that are not found in the **Button** class defined by the AWT. For example,

Commonly used Constructors:

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:

void addActionListener(ActionListener l) It is used to add the specified action listener to receive action events from this textfield.

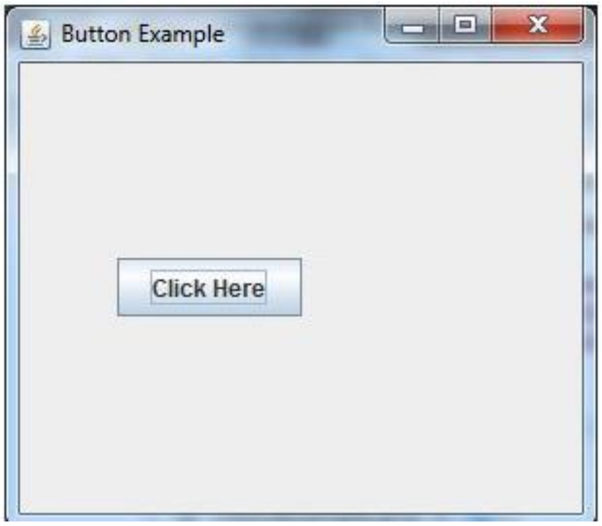
Action getAction() It returns the currently set Action for this ActionEvent source, or null if no Action is set.

void setFont(Font f) It is used to set the current font.

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

JButton Example

```
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);
        }
}
```

JTextArea

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

Commonly used Constructors:

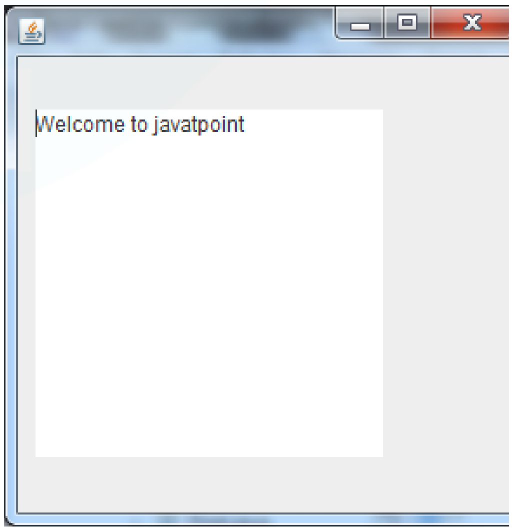
JTextArea()	Creates a text area that displays no text initially.
JTextArea(String s)	Creates a text area that displays specified text initially.
JTextArea(int row, int column)	Creates a text area with the specified number of rows and columns that displays no text initially.
JTextArea(String s, int row,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

Commonly used Methods:

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.

JTextArea Example

```
import javax.swing.*;
public class TextAreaExample
{
    TextAreaExample()
    {
        JFrame f= new JFrame();
        JTextArea 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);
    }
```


Welcome to javatpoint

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

Icons

In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image.

Two of its constructors are shown here:

ImageIcon(String filename): uses the image in the file named filename

ImageIcon(URL url): uses the image in the resource identified by url

The **ImageIcon** class implements the **Icon** interface that declares the methods shown here:

int getIconHeight( ): Returns the height of the icon in pixels

int getIconWidth( ): Returns the width of the icon in pixels.

void paintIcon(Component comp, Graphics g, int x, int y): Paints the icon at position x, y on the graphics context g. Additional information about the paint operation can be provided in comp.

Icons Example

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

JApplet

Fundamental to Swing is the **JApplet** class, which extends **Applet**.

Applets that use Swing must be subclasses of **JApplet**.

JApplet is rich with functionality that is not found in **Applet**. For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane.

When adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the content pane of the **JApplet** object.

The content pane can be obtained via the method shown here:

Container getContentPane( )

The **add( )** method of **Container** can be used to add a component to a content pane.

void add(comp) //Here, comp is the component to be added to the content pane.

Combo Boxes

Swing provides a combo box (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.

A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry.

You can also type your selection into the text field.

Commonly used Constructors:

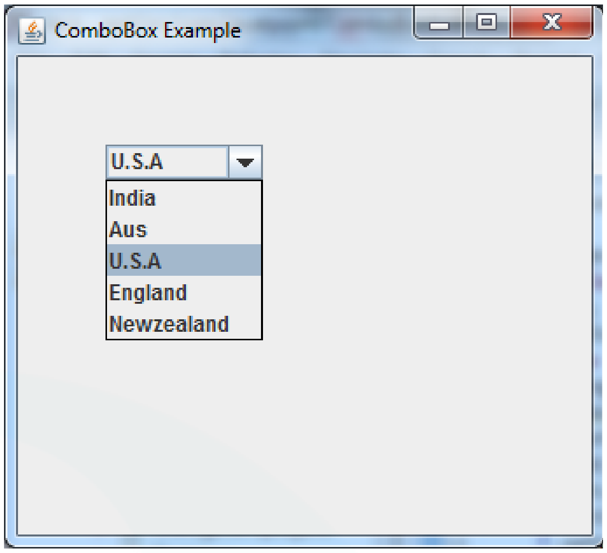
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array.
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector.

Commonly used Methods:

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.

JComboBox Example

```
import javax.swing.*;
public class ComboBoxExample
{
  JFrame f;
  ComboBoxExample()
  {
    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 ComboBoxExample();
  }
}
```



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.

Commonly used Constructors:

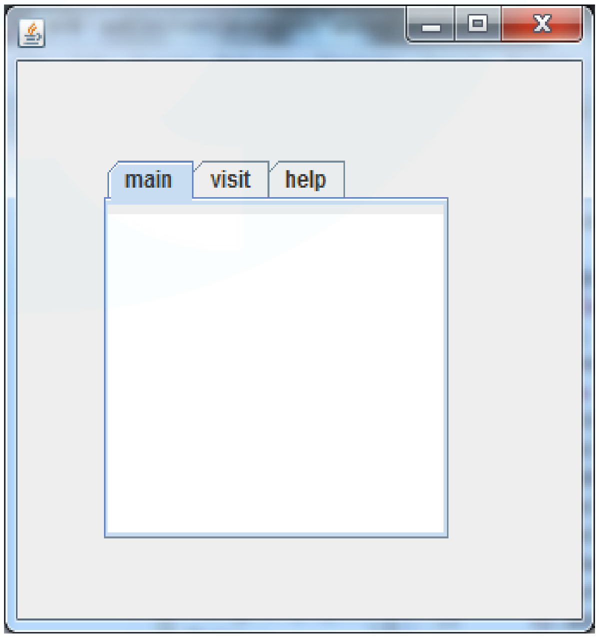
JTabbedPane()Creates an empty TabbedPane with a default tab placement of JTabbedPane.Top.

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.

JTabbedPane Example

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



```
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TabbedPaneExample();
}}
```

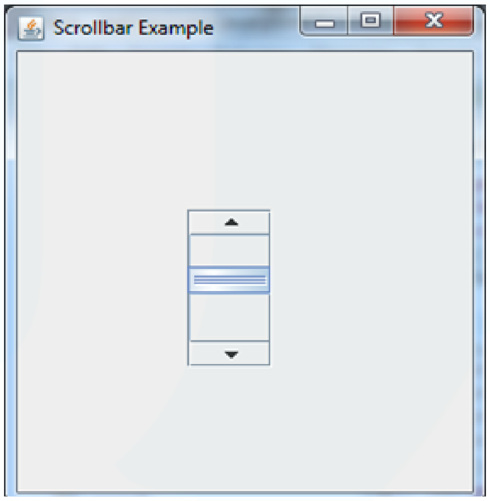
JScrollBar

The object of JScrollbar class is used to add horizontal and vertical scrollbar. It is an implementation of a scrollbar. It inherits JComponent class.

Commonly used Constructors:

JScrollBar()	Creates a vertical scrollbar with the initial values.
JScrollBar(int orientation)	Creates a scrollbar with the specified orientation and the initial values.
JScrollBar(int orientation, int value, int extent, int min, int max)	Creates a scrollbar with the specified orientation, value, extent, minimum, and maximum.

JScrollBar Example



```
import javax.swing.*;
class ScrollBarExample
{
ScrollBarExample(){
    JFrame f= new JFrame("Scrollbar Example");
 JScrollBar s=new JScrollBar();
s.setBounds(100,100, 50,100);
f.add(s);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[])
{
new ScrollBarExample();
}}
```

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:

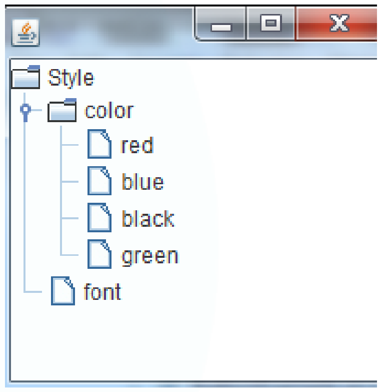
JTree() Creates a JTree with a sample model.

JTree(Object[] value) Creates a JTree with every element of the specified array as the child of a new root node.

JTree(TreeNode root) Creates a JTree with the specified TreeNode as its root, which displays the root node.

JTree Example

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample {
JFrame f;
TreeExample(){
   f=new JFrame();
   DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
   DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
   DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
   style.add(color);
   style.add(font);
   DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
   DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
   DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
   DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
   color.add(red); color.add(blue); color.add(black); color.add(green);
   JTree jt=new JTree(style);
   f.add(jt);
   f.setSize(200,200);
   f.setVisible(true);
}
public static void main(String[] args) {
   new TreeExample();
}}
```



JTable

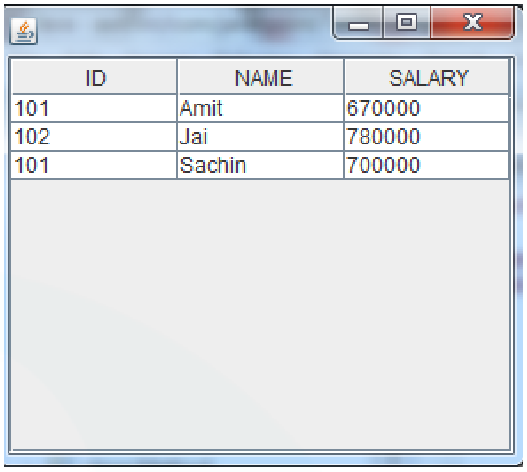
The JTable class is used to display data in tabular form. It is composed of rows and columns.

Commonly used Constructors:

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

JTable Example

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



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

Java JDBC

JDBC stands for Java Database Connectivity, which is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database.

JDBC provides a Java API for updating and querying relational databases using Structured Query Language (SQL)

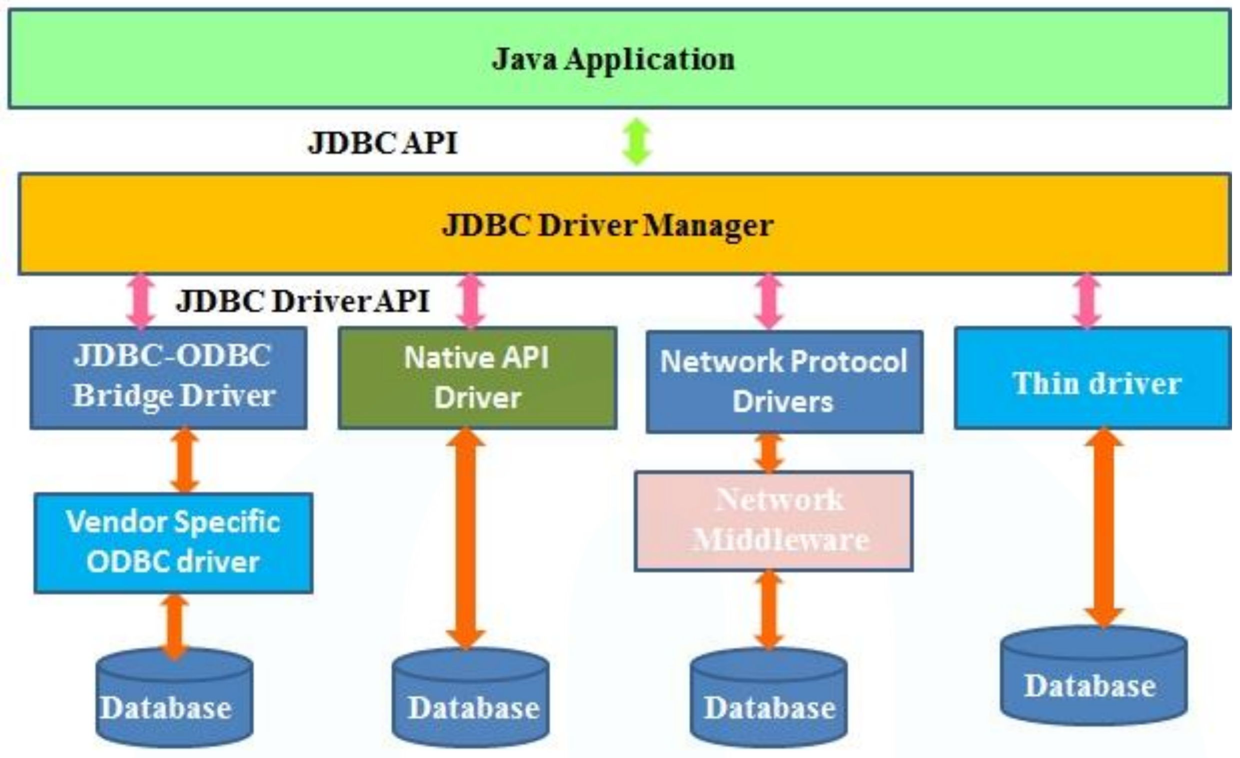
JDBC is independent of any database

JDBC is an Application Programming Interface (API) which consists of a set of Java Interfaces, Classes and JDBC drivers.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

JDBC Architecture



JDBC Architecture consists of two layers –

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- The JDBC API defines a set of Java interfaces that encapsulates the major database functionalities including configuring, running queries, processing results and integrating transactions
- The JDBC driver manager ensures that the correct driver is used to access each data source.
- Structured Query Language (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.

JDBC Driver

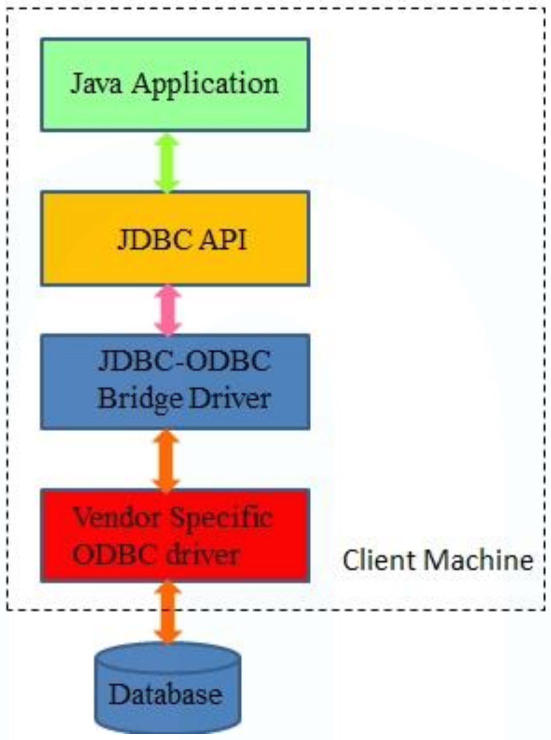
JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)

4. Thin driver (fully java driver)

Drivers are pluggable items so that existing drivers can be replaced by new drivers of the database without rewriting the application

1) JDBC-ODBC bridge driver

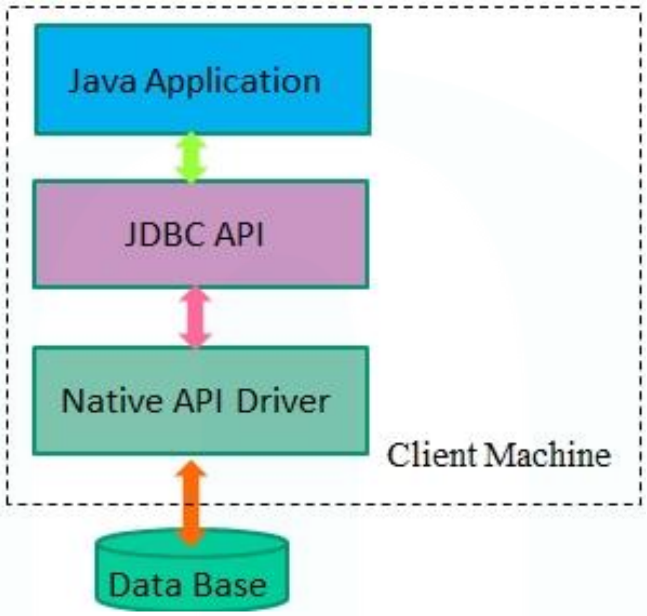


The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.

Advantages	Disadvantages
easy to use.	Performance degraded because JDBC method call is converted into the ODBC function calls.
can be easily connected to any database.	The ODBC driver needs to be installed on the client machine.
The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.	Its not portable because the Bridge driver is not written fully in Java.
	Not good for the Web

	The performance is not good because the JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process which makes this the slowest of all driver types

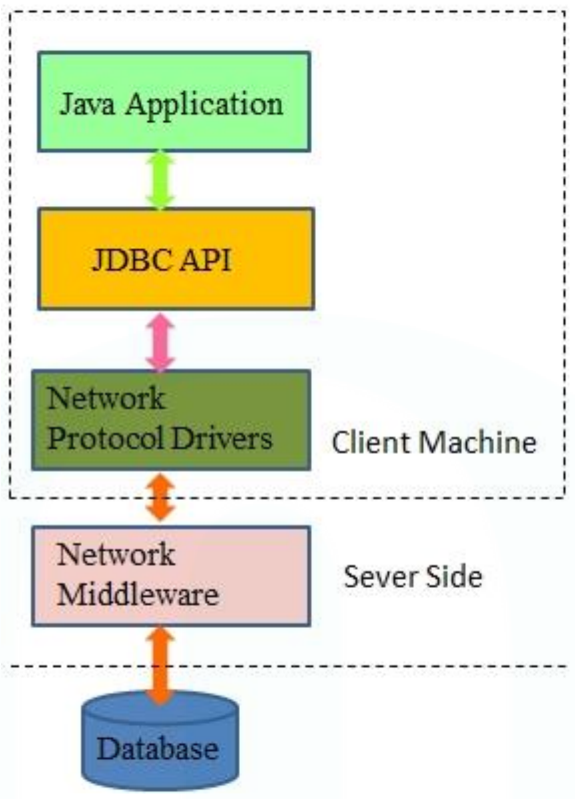
2) Native-API driver



The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Advantages	**Disadvantages**
performance upgraded than JDBC-ODBC bridge driver.	Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet
Offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type1 and also it uses Native API which is Database specific	The Vendor client library needs to be installed on client machine.
	Like Type 1 drivers, it's not written in Java Language which forms a portability issue
	If the Database is changed the native API must also be changed as it is specific to a database

3) Network Protocol driver



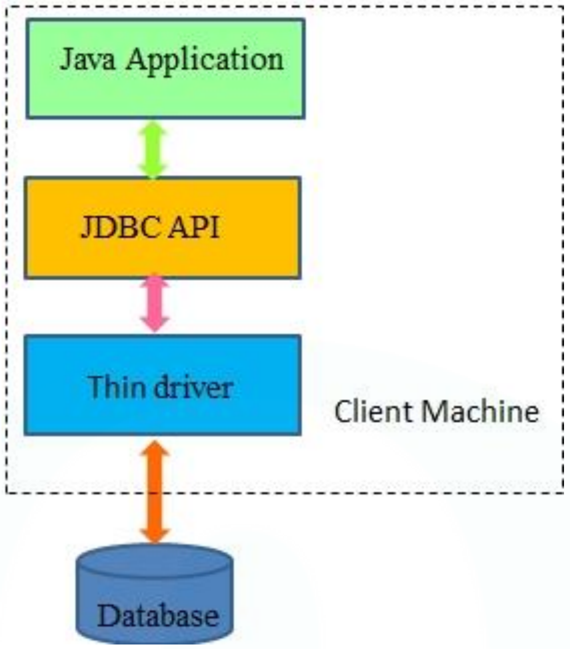
The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Type 3 database requests are passed through the network to the middle-tier server

The middle-tier then translates the request to the database

Advantages	Disadvantages
This driver is server-based so No client side library is required	Network support is required on client machine.
This driver is very flexible allows access to multiple databases using one driver	Requires database-specific coding to be done in the middle tier.
This driver is fully written in Java and hence Portable so it is suitable for the web	Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
There are many opportunities to optimize portability, performance, and scalability	It requires another server application to install and maintain

4) Thin driver



The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

uses java networking libraries to communicate directly with the database server

Advantages	Disadvantages
Better performance than all other drivers.	Drivers depends on the Database.
There is no need to install special software on the client or server	With type 4 drivers the user needs a different driver for each database
The major benefit of using a type 4 JDBC drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues so it is most suitable for the web	
Number of translation layers is very less because type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server so performance is typically quite good	
The drivers can be downloaded dynamically	

JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Creating JDBC Application

5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

1. Register the driver class
2. Creating connection
3. Creating statement
4. Executing queries
5. Closing connection

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Class.forName("oracle.jdbc.driver.OracleDriver");

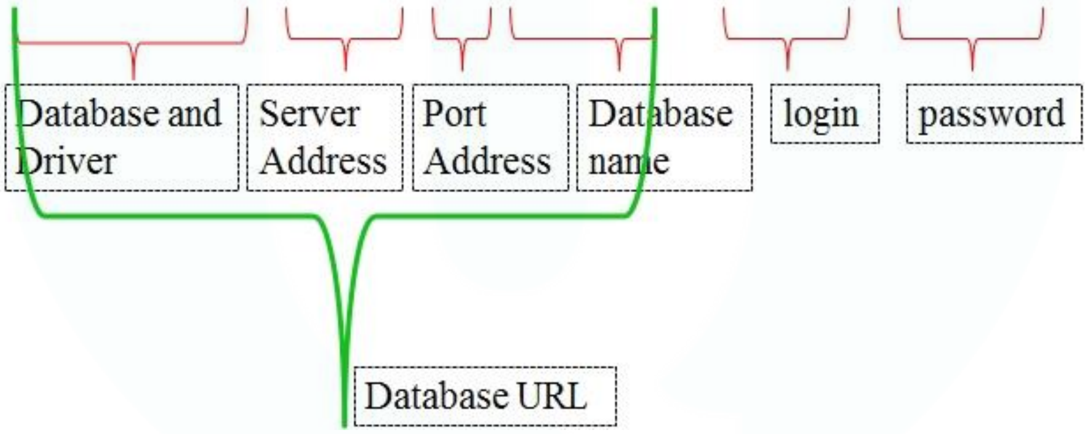
List of JDBC Drivers

Database Name	Driver
IBM DB2	COM.ibm.db2.jdbc.app.DB2Driver
JDBC-ODBC Bridge	sun.jdbc.odbc.JdbcOdbcDriver
MySQL	org.gjt.mm.mysql.Driver
Oracle 9i	oracle.jdbc.driver.OracleDriver
PostgreSQL	org.postgresql.Driver
Sybase	com.sybase.jdbc2.jdbc.SybDriver
Microsoft SQL Server 2000	com.microsoft.sqlserver.jdbc.SQLServerDriver
Cloudscape	COM.cloudscape.core.JDBCDriver
Firebird	org.firebirdsql.jdbc.FBDriver

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Connection con = DriverManager.getConnection("jdbc:postgresql://127.0.0.1:5432/COMPANY", "postgres“,"password");



3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

A Statement object is used to send SQL statements to the Database

First a statement object stmt is obtained from the Database connection object con

Statement stmt = con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

The Query returns a single ResultSet. To get more records from the database a loop like while loop can be used

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
{
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Process the results

- The ResultSet objects provide access to a table
- The objects provide access to the pseudo table that is the result of a SQL query
- The ResultSet objects maintain a cursor pointing to the current row of data This cursor initially points before the first row and is moved to the first row by the next() method

```
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
while (rs.next())
{
 System.out.println("EMPLOYEE ID:" + rs.getInt("ID") + "NAME:" + rs.getString("ENAME")
+ " AGE:"+ rs.getInt("EAGE")+ " SALARY:"+ rs.getFloat("ESALARY"));
}
```

The ResultSet object can return string, integer, float, boolean values according to the values declared in the database like given in the following table

Type	Method
Int	getInt( )
Float	getFloat( )
Boolean	getBoolean( )
Byte	getByte( )
String	getString( )
Long	getLong( )
Date	getDate( )

5) Close the connection object

- The last step in JDBC processing is closing the connection to the Database
- By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.
- Opening a new connection is typically much more expensive than sending queries on existing connections so postpone this step as long as possible
- Many JDBC drivers do automatic connection closing
- Database connection can be closed by the following statements

```
rs.close();
stmt.close();
con.close();
```

Example to connect to the Oracle database in java

For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following information for the oracle database:

Driver class: The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.

Connection URL: The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521: xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.

Username: The default username for the oracle database is **system**.

Password: Password is given by the user at the time of installing the oracle database.

First create a table in oracle database.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, system is the username and oracle is the password of the Oracle database.

```
import java.sql.*;
class OracleCon{
public static void main(String args[]){
```

```
try{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");
//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
//step3 create the statement object
Statement stmt=con.createStatement();
//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
//step5 close the connection object
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

DriverManager class

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Connection interface

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback() etc.
By default, connection commits the changes after executing queries

Statement interface

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

The important methods of Statement interface are as follows:
1) **public ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet.
2**) public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc.
3) **public boolean execute(String sql):** is used to execute queries that may return multiple results.
4) **public int[] executeBatch():** is used to execute batch of commands.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.
But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

Commonly used methods of ResultSet interface
1) **public boolean next():**is used to move the cursor to the one row next from the current position.
2) **public boolean previous():**is used to move the cursor to the one row previous from the current position.
3)**public boolean first():**is used to move the cursor to the first row in result set object.
4)**public boolean last():**is used to move the cursor to the last row in result set object.

Syntax: **public** ResultSetMetaData getMetaData()**throws** SQLException

// ResultSetMetaData rsmd=rs.getMetaData();

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.
If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface
public int getColumnCount()throws SQLException: it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException: it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException: it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException: it returns the table name for the specified column index.

Java Program to Update Records in a Table using JDBC

```
import java.sql.*;
public class JDBCExample
{
        static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
        static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";
        static final String USER = "username";
        static final String PASS = "password";
        public static void main(String[] args)
        {
                Connection conn = null;
                Statement stmt = null;
                try
                {
                        Class.forName("com.mysql.jdbc.Driver");
                        System.out.println("Connecting to a selected database...");
                        conn = DriverManager.getConnection(DB_URL, USER, PASS);
                        System.out.println("Connected database successfully...");
                        System.out.println("Creating statement...");
                        stmt = conn.createStatement();
                        String sql = "UPDATE Registration " +
                        "SET age = 30 WHERE id in (100, 101)";
                         stmt.executeUpdate(sql);
                        // Extract all the records to see the updated records
                         sql = "SELECT id, first, last, age FROM Registration";
                        ResultSet rs = stmt.executeQuery(sql);
                        while(rs.next())
                        {       //Retrieve by column name
                                 int id  = rs.getInt("id");
                                  int age = rs.getInt("age");
                                String first = rs.getString("first");
                                 String last = rs.getString("last");     //Display values
                                System.out.print("ID: " + id);
                                System.out.print(", Age: " + age);
                                System.out.print(", First: " + first);
                                System.out.println(", Last: " + last);
                        }
                         rs.close();
                }catch(SQLException se)
                {
                        se.printStackTrace();//Handle errors for JDBC
                         try
                        {
                                 if(conn!=null)
                                conn.close();
                        }catch(SQLException se){
                                se.printStackTrace();}
                }
        }
}
```