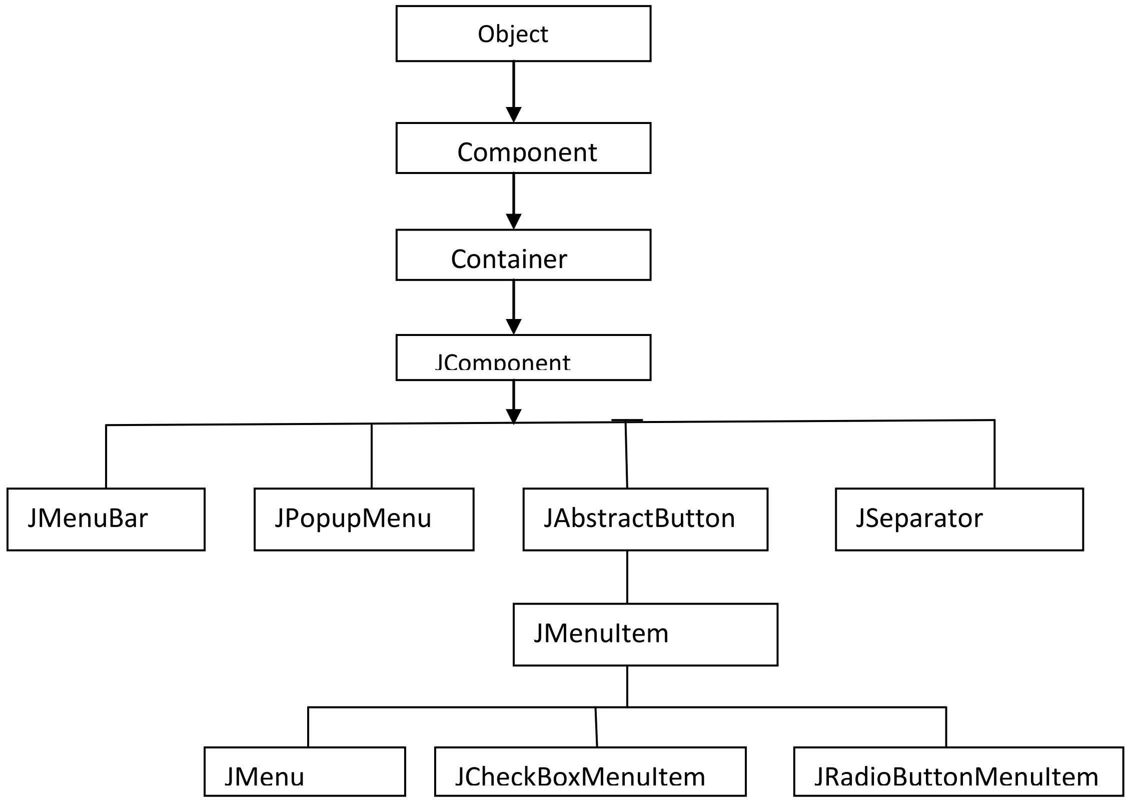
Menus

A menu is a very useful part of a GUI. It displays a list of items that indicate various operations that the user can do. The way we use the menu is simple . Menus usually display several options that are broadly categorized. Each of the menu items will have some action associated with it. A Swing menu consists of a menubar, menuitems and menus. The menubar is the root for all menus and menu items.



JMenuBar
A JMenuBar is a component that can be added to a container similar to any other component and only through a JFrame, JWindow or JInternalFrame’s rootpane. A JMenuBar cannot be added directly to a frame. It consists of a number of JMenus with each JMenu represented as a string within the JMenuBar. The JMenuBar requires two additional classes to supplement its work. They are-

1. singleSelectionMode
2. A Look and feel class

The SingleSelectionMode class keeps track of the menu that is currently selected. The look and feel class is responsible for drawing the menu bar and responding to events that occur in it.

JMenu

JMenu serves dual purposes. When appearing under JMenuBar, it is seen as a text string and when the user clicks on the string, it is seen as a popup menu. JMenu can include standard menu items such as JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem as well as JSeparator. Similar to JMenuBar, the JMenu class also needs two additional classes to supplement its operation. They are JPopupMenu and the look and feel class. JMenu creates a JPopupMenu class and the look and feel class is responsible for drawing the menu in the menubar and for responding to all events that occur in it.

JPopupMenu

This component is used to display an expanded form of menu and can be used in two ways. It is also used for the “pull-right” menu that appears when the user selects a menu item that activates. The JPopupMenu can also be used as a shortcut menu ,which is activated by the right click of the mouse. The following constructors are used to create –

1. public JPopupMenu ()—It creates a JPopupMenu.
2. public JPopupMenu (String label)—It help to creates a JPopupMenu with the specified title.

JMenuItem

It is a component that typically appears as a text sting possibly with an icon in a JMenu or JPopupMenu. When the user clicks and releases the mouse on a JMenuItem, the menu that contains this item will disappear from the screen and the corresponding dialog box will be displayed form the menu item.

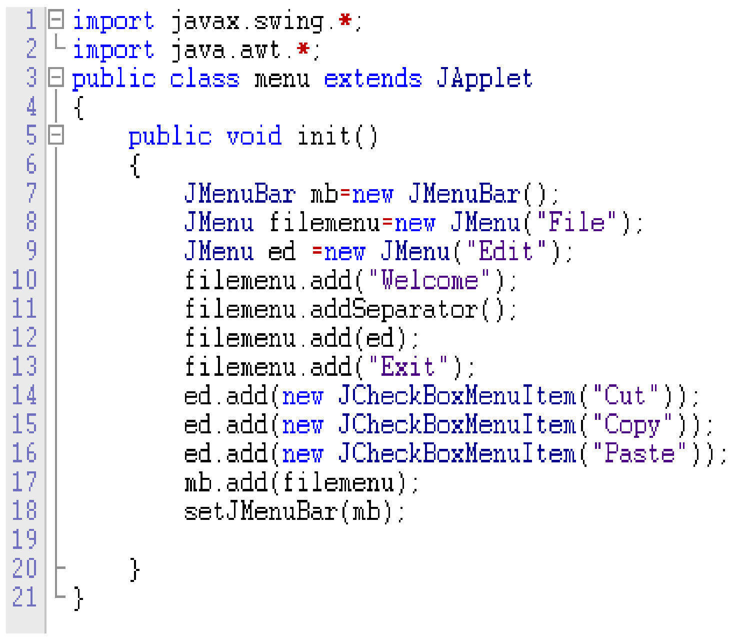
JCheckBoxMenuItem

This menu item contains checkboxes as its items. The checkboxes are created using JCheckBox class and will be displayed selected or unselected . This component may have a text string and or icon.
When a jCheckBoxMenuItem is clicked and released , the state of a menu item changes to selected or deselected. In such a situation , an ItemEvent is generated and is distributed to all registered ItemListeners.

JRadioButtonMenuItem

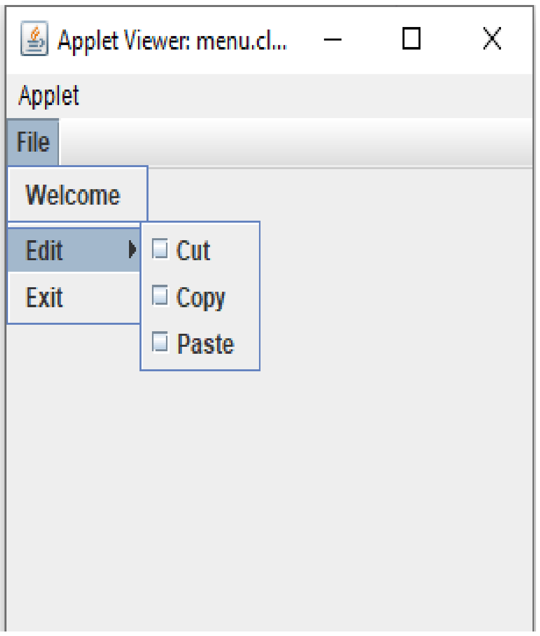
A JRadioButtonMenuItem is a unique Swing menu component because only one can be selected at any point of time. When a JRadioButtonMenuItem is clicked and released , two possibilities may take place . If the radio button is already selected , then it does not change its state, But if it was not initially selected then the selected one will be deselected and clicked item will be selected.
When a JRadioButtonMenuItem changes its state , it creates an ItemEvent, which will be distributed to all the registered ItemListeners.

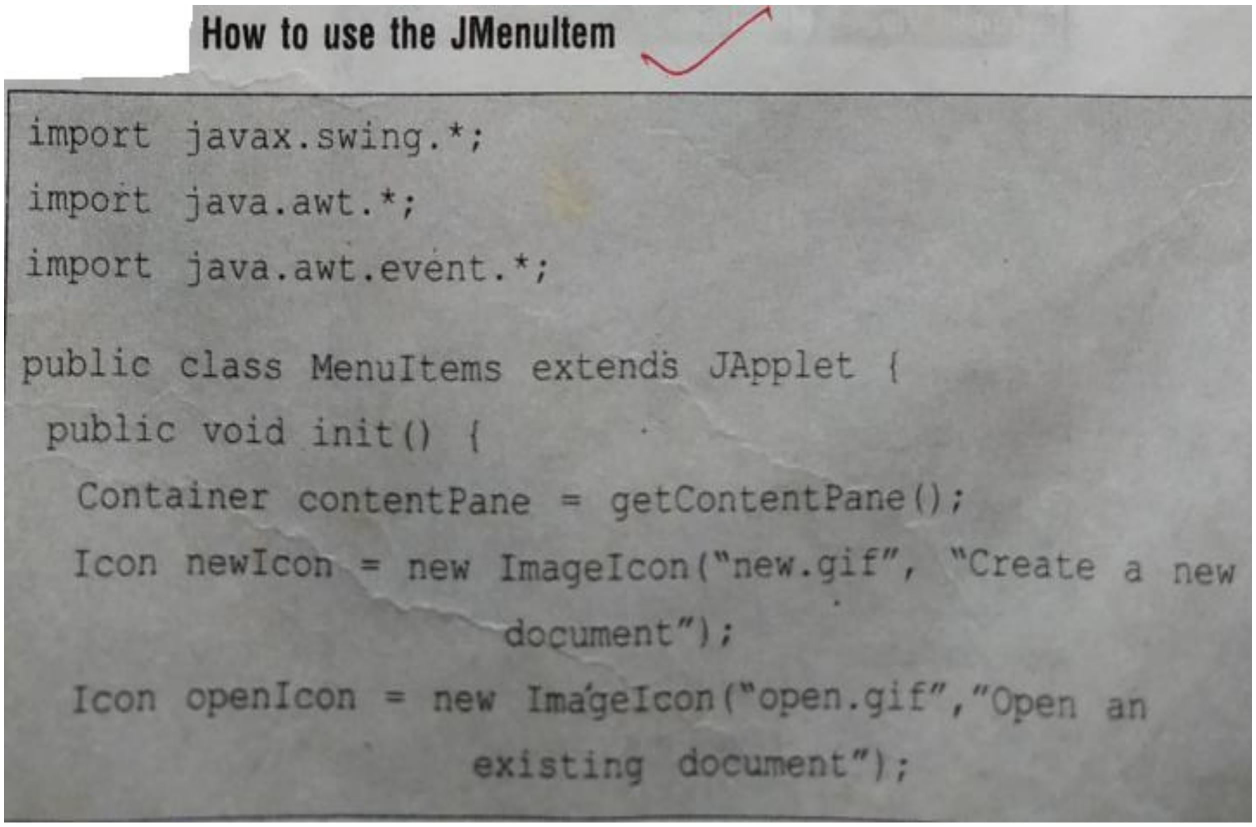
How to create menu



```java
import javax.swing.*;
import java.awt.*;
public class menu extends JApplet
{
    public void init()
    {
        JMenuBar mb=new JMenuBar();
        JMenu filemenu=new JMenu("File");
        JMenu ed =new JMenu("Edit");
        filemenu.add("Welcome");
        filemenu.addSeparator();
        filemenu.add(ed);
        filemenu.add("Exit");
        ed.add(new JCheckBoxMenuItem("Cut"));
        ed.add(new JCheckBoxMenuItem("Copy"));
        ed.add(new JCheckBoxMenuItem("Paste"));
        mb.add(filemenu);
        setJMenuBar(mb);

    }
}
```

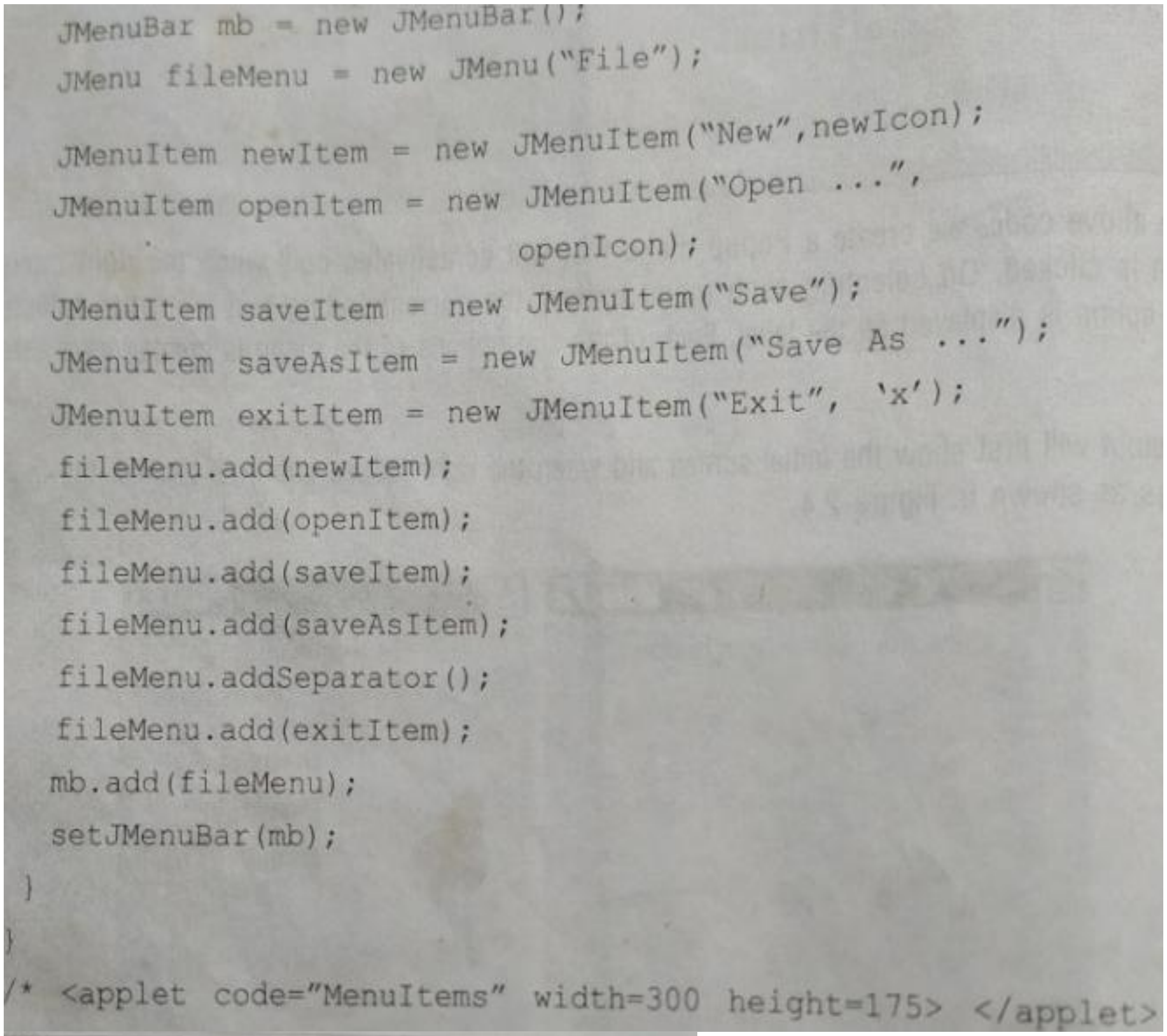




How to use the JMenuItem

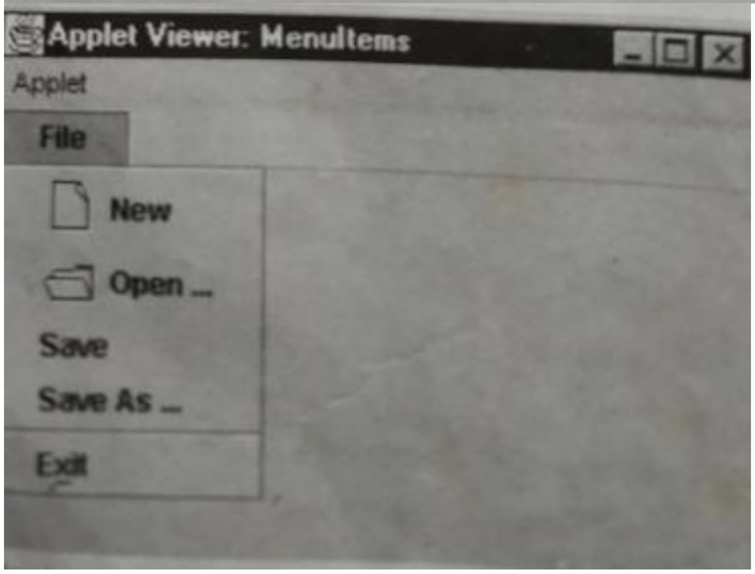
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MenuItems extends JApplet {
 public void init() {
   Container contentPane = getContentPane();
   Icon newIcon = new ImageIcon("new.gif", "Create a new
                         document");
   Icon openIcon = new ImageIcon("open.gif","Open an
                        existing document");
```

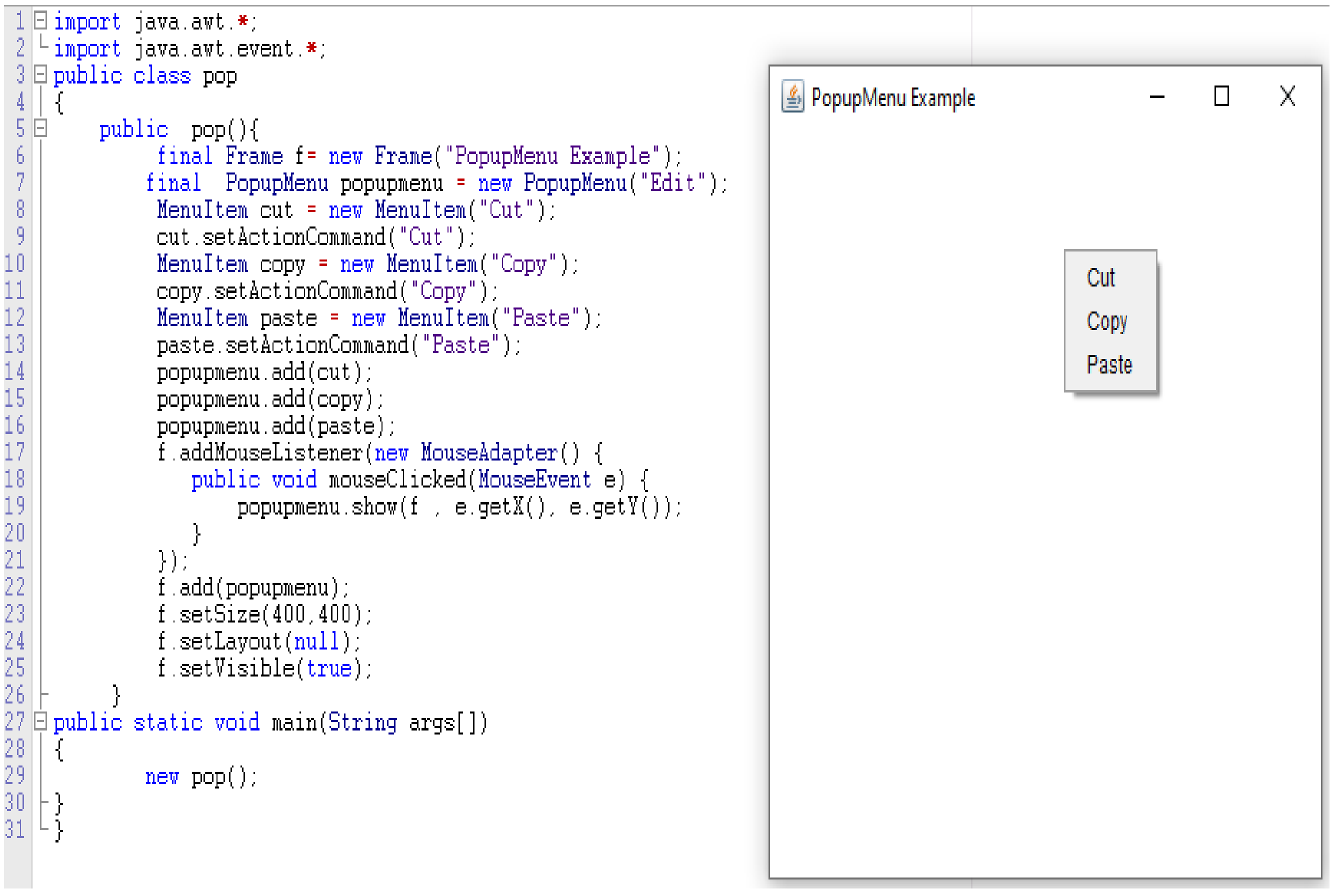


```
  JMenuBar mb = new JMenuBar();
  JMenu fileMenu = new JMenu("File");

  JMenuItem newItem = new JMenuItem("New",newIcon);
  JMenuItem openItem = new JMenuItem("Open ...",
                                   openIcon);
  JMenuItem saveItem = new JMenuItem("Save");
  JMenuItem saveAsItem = new JMenuItem("Save As ...");
  JMenuItem exitItem = new JMenuItem("Exit", 'x');
  fileMenu.add(newItem);
  fileMenu.add(openItem);
  fileMenu.add(saveItem);
  fileMenu.add(saveAsItem);
  fileMenu.addSeparator();
  fileMenu.add(exitItem);
  mb.add(fileMenu);
  setJMenuBar(mb);
 }
}
/* <applet code="MenuItems" width=300 height=175> </applet>
```



PopUPMenu--



```
import java.awt.*;
import java.awt.event.*;
public class pop
{
    public  pop(){
         final Frame f= new Frame("PopupMenu Example");
        final  PopupMenu popupmenu = new PopupMenu("Edit");
         MenuItem cut = new MenuItem("Cut");
         cut.setActionCommand("Cut");
         MenuItem copy = new MenuItem("Copy");
         copy.setActionCommand("Copy");
         MenuItem paste = new MenuItem("Paste");
         paste.setActionCommand("Paste");
         popupmenu.add(cut);
         popupmenu.add(copy);
         popupmenu.add(paste);
         f.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                popupmenu.show(f , e.getX(), e.getY());
            }
         });
         f.add(popupmenu);
         f.setSize(400,400);
         f.setLayout(null);
         f.setVisible(true);
     }
public static void main(String args[])
{
        new pop();
}
}
```

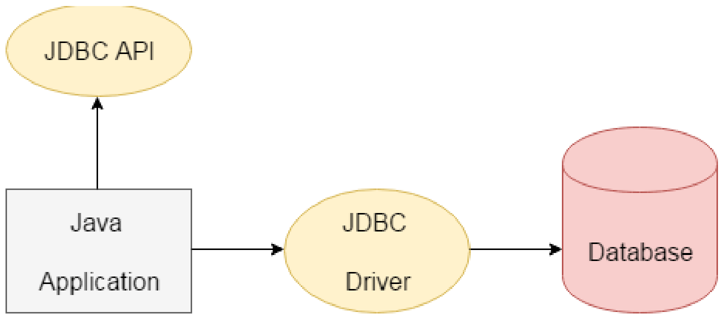
Java JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

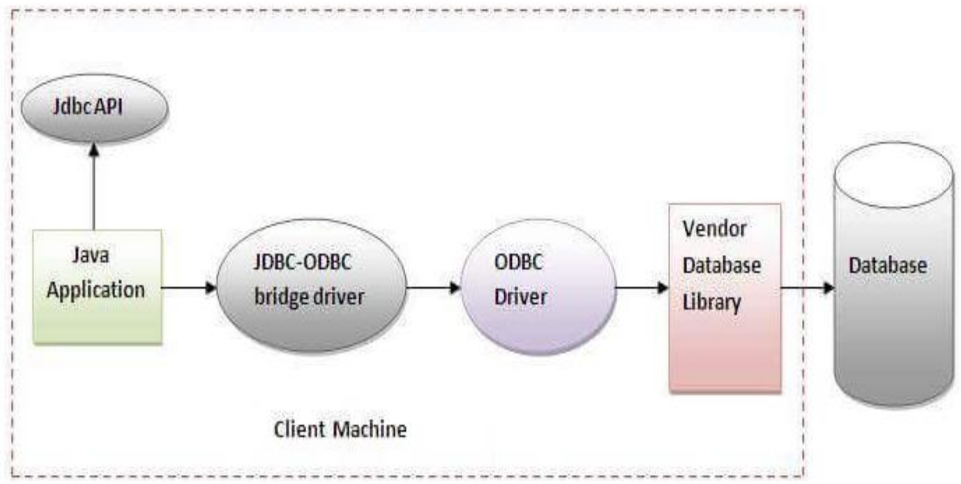
JDBC Driver

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

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

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.



Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

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

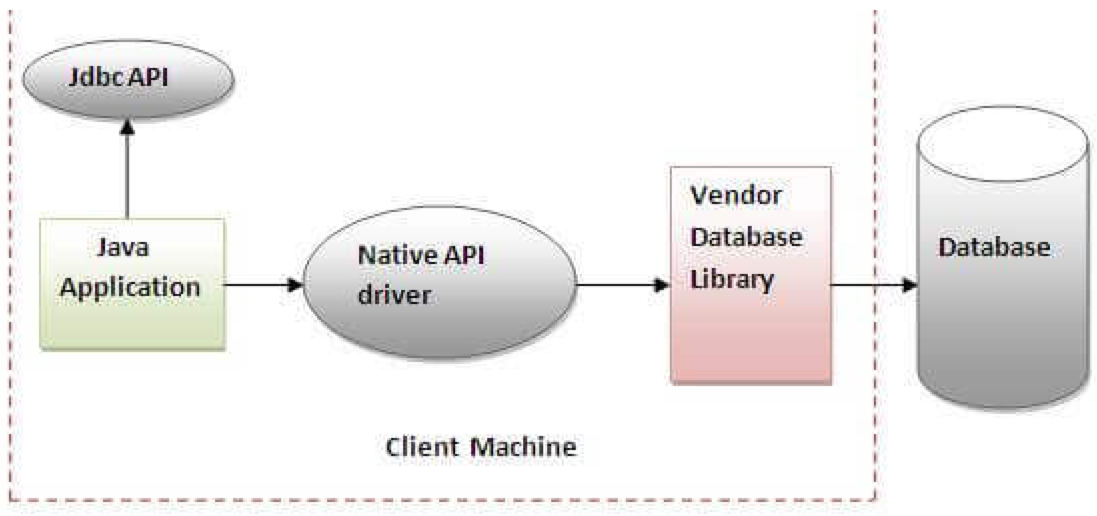


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

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

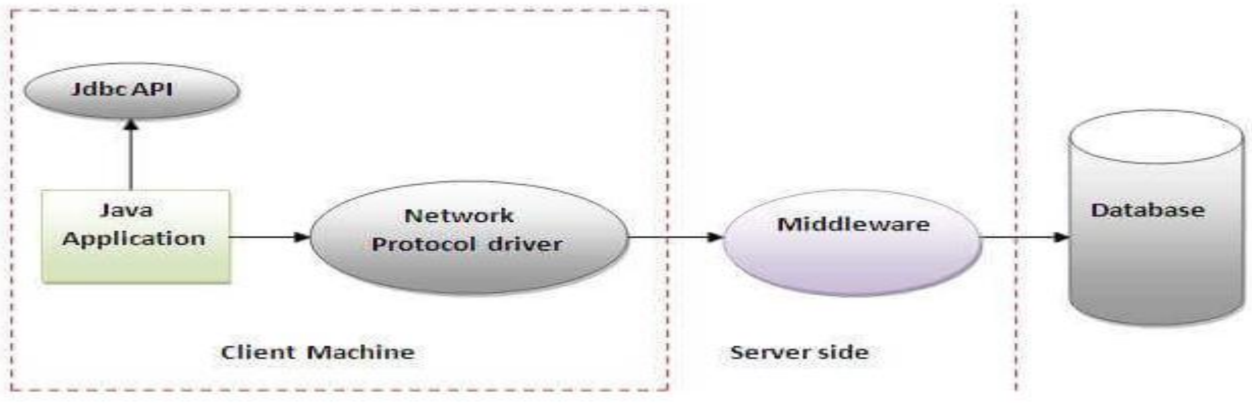


Figure- Network Protocol Driver

Advantage:

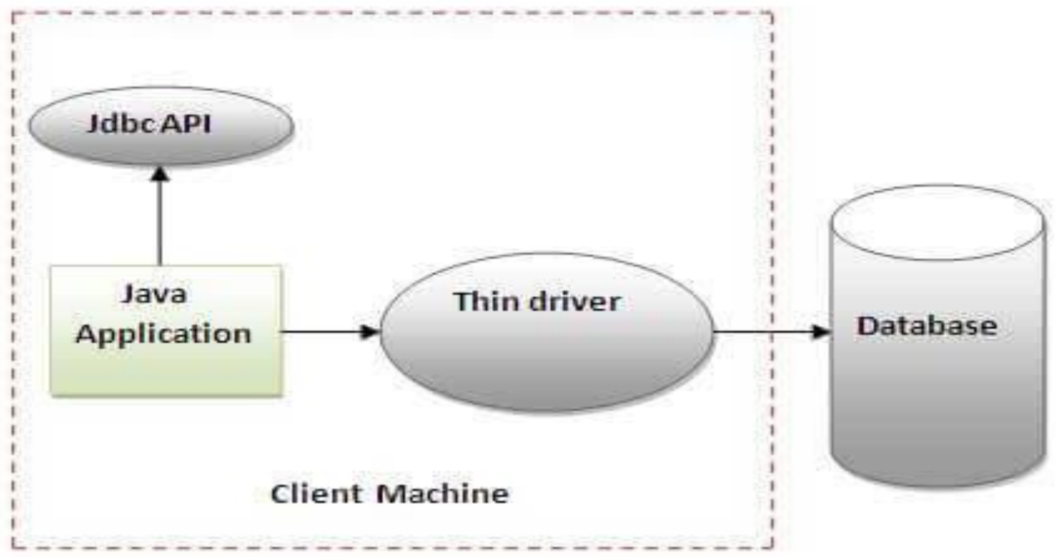
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

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



Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Program for Database

```
import java.sql.*;

class dba

{

        public static void main(String args[])
        {
                        Connection  con;
                        Statement sa;
                         ResultSet  rs;


                try

                {

                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");



                        con=DriverManager.getConnection("jdbc:odbc:we");

                        sa=con.createStatement();

                        rs=sa.executeQuery("select roll,sname from student");

                        while(rs.next())

                                {

                                System.out.print(rs.getString(1)+"   ");

                                System.out.print(rs.getString(2));

                                System.out.println();

                        }

                }

                catch(Exception e){

                        System.out.println(e.toString());

                }       }  }
```

GUI of DataBase

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;

public class da21 extends Frame implements ActionListener
{
        static Connection con;
        static Statement sa;
        static ResultSet rs;
        Label l1,l2;
        TextField t1,t2;
        Button b1,b2,b3,b4;
        Panel p;
        public da21()
        {
                super("using database ");
                setBackground(Color.gray);

                l1=new Label("roll");
                l2=new Label("name");
                t1=new TextField(10);
                t2=new TextField(10);
                b1=new Button("Exit");
                b2=new Button("next");
                b3=new Button("save");
                b4=new Button("clear");
                p=new Panel();
                b1.addActionListener(this);
                b2.addActionListener(this);
                b3.addActionListener(this);
                b4.addActionListener(this);
                p.add(l1);
                p.add(t1);
                p.add(l2);
                p.add(t2);
                p.add(b1);
                p.add(b2);
                p.add(b3);
                p.add(b4);
                p.setLayout(new GridLayout(4,2));
                add(p);
                pack();
                setVisible(true);
                                }

        public static void main(String args[])
        {
                da21 d=new da21();
                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        con=DriverManager.getConnection("jdbc:odbc:we");
                        sa=con.createStatement();
                        rs=sa.executeQuery("select roll,sname from student");
                        rs.next();
                        }
                        catch(Exception e)
                        {
                        System.out.println("error"+e);
```

```
				}
				d.show(rs);
				}
				public void actionPerformed(ActionEvent e2)
				{
					if(e2.getSource()==b1)
				{
				try
				{
					System.exit(0);
				}
				catch(Exception e)
				{
				}
					}
					else if(e2.getSource()==b2)
					{
						try
						{
							rs.next();

						}
						catch(Exception e)
						{
						}
						show(rs);
					}

					else if(e2.getSource()==b3)
					{
						try
						{
							sa.executeUpdate("insert into student(roll,sname)
values("+t1.getText()+",'"+t2.getText()+"')");
						System.out.println("one record saved");
							}
							catch(Exception e)
					{
						System.out.println(e.toString());
					}
					show(rs);
				}
				else if(e2.getSource()==b4)
				{
					try
					{
						t1.setText("");
						t2.setText("");
					}
					catch(Exception e)
					{
					}
				}

		}
		public void show(ResultSet rs)
		{
			try
			{
				t1.setText(rs.getString(1));
				t2.setText(rs.getString(2));
			}
			catch(Exception e)
			{
			//	System.out.println("error"+e);
			}
		}

}
```

==

The PreparedStatement Interface

When we need to perform the same operation several times, we have to think of ways to increase the efficiency of the program. In such a case, we can use a PreparedStatement for runtime efficiency. The execution of PreparedStatement object is faster than that of Statement objects as they are precompiled. PreparedStatement objects are also useful when you have to specify many arguments for a particular SQL command.

The PreparedStatement interface inherits from the Statement interface. However, there are two differences-

- A Statement object compiles and executes the SQL statement when we execute the program. In case of a **PreparedStatement object,** the SQL statement is precompiled.
- The SQL statement contained in a **PreparedStatement** object may have one or more IN parameters whose value is not specified when the SQL statement is created. Such a statement has a question mark ("?") as a placeholder for every IN parameter.

Program---

```
import java.sql.*;
import java.util.*;

class pre
{
		public static void main(String args[])
		{
			try

			{
				Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
			}
			catch(Exception e)
			{
				System.out.println(e.toString());

			}
			try
			{

				Connection con=DriverManager.getConnection("jdbc:odbc:ss");
				PreparedStatement ps=con.prepareStatement("update couse set hours=?
where coursetitle=?");
				ps.setInt(1,300);
				ps.setString(2,"CsIT");
				ps.executeUpdate();
				System.out.println("Recooord updated");
				Statement s=con.createStatement();
				String sa="select * from course";
				ResultSet rs=s.executeQuery(sa);
				while (rs.next())
				{
					System.out.print(rs.getInt(1)+" ");
					System.out.print(rs.getString(2)+" ");
					System.out.print(rs.getInt(3));
					System.out.println();
				}

			}
			catch (SQLException ce)
			{
				System.out.println(ce.toString());
			}
		}
}
```

The CallableStatement Interface

A stored procedure is a block of SQL code created and stored in a database. Stored procedure can be invoked using the **CallableStatement** object. This call is written in an escape syntax that may take one of two forms:

- With a result parameter
- Without a result parameter.

The return value from a stored procedure is passed to an OUT parameter. Both the forms have a different number of parameters used as input(IN parameter) , output(OUT parameter) or both (INOUT parameters). A question mark(?) is used to represent a placeholder for a parameter.

Program

```
import java.sql.*;
class cal
{
        public static void main(String args[])
        {
                        Connection con;
                        Statement sa;

                   ResultSet  rs;

                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        String s="jdbc:odbc:aa";
                        con=DriverManager.getConnection(s,"sa","");
                        String proc="CREATE PROCEDURE mm as select max(salary) from
employee";
                        sa=con.createStatement();
                        sa.execute(proc);
                        CallableStatement cs=con.prepareCall("call mm}");
                        rs=cs.executeQuery();
                        rs.next();
                        int x=rs.getInt(1);
                        System.out.println(x);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e.toString());
                                }
                                }
                                }
```