UNIT 1

SWING (LIGHT WEIGHT)

The Java Foundation Classes are a graphical framework for building portable Java-based graphical user interfaces. JFC consists of the Abstract Window Toolkit, Swing and Java 2D. JFC 1.2 is an extension of the AWT, not replacement of it.

AWT

Java Swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. Swing components facilitate efficient graphical user interface (GUI) development. Swing Components are collection of lightweight visual components they contain replacement for heavyweight AWT components as well as complex user-interface components such as tables and tress. Components contain a pluggable look and feel (PL&F) that allows application to run with native look and feel on different platforms. PL&F allows application to have the same behaviour on various platform. JFC contains operating system neutral look and feel. Swing component do not contain peers. Swing components allow mixing of AWT heavyweight and swing light weight components in an application. Heavyweight components have opaque pixels and are always rectangular whereas lightweight components have transparent pixels and are non-regular.

Swing component are JavaBean compliant. This allows components to be used easily in a Bean aware application building program. The root of the majority of the swing hierarchy is the Jcomponent class. The class is an extension of the AWT container class.

Features of Swing:

1. Platform Independent: It is platform independent, the swing components that are used to build the program are not platform specific. It can be used at any platform and anywhere.

2. Lightweight: Swing components are lightweight which helps in creating the UI lighter. Swings component allows it to plug into the operating system user interface framework that includes the

mappings for screens or device and other user interactions like key press and mouse movements.

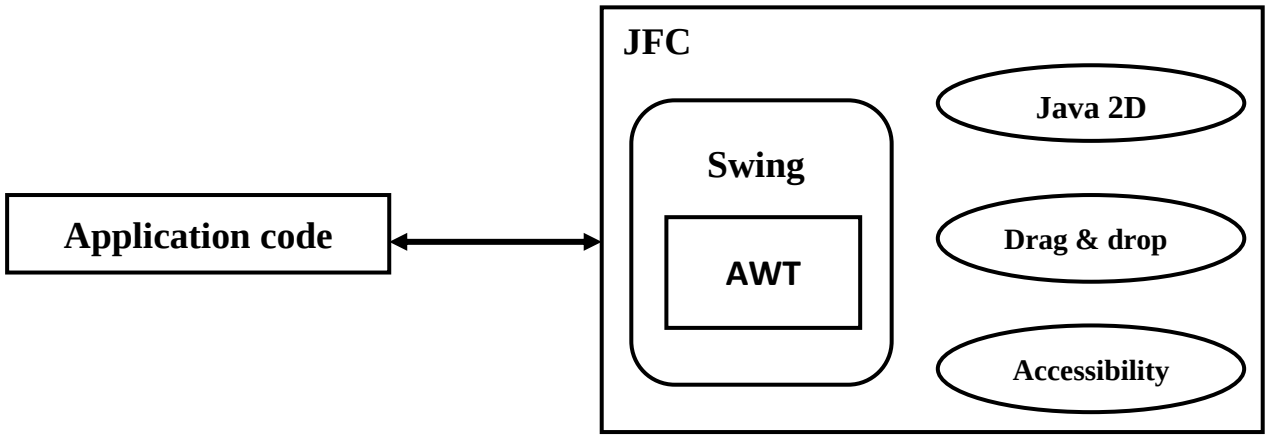
3. Plugging: It has a powerful component that can be extended to provide the support for the user interface that helps in good look and feel to the application. It refers to the highly modular-based architecture that allows it to plug into other customized implementations and framework for user interfaces. Its components are imported through a package called java.swing.

4. Manageable: It is easy to manage and configure. Its mechanism and composition pattern allows changing the settings at run time as well. The uniform changes can be provided to the user interface without doing any changes to application code.

5. MVC: They mainly follows the concept of MVC that is Model View Controller. With the help of this, we can do the changes in one component without impacting or touching other components. It is known as loosely coupled architecture as well.

6. Customizable: Swing controls can be easily customized. It can be changed and the visual appearance of the swing component application is independent of its internal representation.

The Swing architecture is shown in the figure given below:



Java Foundation Class:

The Java Foundation Classes (JFC) are a graphical framework for building portable Java-based graphical user interfaces (GUIs). JFC consists of the Abstract Window Toolkit (AWT), Swing and Java 2D. Together, they provide a consistent user interface for Java programs, regardless of whether the underlying user interface system is Windows, macOS or Linux

- **Swing**
  Swing is the part of the Java Foundation Classes (JFC) that implements a new set of GUI components with a pluggable look and feel. Swing is implemented in 100% Pure Java, and is based on the JDK 1.1 Lightweight UI Framework. The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Windows, Solaris, Macintosh). Swing components include both 100% Pure Java versions of the existing AWT component set (Button, Scrollbar, Label, etc.), plus a rich set of higher-level components (such as tree view, list box, and tabbed panes).

- **Java 2D**
  The Java 2D API is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators. These classes are provided as additions to the java.awt and java.awt.image packages (rather than as a separate package).

- **Accessibility**
  Through the Java Accessibility API, developers will be able to

create Java applications that can interact with assistive technologies such as screen readers, speech recognition systems and Braille terminals. Accessibility enabled Java applications are not dependent on machines that require assistive technology support, rather these applications will run on any Java-enabled machine with or without assistive technologies.

The Java Accessibility API is one of the core foundation services in the Java Foundation Classes, a comprehensive set of graphical user interface components and foundation services designed to simplify deployment of Internet, intranet and desktop applications.

- **Drag and Drop**
  Drag and Drop enables data transfer both across Java and native applications, between Java applications, and within a single Java application.

Difference between Swing and AWT

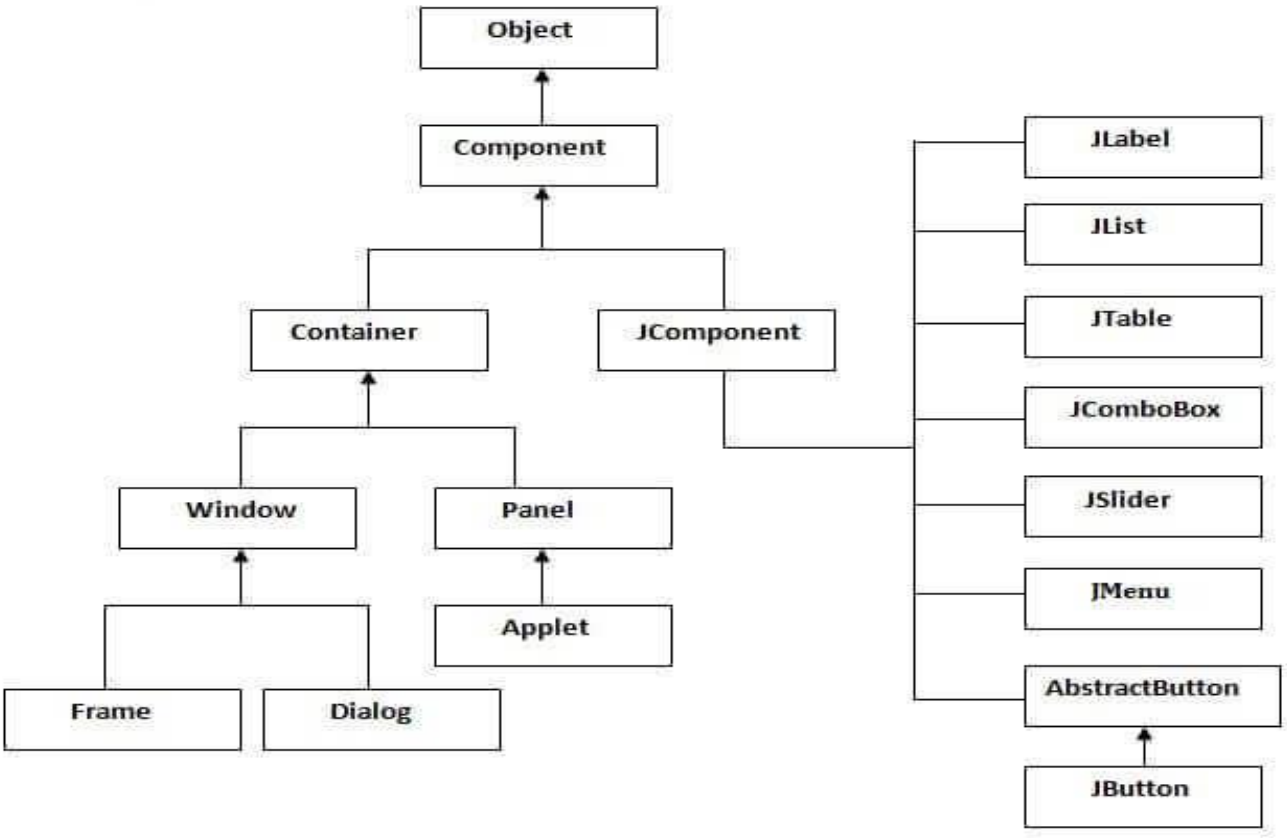
Swing	AWT
Java swing components are platform-independent	AWT components are platform-dependent
Swing components are lightweight.	AWT components are heavyweight.
Swing supports pluggable look and feel.	AWT doesn't support pluggable look and feel.
Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.	AWT provides less components than Swing.
Swing follows MVC.	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.

Component Hierarchy

JComponent Class is the root of visual component class hierarchy in JFC. The visual components are known as the "J" classes. The functionality contained in the JComponent class is available to all the visual components contained in the JFC. The JComponent class is repository of functionality for all visual component.

The JComponent class is at the top of the hierarchy of all visual components contained in the JFC. The hierarchy is shown in the following figure:



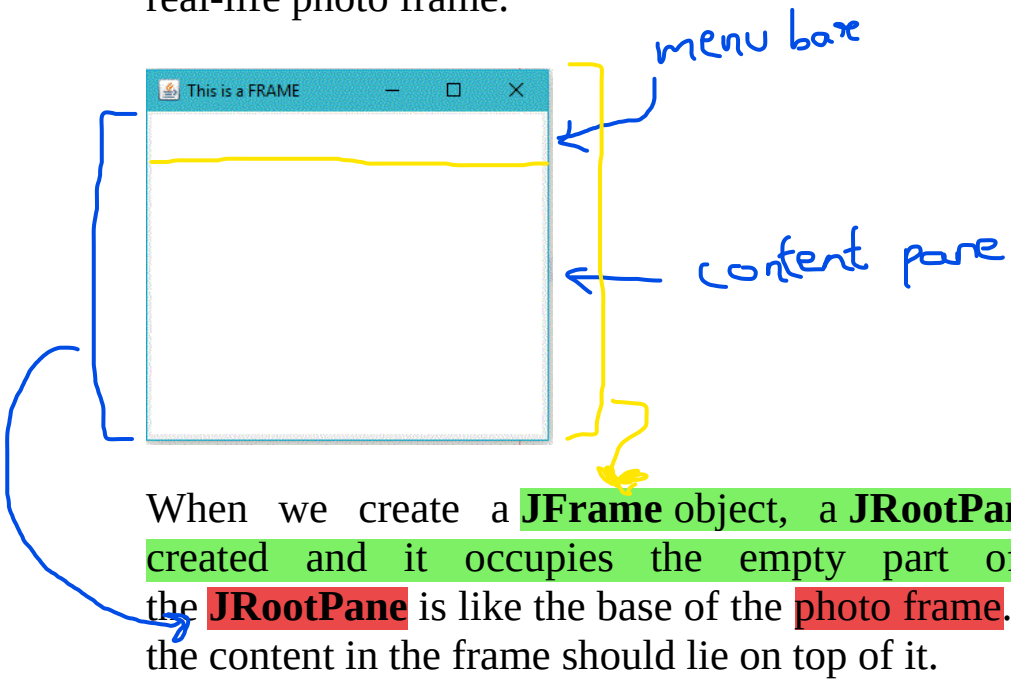
Panes:

There are 4 types of panes

- RootPane
- GlassPane
- LayeredPane
- ContentPane

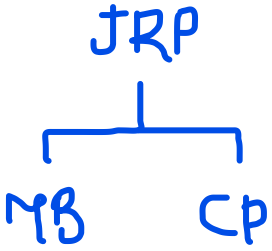
What is RootPane, GlassPane and ContentPane?

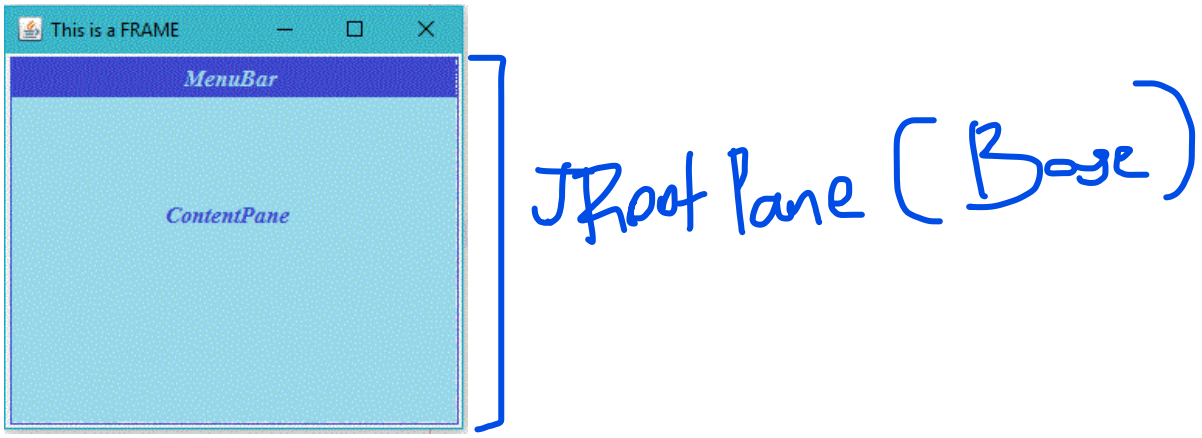
When you look at any window in a computer, the outer part containing the minimize, restore and close buttons and the overall border can be thought of as a frame. The frame that we are imaging is still hollow! That means there is nothing in the middle. So, the frame is similar to a real-life photo frame.



When we create a **JFrame** object, a **JRootPane** is automatically created and it occupies the empty part of the frame. So, the **JRootPane** is like the base of the photo frame. This means that all the content in the frame should lie on top of it.

Now the **JRootPane** has two children. There is an optional **MenuBar** and a **ContentPane**. The menu bar occupies the upper part while the rest of the part is used by the **ContentPane**.





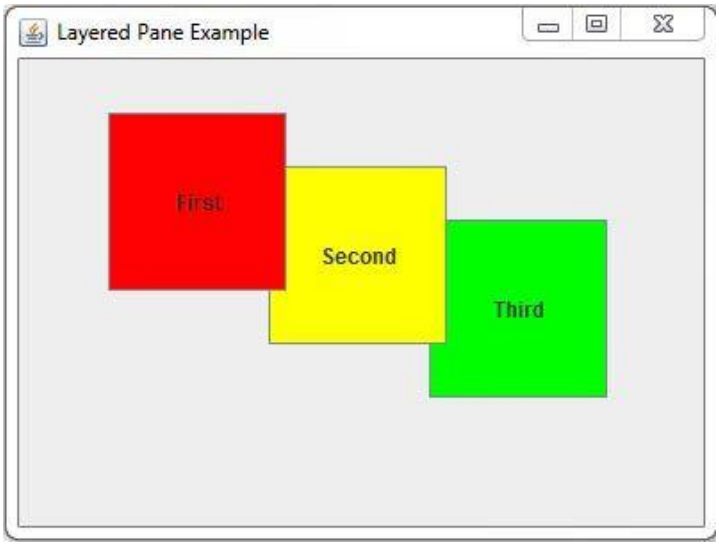
As the name suggests, all the contents that we might add will be placed on the **ContentPane**. Hence, the **ContentPane** is very similar to the photograph in the photo frame. It holds the visible content. Whenever we add a component to a **JFrame** using a method like *frame.add(component)*, the **ContentPane** is implicitly called and the component is added to it. As the **ContentPane** has no default layout, the component takes up the entire area of the component pane. In case we add a new component, the old component is overlapped by the new one. Finally, we have the **GlassPane**. It covers the entire visible area of the root pane. It behaves just like a sheet of glass and is completely transparent unless we implement the glass pane's *paintComponent* method. The glass pane can also perform complex tasks like intercepting the mouse pointer etc. It is similar to the glass on the photo frame which is invisible and lies on top of the photograph.

We can summarize this discussion as follows. The **JFrame** object has a **RootPane** at its middle. This root pane acts as a base for the **MenuBar** and the **ContentPane**. All the content is placed on this content pane. Finally, a transparent **GlassPane** covers the entire visible area of the RootPane.

What are LayeredPane?

A layered pane is a Swing container which is used to hold the various components using the concept of layers. The components present in the

upper layer overlaps the components present in the lower layer. The layered pane is created using the JLayeredPane class. the only constructor of this class is JLayeredPane ().



Example for JRootPane

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JRootPane;

public class JRootPaneExample {
    public static void main(String[] args) {
        JFrame f = new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JRootPane root = f.getRootPane();

        // Create a menu bar
        JMenuBar bar = new JMenuBar();
        JMenu menu = new JMenu("File");
        bar.add(menu);
        menu.add("Open");
        menu.add("Close");
        root.setJMenuBar(bar);
```

```
        // Add a button to the content pane
        root.getContentPane().add(new JButton("Press Me"));

        // Display the UI
        f.pack();
        f.setVisible(true);
      }
}
```

Example for JContentPane and JGlassPane

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

public class RPane extends JFrame{
    public static void main(String[] args) {

//Creating a frame
    JFrame frame = new JFrame("This is a Frame");
    frame.setLayout(new FlowLayout());
    frame.setSize(279,200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
//Creating sample content
    JButton click = new JButton("click!");

    JTextField content = new JTextField (30);
    content.setText(" Basic Pane Example ");
//frame. getContentPane () returns the underlying ContentPane
    JPanel panel = (JPanel)frame.getContentPane();
    panel.setLayout(new BorderLayout());
    panel.add(content,BorderLayout.CENTER);

//frame. getGlassPane () returns the underlying GlassPane
```

```
        JPanel glass = (JPanel)frame.getGlassPane();

        glass.setVisible(true);
        glass.setLayout(new BorderLayout());
        glass.add(click, BorderLayout.SOUTH);

        click.addActionListener(e -> {
                if(panel.isVisible())
                {
                        panel.setVisible(false);
                }
                 else
                {
                        panel.setVisible(true);
                }
      });
   }
}
```

Swing Components

Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. All these components are lightweight components. They all are derived from **JComponent** class. It supports the pluggable look and feel.

Different types of Swing Components are:

- JLabel
- JTextfield
- JPasswordfield
- JTextarea
- JButton
- JCheckbox
- JRadiobutton
- JCombobox
- JList

Classes

Java JLabel:

JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus . JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.

Constructor of the class are :

1. **JLabel() :** creates a blank label with no text or image in it.
2. **JLabel(String s) :** creates a new label with the string specified.
3. **JLabel(Icon i) :** creates a new label with a image on it.
4. **JLabel(String s, Icon i, int align) :** creates a new label with a string, an image and a specified horizontal alignment

Commonly used methods of the class are :

1. **getIcon() :** returns the image that that the label displays
2. **setIcon(Icon i) :** sets the icon that the label will display to image i
3. **getText() :** returns the text that the label will display
4. **setText(String s) :** sets the text that the label will display to string s

JTextfield:

JTextField is a part of javax.swing package. The class JTextField is a component that allows editing of a single line of text. JTextField inherits the JTextComponent class and uses the interface SwingConstants.

The constructor of the class are :

JTextField() : constructor that creates a new TextField

JTextField(int columns) : constructor that creates a new empty TextField with specified number of columns.

JTextField(String text) : constructor that creates a new empty text field initialized with the given string.

JTextField(String text, int columns) : constructor that creates a new empty textField with the given string and a specified number of columns .

JTextField(Document doc, String text, int columns) : constructor that creates a textfield that uses the given text storage model and the given number of columns.

Methods of the JTextField are:
setColumns(int n) :set the number of columns of the text field.
setFont(Font f) : set the font of text displayed in text field.
addActionListener(ActionListener l) : set an ActionListener to the text field.
int getColumns() :get the number of columns in the textfield.

Java JPasswordField:
PasswordField is a part of javax.swing package . The class JPasswordField is a component that allows editing of a single line of text where the view indicates that something was typed by does not show the actual characters. JPasswordField inherits the JTextField class in javax.swing package.

Constructors of the class are :

1. **JPasswordField()**: constructor that creates a new PasswordField
2. **JPasswordField(int columns)** : constructor that creates a new empty PasswordField with specified number of columns.
3. **JPasswordField(String Password)** : constructor that creates a new empty Password field initialized with the given string.
4. **JPasswordField(String Password, int columns)** : constructor that creates a new empty PasswordField with the given string and a specified number of columns .
5. **JPasswordField(Document doc, String Password, int columns)** : constructor that creates a Passwordfield that uses the given text storage model and the given number of columns.

Commonly used method of JPasswordField :

1. **char getEchoChar()** : returns the character used for echoing in JPasswordField.
2. **setEchoChar(char c)** : set the echo character for JPasswordField.

3. **String getPassword()** : returns the text contained in JPasswordField.
4. **String getText()** : returns the text contained in JPasswordField.

Example for JLable, JTextfield and JPasswordfield is given below:

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

public class PassDemo extends JPanel
{

// create an object of the JLabel class
        JLabel lblName;
        JLabel lblPasswd;

// create an object of the JPassword class
        JTextField txtName;
        JPasswordField txtPasswd;

        public PassDemo()   constructor
        {
                lblName = new JLabel("Enter the User Name: ");
                txtName = new JTextField(10);
                lblPasswd = new JLabel("Enter the Password: ");
                txtPasswd = new JPasswordField(10);
                txtPasswd.setEchoChar('*');

        // Add tooltips to the text fields
                txtName.setToolTipText("Enter User Name");
                txtPasswd.setToolTipText("Enter Password");

        //Add labels to the Panel.
                add(lblName);
                add(txtName);
                add(lblPasswd);
```

```
            add(txtPasswd);
        }

        public static void main(String[] args)
        {
        // calls the PassDemo constructor.
            PassDemo demo = new PassDemo();

        // set the text on the frame
            JFrame frm = new JFrame("Password Demo");
            frm.setContentPane(demo);

        /* setSize() method is used to specify the width and height of the
frame */
            frm.setSize(275,300);

        // To display the Frame
            frm.setVisible(true);
            WindowListener listener = new WindowAdapter()
            {
                public void windowClosing(WindowEvent winEvt)
                {
                    System.exit(0);
                }
            };  // End of WindowAdaptor() method

        // Window listener activates the windowClosing() method
            frm.addWindowListener(listener);
        }  // End of main() method
}       // End of class declaration
```

JTextArea:

JTextArea is a part of java Swing package . It represents a multi line area that displays text. It is used to edit the text .
JTextArea inherits JComponent class. The text in JTextArea can be

set to different available fonts and can be appended to new text . A text area can be customized to the need of user .

Constructors of JTextArea are:

1. **JTextArea() :** constructs a new blank text area .
2. **JTextArea(String s) :** constructs a new text area with a given initial text.
3. **JTextArea(int row, int column) :** constructs a new text area with a given number of rows and columns.
4. **JTextArea(String s, int row, int column) :** constructs a new text area with a given number of rows and columns and a given initial text.

Commonly used methods :

1. **append(String s) :** appends the given string to the text of the text area.
2. **getLineCount() :** get number of lines in the text of text area.
3. **setFont(Font f) :** sets the font of text area to the given font.
4. **setColumns(int c) :** sets the number of columns of the text area to given integer.
5. **setRows(int r) :** sets the number of rows of the text area to given integer.
6. **getColumns() :** get the number of columns of text area.
7. **getRows() :** get the number of rows of text area.

JCheckbox:

JCheckBox is a part of Java Swing package . JCheckBox can be selected or deselected . It displays it state to the user . JCheckBox is an implementation to checkbox . JCheckBox inherits JToggleButton class.

Constructor of the class are :

1. **JCheckBox()** : creates a new checkbox with no text or icon
2. **JCheckBox(Icon i)** : creates a new checkbox with the icon specified
3. **JCheckBox(Icon icon, boolean s)** : creates a new checkbox with the icon specified and the boolean value specifies whether it is selected or not.

4. **JCheckBox(String t)** :creates a new checkbox with the string specified
5. **JCheckBox(String text, boolean selected)** :creates a new checkbox with the string specified and the boolean value specifies whether it is selected or not.
6. **JCheckBox(String text, Icon icon)** :creates a new checkbox with the string and the icon specified.
7. **JCheckBox(String text, Icon icon, boolean selected)**: creates a new checkbox with the string and the icon specified and the boolean value specifies whether it is selected or not.

Methods to add Item Listener to checkbox.

1. **addActionListener(ItemListener l)**: adds item listener to the component
2. **itemStateChanged(ItemEvent e)** : abstract function invoked when the state of the item to which listener is applied changes
3. **getItem()** : Returns the component-specific object associated with the item whose state changed
4. **getStateChange()** : Returns the new state of the item. The ItemEvent class defines two states: SELECTED and DESELECTED.
5. **getSource()** : Returns the component that fired the item event.

Commonly used methods:

1. **setIcon(Icon i)** : sets the icon of the checkbox to the given icon
2. **setText(String s)** :sets the text of the checkbox to the given text
3. **setSelected(boolean b)** : sets the checkbox to selected if boolean value passed is true or vice versa
4. **getIcon()** : returns the image of the checkbox
5. **getText()** : returns the text of the checkbox
6. **updateUI()** : resets the UI property with a value from the current look and feel.
7. **getUI()** : returns the look and feel object that renders this component.
8. **paramString()** : returns a string representation of this JCheckBox.
9. **getUIClassID()** : returns the name of the Look and feel class that renders this component.

10. **getAccessibleContext()** : gets the AccessibleContext associated with this JCheckBox.
11. **isBorderPaintedFlat()** : gets the value of the borderPaintedFlat property.
12. **setBorderPaintedFlat(boolean b)** : sets the borderPaintedFlat property

JRadioButton:

We use the JRadioButton class to create a radio button. Radio button is use to select one option from multiple options. It is used in filling forms, online objective papers and quiz.

We add radio buttons in a ButtonGroup so that we can select only one radio button at a time. We use "ButtonGroup" class to create a ButtonGroup and add radio button in a group.

Methods Used :

1. JRadioButton() : Creates a unselected RadioButton with no text.
   Example:
2. JRadioButton j1 = new JRadioButton()
3. JButton(String s) : Creates a JButton with a specific text.
   Example:
4. JButton b1 = new JButton("Button")
5. JLabel(String s) : Creates a JLabel with a specific text.
   Example:
6. JLabel L = new JLabel("Label 1")
7. ButtonGroup() : Use to create a group, in which we can add JRadioButton. We can select only one JRadioButton in a ButtonGroup.
   Steps to Group the radio buttons together.
   - Create a ButtonGroup instance by using "ButtonGroup()" Method.
   - ButtonGroup G = new ButtonGroup()
   - Now add buttons in a Group "G", with the help of "add()" Method.

Example:

```
G.add(Button1);
G.add(Button2);
```

8. isSelected() : it will return a Boolean value true or false, if a JRadioButton is selected it Will return true otherwise false.
   Example:

```
JRadioButton.isSelected()
```

9. Set(…) and Get(…) Methods :
   i) Set and get are used to replace directly accessing member variables from external classes.
   ii) Instead of accessing class member variables directly, you define get methods to access these variables, and set methods to modify them.

JComboBox:

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

Constructor of the JComboBox are:

1. **JComboBox()** : creates a new empty JComboBox .
2. **JComboBox(ComboBoxModel M)** : creates a new JComboBox with items from specified ComboBoxModel
3. **JComboBox(E [ ] i)** : creates a new JComboBox with items from specified array.
4. **JComboBox(Vector items)** : creates a new JComboBox with items from the specified vector

Commonly used Methods are :

1. **addItem(E item)** : adds the item to the JComboBox
2. **addItemListener(ItemListener l)** : adds a ItemListener to JComboBox
3. **getItemAt(int i)** : returns the item at index i
4. **getItemCount()**: returns the number of items from the list
5. **getSelectedItem()** : returns the item which is selected

6. **removeItemAt(int i)** : removes the element at index i
7. **setEditable(boolean b)** : the boolean b determines whether the combo box is editable or not .If true is passed then the combo box is editable or vice versa.
8. **setSelectedIndex(int i)**: selects the element of JComboBox at index i.
9. **showPopup()** :causes the combo box to display its popup window.
10. **setUI(ComboBoxUI ui)**: sets the L&F object that renders this component.
11. **setSelectedItem(Object a)**: sets the selected item in the combo box display area to the object in the argument.
12. **setSelectedIndex(int a)**: selects the item at index anIndex.
13. **setPopupVisible(boolean v)**: sets the visibility of the popup.
14. **setModel(ComboBoxModel a)** : sets the data model that the JComboBox uses to obtain the list of items.
15. **setMaximumRowCount(int count)**: sets the maximum number of rows the JComboBox displays.
16. **setEnabled(boolean b)**: enables the combo box so that items can be selected.
17. **removeItem(Object anObject)** : removes an item from the item list.
18. **removeAllItems()**: removes all items from the item list.
19. **removeActionListener(ActionListener l)**: removes an ActionListener.
20. **isPopupVisible()** : determines the visibility of the popup.

JList:

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

Constructor for JList are :

1. **JList()**: creates an empty blank list
2. **JList(E [ ] l)** : creates an new list with the elements of the array.

3. **JList(ListModel d)**: creates a new list with the specified List Model
4. **JList(Vector l)** : creates a new list with the elements of the vector

Commonly used methods are :

1. **getSelectedIndex():** returns the index of selected item of the list.
2. **getSelectedValue():** returns the selected value of the element of the list
3. **setSelectedIndex(int i):** sets the selected index of the list to i
4. **setSelectionBackground(Color c):** sets the background Color of the list.
5. **setSelectionForeground(Color c):** Changes the foreground color of the list
6. **setVisibleRowCount(int v):** Changes the visibleRowCount property
7. **setSelectedValue(Object a, boolean s):** selects the specified object from the list.
8. **setSelectedIndices(int[] i):** changes the selection to be the set of indices specified by the given array.
9. **setFixedCellWidth(int w):** Changes the cell width of list to the value passed as parameter.

JDBC (Java DataBase Connectivity):

What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

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.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

JDBC Architecture

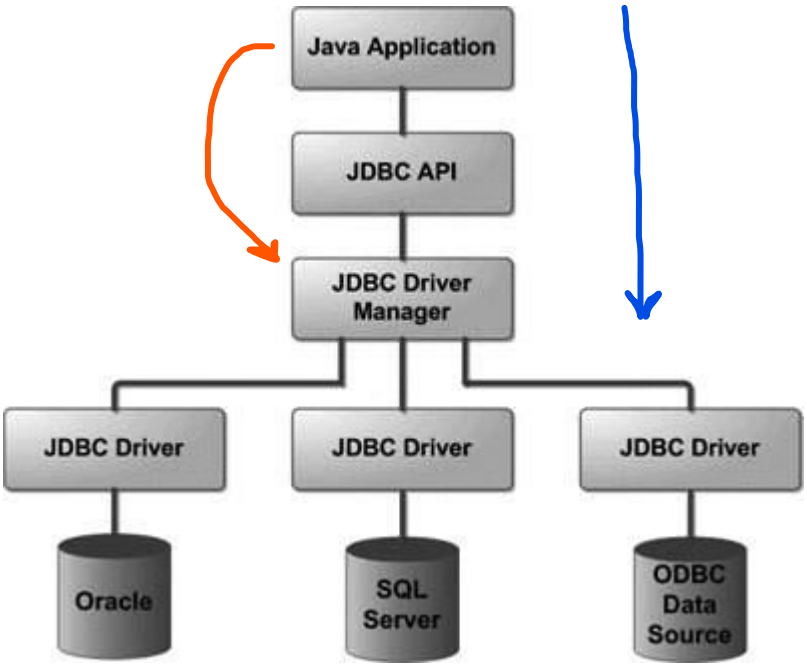
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **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 driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −

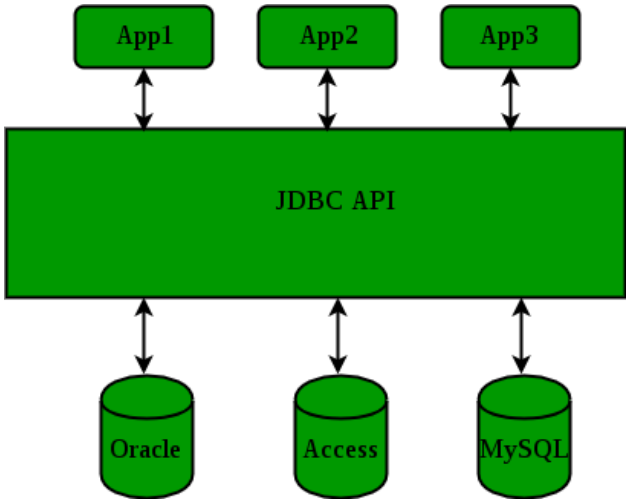


Common JDBC Components

The JDBC API provides the following interfaces and classes −

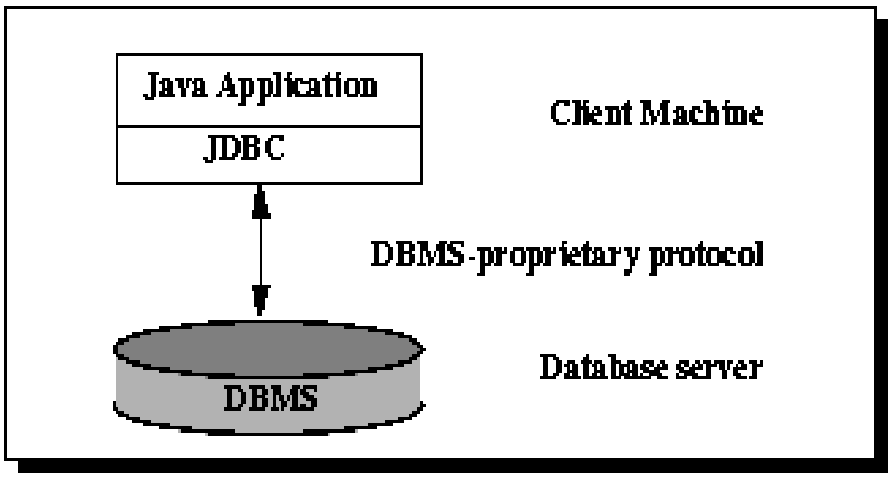
- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **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.

Structure of JDBC



Two-tier Architecture

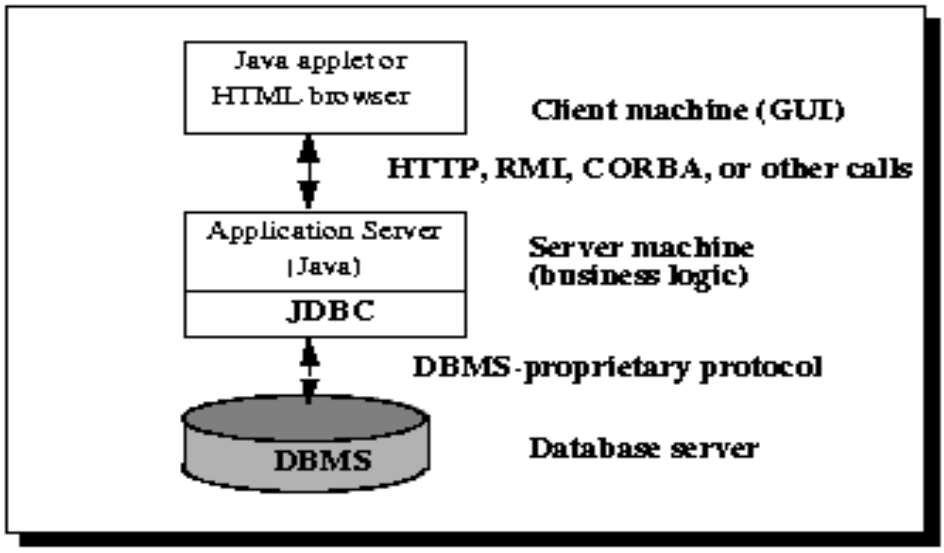
In the two-tier model, a Java application talks directly to the data source.
This requires a JDBC driver that can communicate with the particular data source being accessed.
A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.
The data source may be located on another machine to which the user is connected via a network.
This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server.



Three-tier Architecture

With the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans, the Java platform is fast becoming the standard platform for middle-tier development.
This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.
With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture.

Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets.
The JDBC API is also what allows access to a data source from a Java middle tier.



JDBC Drivers:

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

Type-1 driver or JDBC-ODBC bridge driver

Type-1 driver or 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. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

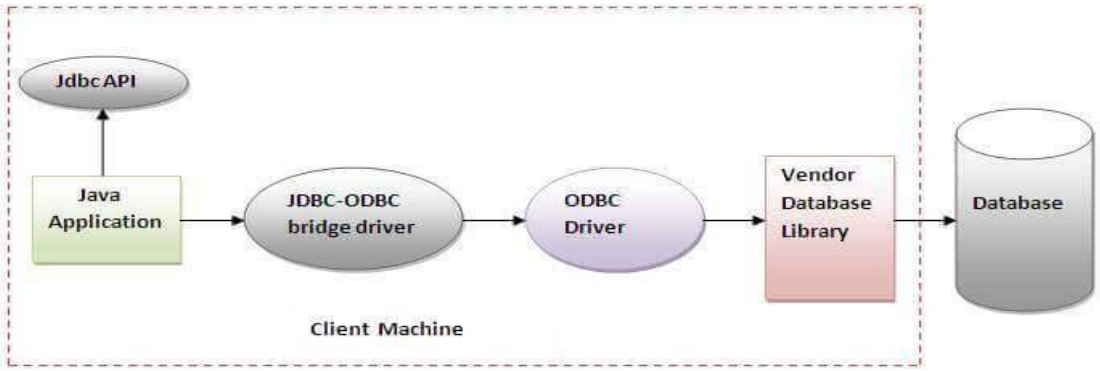


Figure- JDBC-ODBC Bridge Driver

Type-2 driver or Native-API driver

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 driver.

- Driver needs to be installed separately in individual client machines
- The Vendor client library needs to be installed on client machine.
- Type-2 driver isn't written in java, that's why it isn't a portable driver
- It is a database dependent driver.

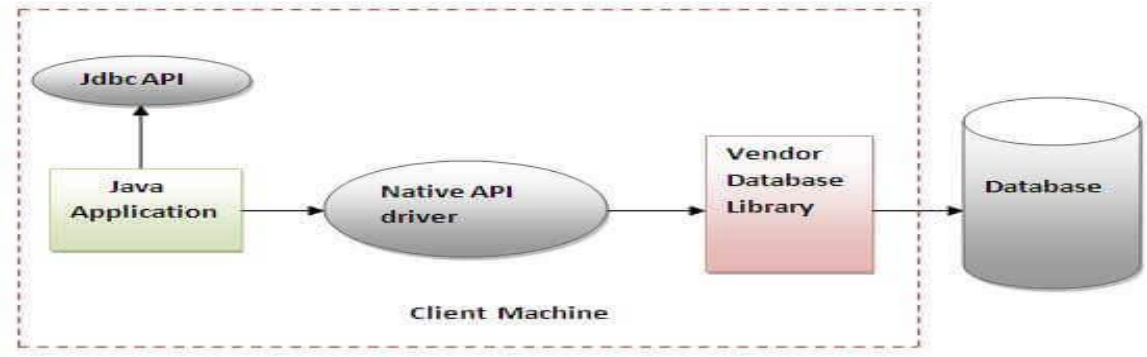


Figure- Native API Driver

Type-3 driver or Network Protocol driver Portable

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
- Switch facility to switch over from one database to another database.

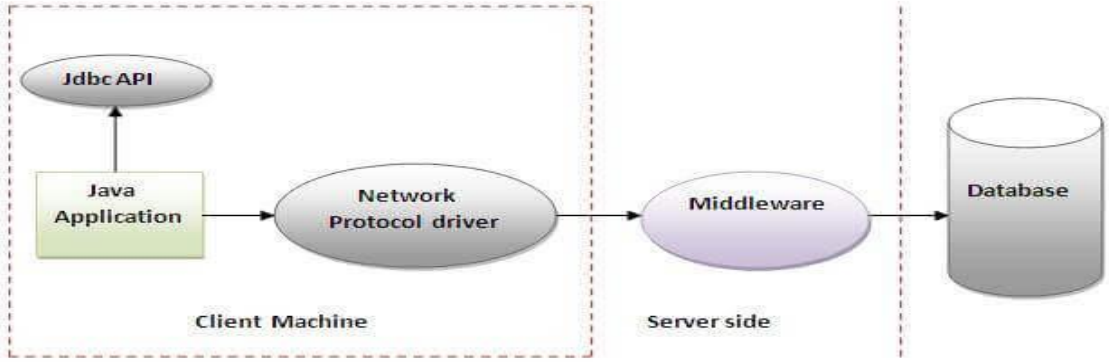


Figure- Network Protocol Driver

Type-4 driver or Thin Driver

portable

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

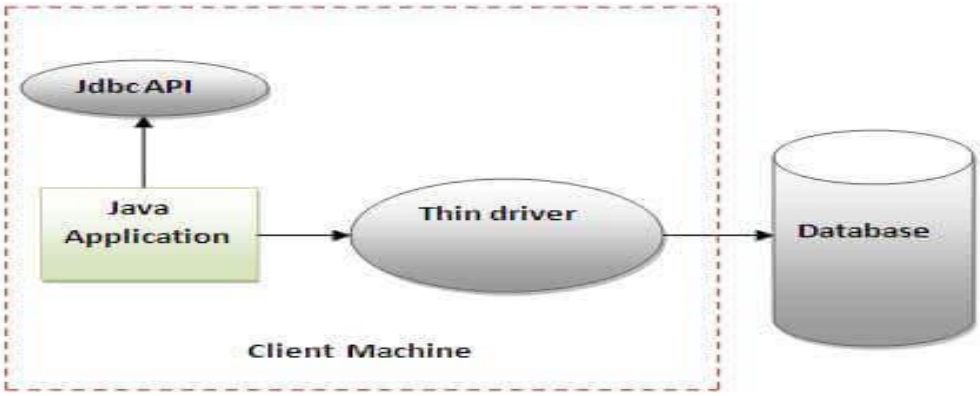


Figure- Thin Driver

JDBC Statements:

Once a connection is obtained we can interact with the database. The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
```

```
  . . .
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
  stmt = conn.createStatement( );
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  stmt.close();
}
```

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
```

```
catch (SQLException e) {
  . . .
}
finally {
  pstmt.close();
}
```

The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure −

```
CREATE OR REPLACE PROCEDURE getEmpName
  (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END;
```

NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database −

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
  (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

pstmt → IN

cstmt → IN, OUT, INOUT

Here are the definitions of each −

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure −

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

ResultSet:

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories –

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet –

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by

	others to the database that occur after the result set was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

```
try {
  Statement stmt = conn.createStatement(
                   ResultSet.TYPE_FORWARD_ONLY,
                   ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
  ....
}
finally {
  ....
}
```

Connection Modes

Connection with URL

```
String url      = "jdbc:h2:~/test";   //database specific url.
Connection connection =
DriverManager.getConnection(url);
```

2) **Connection with URL, user, password**

```
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";
Connection connection =
DriverManager.getConnection(url, user, password);
```

3) **Connection With URL and Properties**

```
String url      = "jdbc:h2:~/test";   //database specific url.

Properties properties = new Properties( );
properties.put( "user", "sa" );
properties.put( "password", "" );

Connection connection =
DriverManager.getConnection(url, properties);
```

4) **Connection via Try-With-Resources**

```
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";

try(Connection connection =
DriverManager.getConnection(url, user, password)) {

//use the JDBC Connection inhere
}
```

JDBC Transactions

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions −

- To increase performance.
- To maintain the integrity of business processes.
- To use distributed transactions.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit −

```
conn.setAutoCommit(false);
```

Commit & Rollback

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows −

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code −

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

For a better understanding, let us study the Commit - Example Code.

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints −

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object −

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees " +
                "VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   // If there is no error, commit the changes.
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

In this case, none of the above INSERT statement would success and everything would be rolled back.

Batch Updations:

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The **addBatch()** method of *Statement*, *PreparedStatement*, and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

- Create a Statement object using either *createStatement()* methods.
- Set auto-commit to false using *setAutoCommit()*. false
- Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.
- Execute all the SQL statements using *executeBatch()* method on created statement object.
- Finally, commit all the changes using *commit()* method.

Example

The following code snippet provides an example of a batch update using Statement object −

```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
             "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

For a better understanding, let us study the Batching - Example Code.

Batching with PrepareStatement Object

Here is a typical sequence of steps to use Batch Processing with PrepareStatement Object −

1. Create SQL statements with placeholders.
2. Create PrepareStatement object using either *prepareStatement()* methods.
3. Set auto-commit to false using *setAutoCommit()*. false
4. Add as many as SQL statements you like into batch using *addBatch()* method on created statement object.

5. Execute all the SQL statements using *executeBatch()* method on created statement object.
6. Finally, commit all the changes using *commit()* method.

The following code snippet provides an example of a batch update using PrepareStatement object −

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(?, ?, ?, ?)";

// Create PrepareStatement object
PreparedStatemen pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.
//Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```