**ODBC (Open Database Connectivity)**

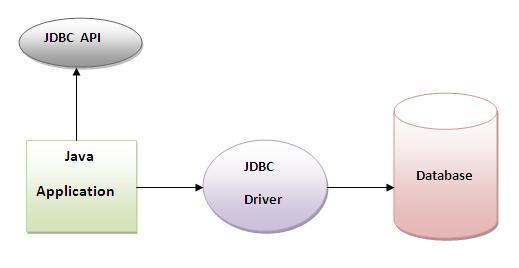
ODBC is an interface to access database management systems (DBMS). ODBC was developed by SQL Access Group in 1992 at a time there were no standard medium to communicate between a database and an application. It does not depend on a specific programming language or a database system or an operating system. Programmers can use ODBC interface to write applications that can query data from any database, regardless of the environment it is running on or the type of DBMS it uses.

ODBC is an open standard application programming interface (API) for accessing a database.

ODBC is an open interface which can be used by any application to communicate with any database system. ODBC is language independent.

**JDBC (Java database connectivity)**

JDBC is an interface that can be used by Java applications to access databases. Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



**Why use JDBC**

Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC (open database connectivity) 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).

**What is API**

API (Application programming interface) is a document that contains 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

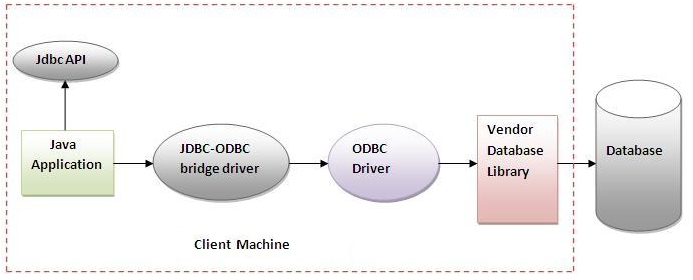
**JDBC Driver**

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.

 **Advantages:**

easy to use.

can be easily connected to any database.

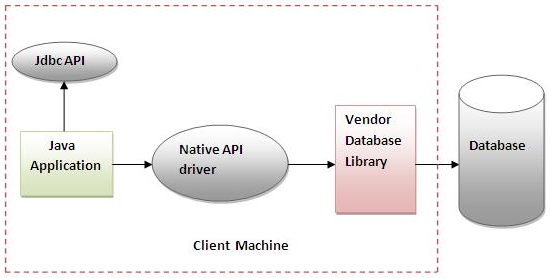
**Disadvantages**:

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.



**Advantage:**

performance upgraded than JDBC-ODBC bridge driver.

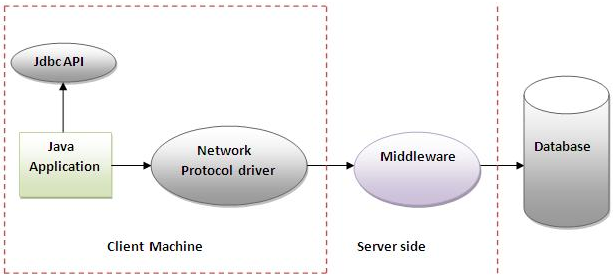
**Disadvantage:**

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.



**Advantage:**

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

**Disadvantages:**

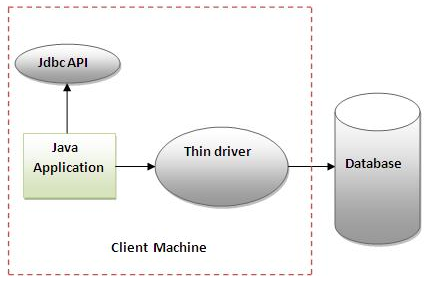
Network support is required on client machine.

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 depends on the Database.

**Steps to connect to the database in java**

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

* + Register the driver class
  + Creating connection
  + Creating statement
  + Executing queries
  + Closing connection

**1) Register the driver class**

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

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

**Example to register the OracleDriver class**

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

**2) Create the connection object**

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

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException

2) public static Connection getConnection(String url,String name,String password)

throws SQLException

Example to establish connection with the Oracle database

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","password");

**3) Create the Statement object**

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

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object

Statement stmt=con.createStatement();

**4) Execute the query**

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

**Syntax of executeQuery() method**

public ResultSet executeQuery(String sql)throws SQLException

**Example to execute query**

ResultSet rs=stmt.executeQuery("select \* from emp");

while(rs.next()){

System.out.println(rs.getInt(1)+" "+rs.getString(2));

}

**5) Close the connection object**

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

**Syntax of close() method**

public void close()throws SQLException

**Example to close connection**

con.close();

**Example to connect to the Oracle database**

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

* **Driver class:** The driver class for the oracle database is oracle.jdbc.driver.OracleDriver.
* **Connection URL:** The connection URL for the oracle10G database is jdbc:oracle:thin:@localhost:1521:xe where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these informations from the tnsnames.ora file.
* **Username:** The default username for the oracle database is system.
* **Password:** Password is given by the user at the time of installing the oracle database.

Let's first create a table in oracle database.

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

**Example to Connect Java Application with Oracle database**

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

**import** java.sql.\*;

**class** OracleCon

{

**public** **static** **void** main(String args[])

{

**Try**

{

//step1 load the driver class

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

//step2 create  the connection object

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object

Statement stmt=con.createStatement();

//step4 execute query

ResultSet rs=stmt.executeQuery("select \* from emp");

**while**(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object

con.close();

}

**catch**(Exception e)

{ System.out.println(e);}

  }

}

**ResultSet**

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. When the **Statement** object executes the query then it returns data in ResultSet. The object of **ResultSet** maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row. The java.sql.ResultSet interface represents the result set of a database query.

**Methods of ResultSet interface**

|  |  |
| --- | --- |
| **public boolean next():** | is used to move the cursor to the one row next from the current position. |
| **public boolean previous():** | is used to move the cursor to the one row previous from the current position. |
| **public boolean first():** | is used to move the cursor to the first row in result set object. |
| **public boolean last():** | is used to move the cursor to the last row in result set object. |
| **public int getInt(int columnIndex):** | is used to return the data of specified column index of the current row as int. |
| **Public int getInt(String columnName):** | is used to return the data of specified column name of the current row as int. |
| **Public String getString(int olumnIndex):** | is used to return the data of specified column index of the current row as String. |
| **Public String getString(String columnName):** | is used to return the data of specified column name of the current row as String. |

**Types of Statements**

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

They also define methods that help bridge data type differences between Java and SQL data types used in a database

There are 3 types of Statements

**Statement**:

A Statement is an interface that represents a SQL statement. we execute Statement objects. We need a Connection object to create a Statement object.

It can be used for general-purpose access to the database. It is useful when we are using static SQL statements at runtime. It is an interface that is used to implement simple SQL statements with no parameters.

Statement object to execute a SQL statement, we need to create one using the Connection object's createStatement( ) method.

Example:

Statement stmt = null;

try {

stmt = conn.createStatement( );

}

catch (SQLException e) {

}

finally {

}

we 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 we 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 we expect to get a result set, as we would with a SELECT statement.

Statement object calls to the close() method to ensure proper cleanup.

**PreparedStatement** **(Extends Statement)**:

It can be used when we plan to use the same SQL statement many times. The PreparedStatement interface accepts input parameters at runtime. It is used for precompiling SQL statements that might contain input parameters. This statement gives us the flexibility of supplying arguments dynamically.

Example:

PreparedStatement pstmt = null;

try {

String SQL = "Update Employees SET age = ? WHERE name = ?";

pstmt = conn.prepareStatement(SQL);

pstmt.setInt(1,10);// 1 specifies first parameter in query

pstmt.setString(2,”binod”);

int count=pstmt.executeUpdate();

}

catch (SQLException e) {

}

finally {

}

All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth.

**CallableStatement (Extends PreparedStatement):**

CallableStatement interface is used to call the database stored procedures and functions. It is used to execute stored procedures that may contain both input and output parameters.

Example of 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;

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.

Example

CallableStatement cstmt = null;

try {

String SQL = "{call getEmpName (?, ?)}";

cstmt = conn.prepareCall (SQL);

}

catch (SQLException e) {

}

finally {

}

**SQL(Structured Query Language)**

It is a standard language for storing, manipulating and retrieving data in relational databases.

All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

**SQL is widely popular because it offers the following advantages −**

* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.