CHAPTER - 15

SWINGS

15. Introduction

GUI represents a pictorial interface to a program. GUI environment provides a distinctive appearance the user to interact with the program. GUI's are built from GUI components.

Swing is a set of classes that provides more powerful and flexible functionality than in possible with the standard AWT components. The classes that are used to create the GUI components are part of the swing GUI components from package javax.swing. These are the newest GUI components of the Java2 platform. Swing components are written, manipulated and displayed completely in Java, so called pure Java components.

The outset that Swing is built upon the foundation of the AWT. So, that Swing provides all the facilities that are supported by the AWT in addition that it also provides some additional components such as tabbed panes, scroll panes, trees and tables.

Main difference between AWT and Swing components are in AWT, a Java program executing on different Java platform provides different appearance and some times even different interactions on each platform. AWT components are tied to the local platform are corresponding called heavy weight components. Whereas, Swing component programs are gave same look in any Java platform even we change the environment. So, that the program running with Swing components are often referred as lightweight components. These are platform independent.

The Swing related classes are contained in javax.swing package. Some of the classes are

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 version of push button
JcheckBox	: The Swing check box class
JcomboBox	: Encapsulates a combo box
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
JtextField	: The Swing version of a text field
Jtree	: Encapsulates a tree-based control

15.1 JApplet:

Fundamental to Swing is the **JApplet** class, which extends **Applet.** 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,** call add() method of the applet. Instead, we need to call add() on the content pane of the JApplet object. The content pane can be obtained via the method:

Container getContentPane()

The add() method of the Container was then used to add a component to the content pane.

Ex:
```
Container c=getContentPane();
c.add(comp);
```

Note: add(), remove() and setLayout() are directly available in JApplet.

15.2 Jframe:

JFrame extends the AWT class Frame. It contains additional features that enable it to support Swing components.

15.3 Jcomponent:

Jcomponent class extends the AWT Component and Container classes. It is the top-level class for all swing components.

Icons: Icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. ImageIcon class provides a constructor as:

ImageIcon(String filename)

ImageIcon class implements the **Icon** interface and provides the methods as:

1. int getIconHeight(): returns the height of the icon in pixels
2. int getIconWidth(): returns the width of the icon in pixels

15.4 Labels:

Swing labels are instances of the **JLabel** class, which extends Jcomponent. It displays text and/or an icon. Constructors provided by the JLabel class are:

JLabel(Icon I)

JLabel(String S)

JLabel(String S, Icon I, int Align)

S and I are text and icon used for the label. Align argument is LEFT, RIGHT, CENTER, LEADING or TRAILING

Methods:

1. Icon getIcon()
2. Sting getText()
3. void setIcon(Icon I)
4. void setText(String s)

```
// Example program
import javax.swing.*;
import java.awt.*;
/* <applet code="Jl.class" width=200 height=150>
</Applet> */
public class Jl extends JApplet
{
    public void init()
    {
        Container c=getContentPane();
        c.setLayout(new FlowLayout(FlowLayout.CENTER));
        ImageIcon i=new ImageIcon("HLPstep1.gif");
        JLabel j=new JLabel("Hello",i,JLabel.CENTER);
        c.add(j);
    }
}
```

15.5 TextFields:

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

JTextField()

JTextField(int col)

JTextField(String s, int col)

JTextField(String s)

```
// Example program
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <applet code="Jt.class" width=300 height=250>
</Applet> */
public class Jt extends JApplet implements ActionListener
{
        JTextField t1;
        public void init()
        {
        Container c=getContentPane();
        c.setLayout(new FlowLayout(FlowLayout.CENTER));
        t1=new JTextField(15);
        c.add(t1);
        t1.addActionListener(this);
        }
        public void actionPerformed(ActionEvent e)
        {
        showStatus(t1.getText());
        }
}
```

15.6 Buttons:

Swing buttons are subclasses of the **AbstractButton** class, which extends **Jcomponent. AbstractButton** contains many methods to allow control the behavior of buttons, check boxes, and radio buttons. Some of the methods to handle these things are

1. void setDisabledIcon(Icon d)
2. void setPressedIcon(Icon d)
3. void setSelectedIcon(Icon d)
4. void setRolloverIcon(Icon d)
5. String getText()
6. void setText(String s)

JButton class provides the functionality of the push buttons. Constructors provided by the JButton class are:

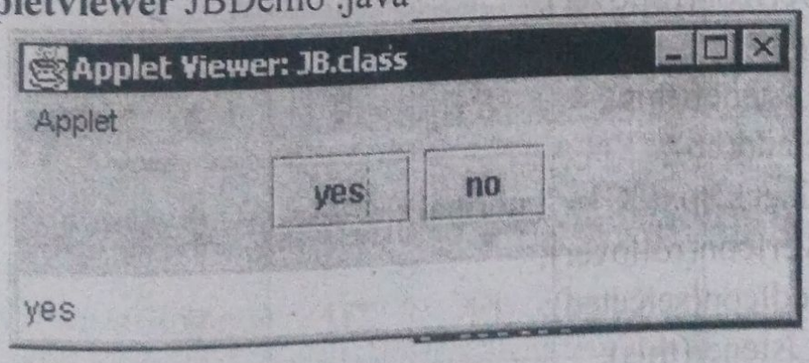
JButton(Icon I)
JButton(String s)
JButton(String s, Icon I)

```
// Example Program
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/* <applet code="JBDemo.class" width=300 height=250>
</Applet> */
public class JBDemo extends JApplet implements ActionListener
{
    JButton b1,b2;
    public void init()
    {
    Container c=getContentPane();
    c.setLayout(new FlowLayout(FlowLayout.CENTER));
    b1=new JButton("yes");
    c.add(b1);
    b1.addActionListener(this);
    b2=new JButton("no");
    c.add(b2);
    b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
    showStatus(e.getActionCommand());
    }
}
```

OUTPUT

C:VIJAY>javac JBDemo .java

C:\vijay>**appletviewer** JBDemo .java



15.7 Check Boxes

implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are shown here:

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

Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
JTextField jtf;
public void init() {

// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("C++", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
```

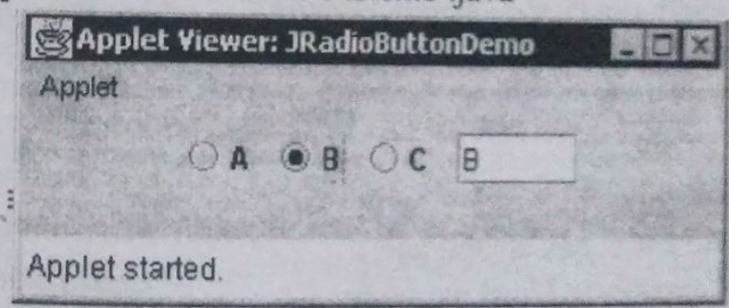


```
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);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

OUTPUT

C:VIJAY>javac JCheckBoxDemo .java

C:\vijay>appletviewer JCheckBoxDemo .java



15.8 Radio Buttons

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

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

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.

The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

The following example illustrates how to use radio buttons

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
JTextField tf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
// Define a button group
ButtonGroup bg = new ButtonGroup();
```

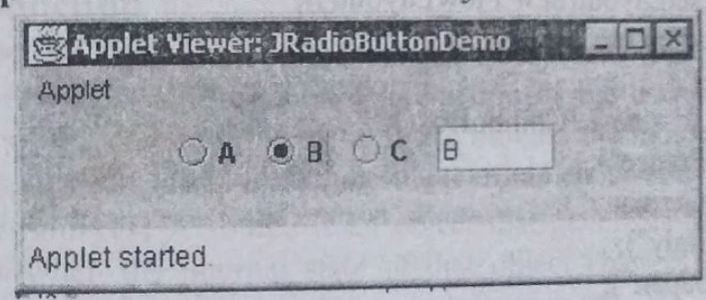


```
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```

OUTPUT

C:VIJAY>javac JRadioButtonDemo .java

C:\vijay>appletviewer JRadioButtonDemo .java



15.9 Combo Boxes

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field. Two of **JComboBox**'s constructors are shown here:

JComboBox( )

JComboBox(Vector *v*)

Here, *v* is a vector that initializes the combo box.

Items are added to the list of choices via the **addItem( )** method, whose signature is

shown here:

void addItem(Object *obj*)

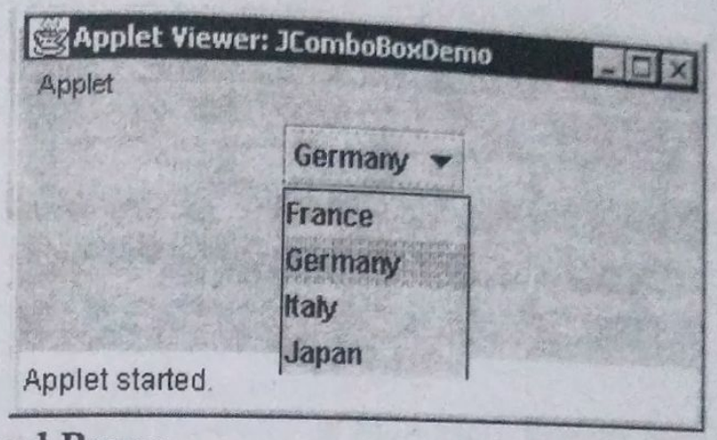
Here, *obj* is the object to be added to the combo box.

The following example contains a combo box and a label.

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

OUTPUT

```
C: >javac JComboBoxDemo .java
C:\ >appletviewer JComboBoxDemo .java
```

15.10 Tabbed Panes

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

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

void addTab(String str, Component comp)

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

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

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
public void init() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
```

```
getContentPane().add(jtp);
}
}
class CitiesPanel extends JPanel {
public CitiesPanel() {
JButton b1 = new JButton("New York");
add(b1);
JButton b2 = new JButton("London");
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
add(b4);
}
}
class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}
class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}
```

15.11 Scroll Panes

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

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

JScrollPane(Component comp)

JScrollPane(int vsb, int hsb)

JScrollPane(Component comp, int vsb, int hsb)

Here, comp is the component to be added to the scroll pane.

vsb and hsb are **int** constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the **ScrollPaneConstants** interface.

Someexamples of these constants are described as follows:

Constant Description

HORIZONTAL_SCROLLBAR_ALWAYS Always provide horizontal scroll bar

HORIZONTAL_SCROLLBAR_AS_NEEDED Provide horizontal scroll bar, if needed

VERTICAL_SCROLLBAR_ALWAYS Always provide vertical scroll bar

VERTICAL_SCROLLBAR_AS_NEEDED Provide vertical scroll bar, if needed

Here are the steps that you should follow to use a scroll pane in an applet:

1. Create a **JComponent** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.

The following example illustrates a scroll pane.

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

```
int b = 0;
for(int i = 0; i < 20; i++) {
for(int j = 0; j < 20; j++) {
jp.add(new JButton("Button " + b));
++b;
}
}
// Add panel to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

15.12 Trees

A *tree* is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class, which extends **JComponent**. Some of its constructors are shown here:

JTree(Hashtable *ht*)

JTree(Object *obj*[ ])

JTree(TreeNode *tn*)

JTree(Vector *v*)

The first form creates a tree in which each element of the hash table *ht* is a child node.

Each element of the array *obj* is a child node in the second form. The tree node tn is the root of the tree in the third form. Finally, the last form uses the elements of vector v as child nodes.

A **JTree** object generates events

The **addTreeExpansionListener( )** and **removeTreeExpansionListener( )** methods allowlisteners to register and unregister for these notifications. The signatures of these methods are shown here:

void addTreeExpansionListener(TreeExpansionListener *tel*)

void removeTreeExpansionListener(TreeExpansionListener *tel)*

The steps that you should follow to use a tree in an applet:

1. Create a **JTree** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

The following example illustrates how to create a tree and recognize mouse clicks on it.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEventsDemo" width=400 height=200>
</applet>
*/
public class JTreeEventsDemo extends JApplet {
JTree tree;
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);
// Create subtree of "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
```

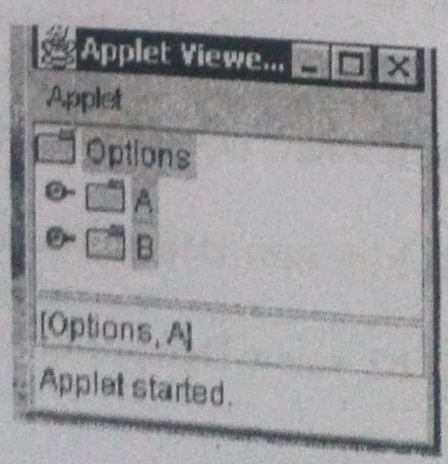
```
b.add(b2);
DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
b.add(b3);
// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(tree, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);

// Add text field to applet
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
public void mouseClicked(MouseEvent me) {
doMouseClicked(me);
}
});
}
void doMouseClicked(MouseEvent me) {
TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
if(tp != null)
jtf.setText(tp.toString());
else
jtf.setText("");
}
}
```

OUTPUT

C: >javac JTreeEventsDemo .java

C:\ >appletviewer JTreeEventsDemo .java



15.13 Tables

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

One of its constructors is shown here:

JTable(Object data[ ][ ], Object colHeads[ ])

Here, data is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Here are the steps for using a table in an applet:

1. Create a **JTable** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table

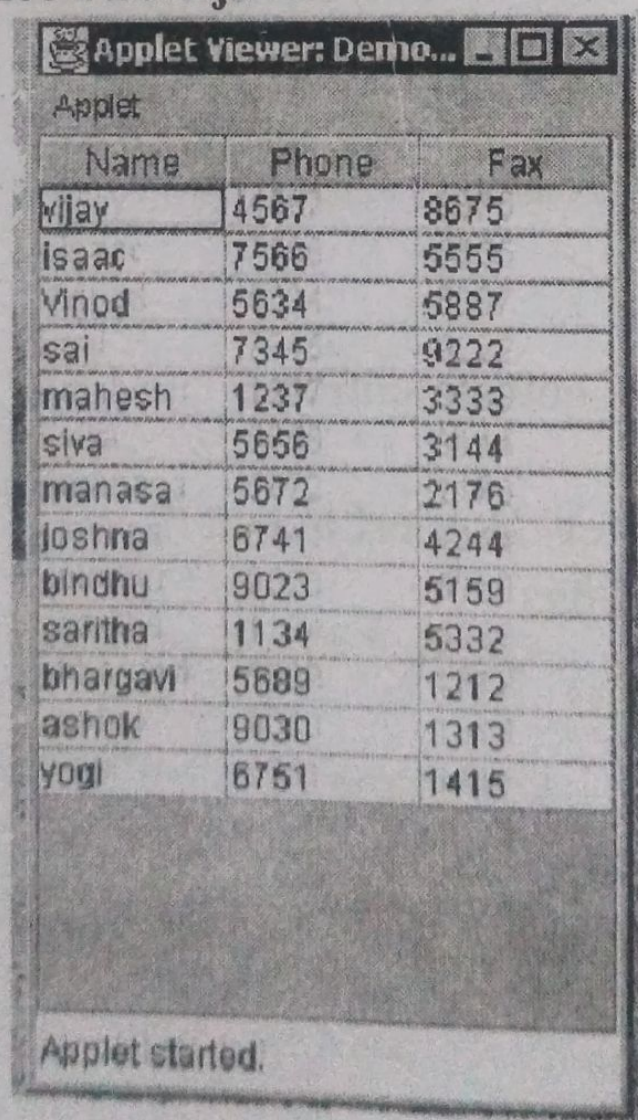
```
import java.awt.*;
import javax.swing.*;
/*
<applet code="DemoJTable" width=200 height=300>
</applet>
*/
public class DemoJTable extends JApplet {
public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "vijay", "4567", "8675" },
{ "isaac", "7566", "5555" },
{ "Vinod", "5634", "5887" },
{ "sai", "7345", "9222" },
{ "mahesh", "1237", "3333" },
{ "siva", "5656", "3144" },
{ "manasa", "5672", "2176" },
```

```
{ "joshna", "6741", "4244" },
{ "bindhu", "9023", "5159" },
{ "saritha", "1134", "5332" },
{ "bhargavi", "5689", "1212" },
{ "ashok", "9030", "1313" },
{ "yogi", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

OUTPUT

C: >javac DemoJTable.java

C:\ >appletviewer DemoJTable.java



CHAPTER - 16

JDBC (JAVA DATA BASE CONNECTIVITY)

16. Overview

This provides a brief introduction to JDBC. Explains what is JDBC and how it can be used to access RDBMS. Provides a brief overview of JDBC architecture.

16.1 What is JDBC?

JDBC stands for "Java DataBase Connectivity". It is an API (Application Programming Interface) which consists of a set of Java classes, interfaces and exceptions and a specification to which both JDBC driver vendors and JDBC developers (like you) adhere when developing applications.

JDBC is a very popular data access standard. RDBMS (Relational Database Management Systems) or third-party vendors develop drivers which adhere to the JDBC specification. Other developers use these drivers to develop applications which access those databases e.g. you'll use Connector JDBC driver to access MySQL database. Since the drivers adhered to JDBC specification, the JDBC application developers can replace one driver for their application with another better one without having to rewrite their application. If they had used some proprietary API provided by some RDBMS vendor, they will not have been able to change the driver and/or database without having to rewrite the complete application.

16.2 Who develops JDBC Specification?

SUN prepares and maintains the JDBC specification. Since JDBC is just a specification (suggestions for writing and using JDBC drivers), third-party vendors develop JDBC drivers adhering to this specification. JDBC developers then use these drivers to access data sources.

16.3 Why use JDBC?

JDBC is there only to help you (a Java developer) develop data access applications without having to learn and use proprietary APIs provided

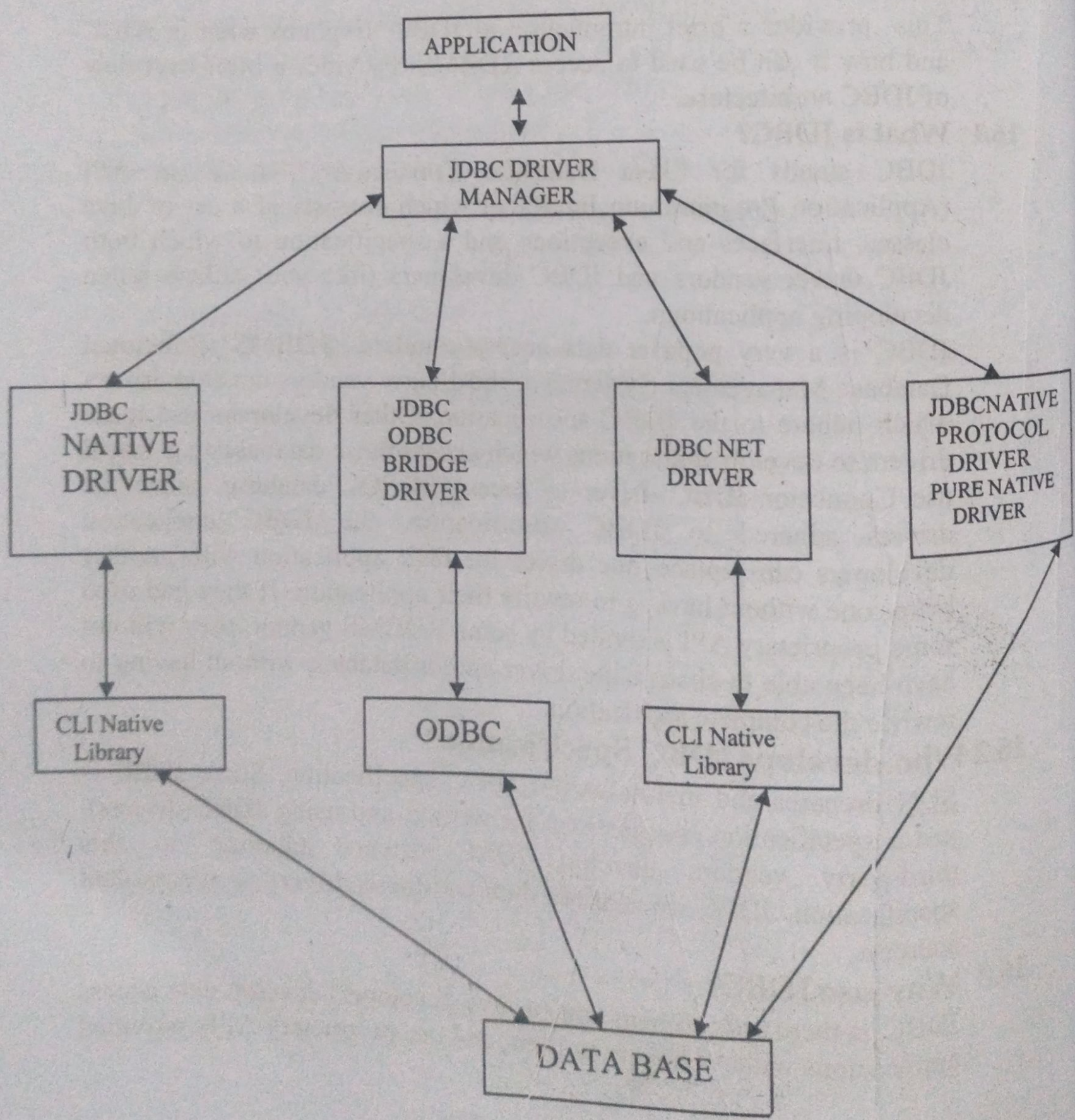
by different RDBMS vendors. You just have to learn JDBC and then you can be sure that you'll be able to develop data access applications which can access different RDBMS using different JDBC drivers.

16.4 JDBC Architecture

We'll divide it into 2 parts:

- JDBC API (java.sql & javax.sql packages)
- JDBC Driver Types

Architecture of JDBC Application:



16.5 JDBC API

The JDBC API is available in the java.sql and javax.sql packages. Following are important JDBC classes, interfaces and exceptions in the java.sql package:

- DriverManager - Loads JDBC drivers in memory. Can also be used to open connections to a data source.
- Connection - Represents a connection with a data source. Is also used for creating Statement, PreparedStatement and CallableStatement objects.
- Statement - Represents a static **SQL** statement. Can be used to retrieve ResultSet object/s.
- PreparedStatement - Higher performance alternative to Statement object, represents a precompiled SQL statement.
- CallableStatement - Represents a stored procedure. Can be used to execute stored procedures in a RDBMS which supports them.
- ResultSet - Represents a database result set generated by using a SELECT SQL statement.
- SQLException - An exception class which encapsulates database base access errors.

javax.sql is part of J2SE 1.4 and J2EE 1.3. It adds following features to JDBC in addition to the ones provided by java.sql package:

- DataSource - Abstracts a data source. This object can be used in place of DriverManager to efficiently obtain data source connections (possibly using hidden connection pooling).
- Provides built-in connection pooling.
- XADataSource, XAConnection - Allows/supports distributed transactions.
- RowSet - It extends ResultSet interface to add support for disconnected result sets.

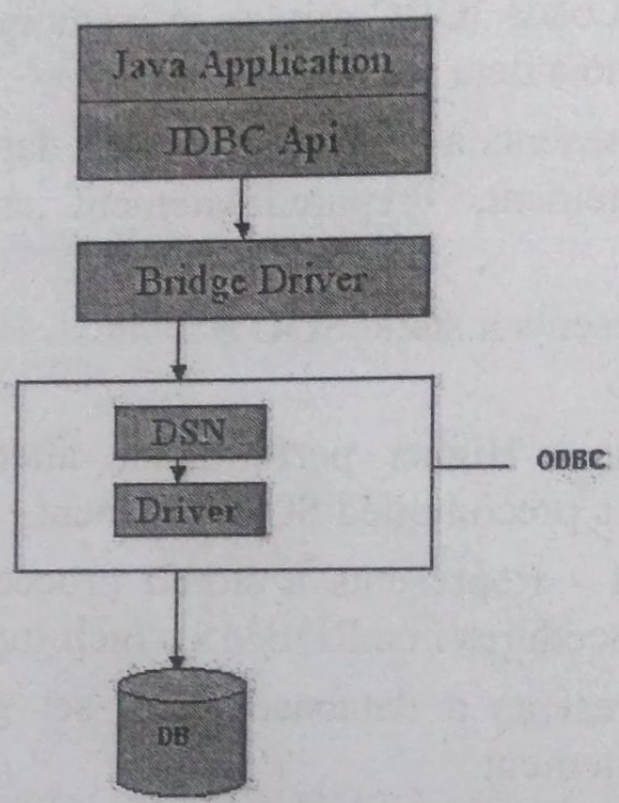
16.6 Types of JDBC Drivers

JDBC drivers are divided into four types or levels. The four types are:

Type1: JDBC-ODBC Bridge driver (Bridge)
Type2: Native-API/partly Java driver (Native)
Type3: AllJava/Net-protocol driver (Middleware)
Type 4: All Java/Native-protocol driver (Pure)

i) Type 1: JDBC-ODBC Bridge driver

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Type 1: JDBC-ODBC Bridge

Advantage

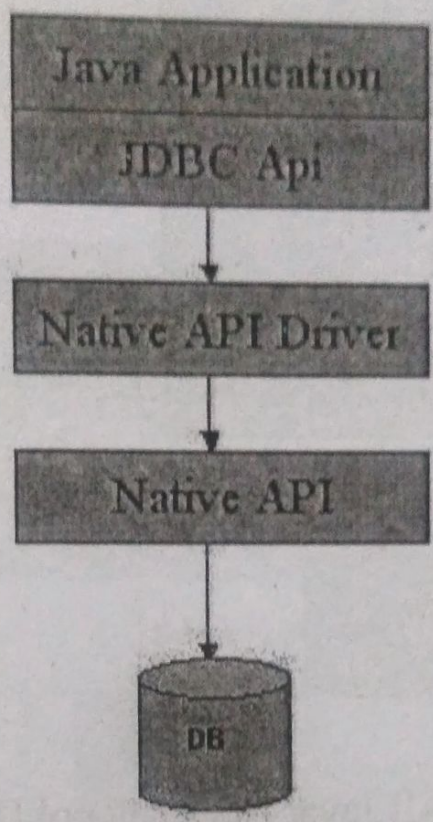
The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

ii) Type 2: Native-API/partly Java driver

The Type 2 driver converts JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Example: Oracle will have oracle native api.



Type 2: Native api/ Partly Java Driver

Advantage

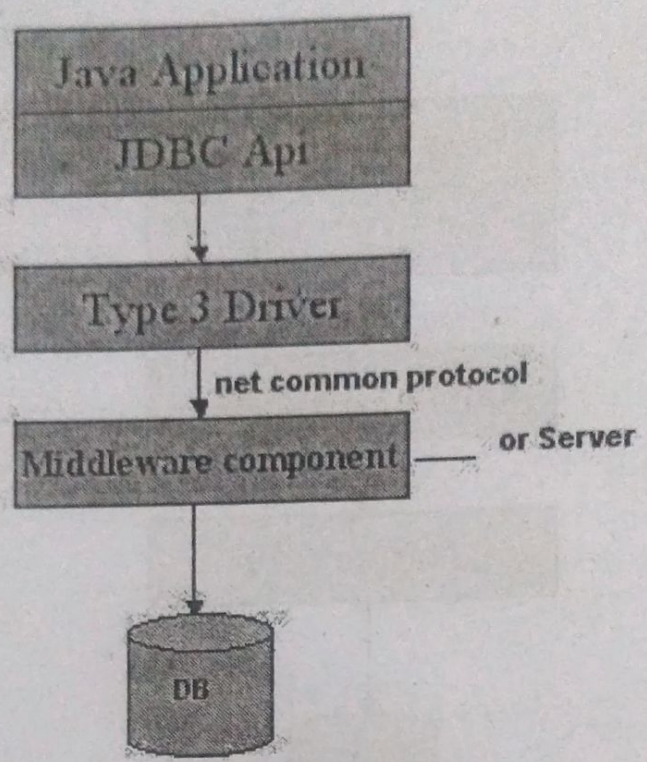
Type 2 drivers typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

iii) Type 3: All Java/Net-protocol driver

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



Type 3: All Java/ Net-Protocol Driver

Advantage

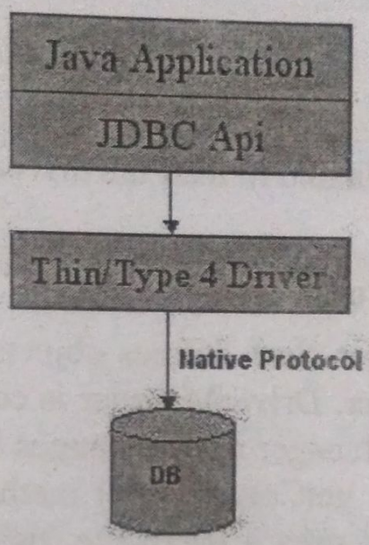
1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

iv) *Type 4: Native-protocol/all-Java driver*

The Type 4 uses java networking libraries to communicate directly with the database server.



Type 4: Native-protocol/all-Java driver

Advantage

1. Type 4 drivers are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. 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, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

16.7 JDBC Basics - Java Database Connectivity Steps

Before you can create a java jdbc connection to the database, you must first import the java.sql package.

1. **Loading a database driver,**

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly

used. The return type of the Class.forName(String ClassName) method is "Class". Class is a class in java.lang package.

```
try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");   //Or any other
driver
}
catch(Exception x){
        System.out.println( "Unable to load the driver class!" );
}
```

2. Creating a oracle jdbc Connection

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system. Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

JDBC URL Syntax:: jdbc: <subprotocol>: <subname>

- *Each driver has its own subprotocol*
- *Each subprotocol has its own syntax for the source*

Example: For example, we're using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
try{
        Connection dbConnection = DriverManager.getConnection(url,
"loginName", "Password")
}
catch( SQLException x ){
        System.out.println( "Couldn't get connection!" );
}
```

3. Creating a jdbc Statement object.

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

Statement statement = dbConnection.createStatement();

A statement object is used to send and execute SQL statements to a database.

Three kinds of Statements

Statement: Execute simple sql queries without parameters.

Statement createStatement()

a) Creates an SQL Statement object.

Prepared Statement: Execute precompiled sql queries with or without parameters.

b) PreparedStatement prepareStatement(String sql)

returns a new PreparedStatement object. PreparedStatement objects are precompiled SQL statements.

c) Callable Statement: Execute a call to a database stored procedure. CallableStatement prepareCall(String sql)

returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

4. Executing a SQL statement with the Statement object, and returning a java ResultSet.

Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements: executeQuery(), executeUpdate(), and execute(). For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. execute() executes an SQL statement that is written as String object.

ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

16.8 Test JDBC Driver Installation

```
import javax.swing.JOptionPane;

    public class TestJDBCDriverInstallation_Oracle {
      public static void main(String[] args) {
          StringBuffer output  = new StringBuffer();
          output.append("Testing oracle jdbc driver installation \n");
          try {
             String className = "sun.jdbc.odbc.JdbcOdbcDriver";
             Class driverObject = Class.forName(className);
             output.append("Driver : "+driverObject+"\n");
             output.append("JDBC Driver Installation Successful");
                        JOptionPane.showMessageDialog(null, output);
            } catch (Exception e) {
                    output  = new StringBuffer();
                    output.append("JDBC Driver Installation FAILED\n");
            // JDBC Driver installation Failed
                    JOptionPane.showMessageDialog(null, output);
      System.out.println("Failed: JDBC Driver Error: " + e.getMessage());
          }
      }
```

16.9 Test JDBC Connection, JDBC Driver Name and Version

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.SQLException;
public class JDBCDriverInformation {
      static String userid="scott", password = "tiger";
      static String url = "jdbc:odbc:bob";
      static Connection con = null;
      public static void main(String[] args) throws Exception {
```

```
        Connection con = getOracleJDBCConnection();
        if(con!= null){
            System.out.println("Got Connection.");
            DatabaseMetaData meta = con.getMetaData();
            System.out.println("Driver Name : "+meta.getDriverName());
            System.out.println("Driver Version : "+meta.getDriverVersion());
        }else{
            System.out.println("Could not Get Connection");
        }
    }
    public static Connection getOracleJDBCConnection(){
        try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(java.lang.ClassNotFoundException e) {
                System.err.print("ClassNotFoundException: ");
                System.err.println(e.getMessage());
        }
        try {
    con = DriverManager.getConnection(url,   userid, password);
        } catch(SQLException ex) {
                System.err.println("SQLException: " + ex.getMessage());
        }

        return con;
    }
```

16.10 JDBC Create Table

Create SQL Table:

Tables are composed of rows and columns. Each row represents a record in the database. Columns are also known as fields, or attributes. You can relate one database table to another by causing a given column in the table to derive its value from the value of a column in another table. If the tables had no columns in common, then there would be no way to relate them to one another. Because you can link tables together you can easily extract data

from multiple tables with a single query, if your query mechanism supports this type of query.

In order to integrate your tables into a single database, you'll need to ensure that each table has a column that contains a value unique to that table. Such a column is called a key. Below is a JDBC Program showing the use of executeUpdate() to create a table jdbc programming.

1) CREATE TABLE Employees (Employee_ID INTEGER,Name VARCHAR(30));

Employees:

Employee_ID	Name

Code:

```
Import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate(create table Emplyee(Employee_ID INTEGER, Name
VARCHAR(30));

System.out.println(table is created);
St.close();
con.close();
}
}
```

2)Create orders table

```
CREATE TABLE Orders (
Prod_ID                    INTEGER,
ProductName                VARCHAR(20),
Employee_ID                INTEGER
);
```

Prod_ID	ProductName	Employee_ID

CODE:

```
Import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate(CREATE TABLE Orders (Prod_ID
INTEGER,ProductName VARCHAR(20),
  Employee_ID INTEGER);
System.out.println(table is created);
St.close();
con.close();
}
}
```

16.11 JDBC Insert Data into SQL Tables

We will insert data into the table's Employees and Orders as created in the previous tutorial, one row at a time, supplying the information to be stored in each column of that row. The values to be inserted into the columns are listed

in the same order that the columns were declared when the table was created. Below is a JDBC Program showing the use of executeUpdate() to create a table and insert data into using jdbc programming.

Employees:

Employee_ID	Name
6323	Vijay
5768	Isaac
1234	Vinod
5678	Sai

Code:

```
import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate(" insert into Emplyee values(6383,'vijay')");
St.executeUpdate(" insert into Emplyee values(5768,'Isaac')");
St.executeUpdate(" insert into Emplyee values(1234,'vinod')");
St.executeUpdate(" insert into Emplyee values(5678,'sai')");

 System.out.println("table values are inserted successfully");
St.close();
con.close();
}
}
```

Orders Table:

Prod_ID	ProductName	Employee_ID
543	Belt	6323
432	Bottle	1234
876	Ring	5678

Code:

```
import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate(" insert into Orders values(543,'Belt',6383)");
St.executeUpdate(" insert into Orders values(432,'bottle',1234)");
St.executeUpdate(" insert into Orders values(876,Ring,5678)");

 System.out.println("table values are inserted successfully");
St.close();
con.close();
}
}
```

16.12 JDBC Select Data

Retrieving Data from SQL Tables

We can use SELECT statements in a java program to retieve the data and display it for the respective Tables. JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results. Select is the SQL keyword that performs a query. We invoke the executeQuery() method, using the statement text as a parameter. The tabular results of the query are captured in the ResultSet object, results. Note that executeQuery() always returns a ResultSet although it need not have any rows in it.

To process each row in the ResultSet, we use the next() method. This method moves the pointer through the rows of data. The ResultSet maintains a cursor pointing to the current row. Because this cursor is initially positioned before the first row, we must call next() before we can see any rows at all. Below is a JDBC Program showing the use of executeQuery() to retrieve values from ResultSets using jdbc programming.

Employees:

Employee_ID	Name
6323	Vijay
5768	Isaac
1234	Vinod
5678	Sai

Code:

```
Import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
RwsultSet res=stmt.executeQuery(" select * from Employees");
While(res.next( ))
{
 System.out.println("res.getInt(1)+" "+res.getString(2));
}
System.out.println("table  is selected successfully");
St.close();
con.close();
}
}
```

Orders:

Prod_ID	ProductName	Employee_ID
543	Belt	6323
432	Bottle	1234
876	Ring	5678

```
Import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
RwsultSet res=stmt.executeQuery(" select * from orders");
While(res.next( ))
{
 System.out.println("res.getInt(1)+" "+res.getString(2)+" "+res.getInt(3));
}
System.out.println("table  is selected successfully");
St.close();
con.close();
}
}
```

16.13 JDBC Update Table

Update SQL Tables

We can use Update statements in a java program to update the data for a Table. Below is a JDBC Program showing the use of executeUpdate() to update a table using jdbc programming.

Employees:

Employee_ID	Name
6333	Vijay
5758	Isaac
124	Vinod
5678	Sai

```
import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate("update Employee set Employee_ID=_Employee_ID+10
where name=vijay");
System.out.println("table values are inserted successfully");
St.close();
con.close();
}
}
```

Orders:

Prod_ID	ProductName	Employee_ID
543	Belt	6323
432	Bottle	1234
876	Ring	5678

```
Import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate("update oreders set Prod_ID=Prod_ID+10 where
ProductName=vijay");
```

```
System.out.println("table values are inserted successfully");
St.close();
con.close();
}
}
```

16.14 JDBC Prepared Statements

Using Prepared Statements

Prepared statements are pre-compiled SQL statements. Precompiled SQL is useful if the same SQL is to be executed repeatedly, for example, in a loop. Prepared statements only save you time if you expect to execute the same SQL over again. Every statement is compiled at some point. To use a prepared statement, you must first create a PreparedStatement object by calling the Connection.prepareStatement() method. PreparedStatement are useful especially in situations where you can use a for loop or while loop to set a parameter to a succession of values.

The syntax is straightforward: just insert question marks for any parameters that you'll be substituting before you send the SQL to the database. As with CallableStatements, you need to call close() to make sure database resources are freed as soon as possible. Below is a JDBC Program showing the use of jdbc prepared statements to insert data into tables using jdbc programming.

Employees:

Employee_ID	Name
6323	Vijay
5768	Isaac
1234	Vinod
5678	Sai

```
import java.sql.*;
Class jdbcdemo
{
public static void main(String args[])throws
IOException,ClassNotFoundException
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
Connection con= DriverManager.getConnection("jdbc:odbc:BATCH");
Statement st = con.createStatement();
St.executeUpdate(" insert into Emplyee values(????,'????')");
System.out.println("table values are inserted at Runtime ");
St.close();
con.close();
}
}
```

Output

```
Enter the value for Employee_ID:: 6323
Enter the Name      :: Vijay
Enter the value for Employee_ID:: 5768
Enter the Name      :: Isaac
Enter the value for Employee_ID:: 1234
Enter the Name      :: Vinod
Enter the value for Employee_ID:: 5678
Enter the Name      :: Sai
```