**J2EE JDBC Concepts**

**Data and Database:**Data, in the context of databases, refers to all the single items that are stored in a database, either individually or as a set. Data in a database is primarily stored in database tables, which are organized into columns that dictate the data types stored therein. If the “Customers” table has a column titled “Telephone Number,” whose data type is defined as “Number,” then only numerals can be stored in that column.The database is a collection of inter-related data which is used to retrieve, insert and delete the data efficiently. It is also used to organize the data in the form of a table, schema, views, and reports, etc.

**Database Schema:**

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.



A database schema can be divided broadly into two categories −

* **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
* **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

**What is JDBC?**

The JDBC stands for Java Database Connectivity. It’s a specification for using data sources in Java applets and applications.

* It’s an API for using low-level JDBC drivers.
* It’s an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
* It’s based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems. The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. This interface is used by the developer so he/she doesn’t need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other Java packages such as java.awt. It’s not currently a part of the standard Java Developer’s Kit (JDK) distribution, but it is slated to be included as a standard part of the general Java API as the java.sql package. Soon after its official incorporation into the JDK and Java API, it will also become a standard package in Java-enabled Web browsers, though there is no definite timeframe for this inclusion. The exciting aspect of the JDBC is that the drivers necessary for connection to their respective databases do not require any pre- installation on the clients: A JDBC driver can be downloaded along with an applet!

**The JDBC Structure**

The JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the highlevel application interface. The lowlevel programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases. JDBC drivers are quite flexible: They can be local data sources or remote database servers. The implementation of the actual connection to the data source/database is left entirely to the JDBC driver. The structure of the JDBC includes these key concepts:

The goal of the JDBC is a DBMS independent interface, a “generic SQL database access framework,” and a uniform interface to different data sources.

•The programmer writes only one database interface; using JDBC, the program can access any data source without recoding.

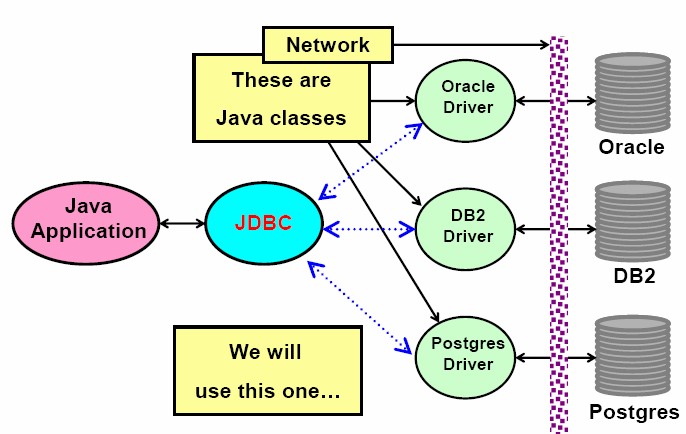


Fig. The JDBC Architecture

**JDBC drivers**

Sun has defined four categories of JDBC drivers. The categories delineate the differences in architecture for the drivers. One difference between architectures lies in whether a given driver is implemented in native code or in Java code. Native code means whatever machine code is supported by a particular hardware configuration. For example, a driver may be written in C and then compiled to run on a specific hardware platform. Another difference lies in how the driver makes the actual connection to the database. The four driver types are as follows:

# Type 1 Driver: JDBC/ODBC Bridge

This type uses bridge technology to connect a Java client to a third-party API such as Open DataBase Connectivity (ODBC). Sun's JDBC-ODBC bridge is an example of a Type 1 driver. These drivers are implemented using native .This driver connects Java to a Microsoft ODBC (Open Database Connectivity) data source.

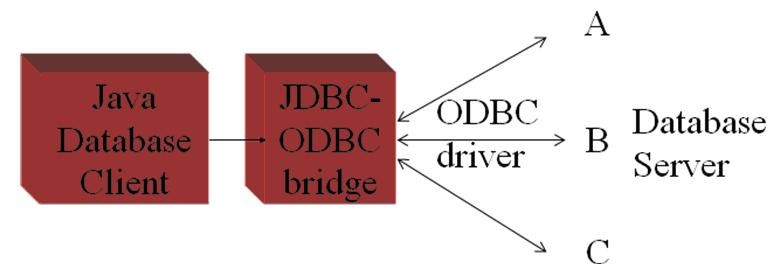


Fig. Type 1 Driver

The Java 2 Software Development Kit from Sun Microsystems, Inc. includes the JDBC-to-ODBC bridge driver (sun.jdbc.odbc.JdbcOdbcDriver). This driver typically requires the ODBC driver to be installed on the client computer and normally requires configuration of the ODBC data source. The bridge driver was introduced primarily to allow Java programmers to build data-driven Java applications before the database vendors had Type 3 and Type 4 drivers.

# Type 2 Driver: Native API Driver

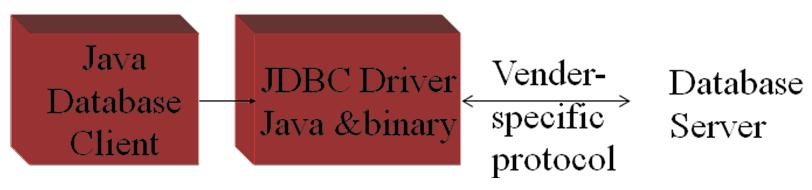


Fig. Type 2 Driver

This type of driver wraps a native API with Java classes. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver. Because a Type 2 driver is implemented using local native code, it is expected to have better performance than a pure Java driver. These drivers enable JDBC programs to use database-specific APIs (normally written in C or C++) that allow client programs to access databases via the Java Native Interface. This driver type translates JDBC into database-specific code. Type 2 drivers were introduced for reasons similar to the Type 1 ODBC bridge driver.

# Type 3 Driver: Network Protocol, Pure Java Driver

These drivers take JDBC requests and translate them into a network protocol that is not database specific. These requests are sent to a server, which translates the database requests into a database-specific protocol.

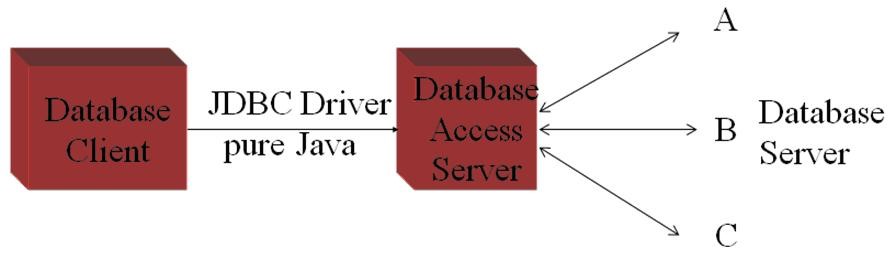


Fig. Type 3 Driver

This type of driver communicates using a network protocol to a middletier server. The middle tier in turn communicates to the database. Oracle does not provide a Type 3 driver. They do, however, have a program called Connection Manager that, when used in combination with Oracle's Type 4 driver, acts as a Type 3 driver in many respects.

# Type 4 Driver: Native Protocol, Pure Java Driver

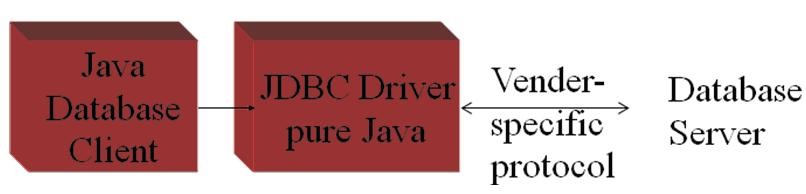


Fig. Type 4 Driver

These convert JDBC requests to database-specific network protocols, so that Java programs can connect directly to a database. This type of driver, written entirely in Java, communicates directly with the database. No local native code is required. Oracle's thin driver is an example of a Type 4 driver.

The JDBC API

The JDBC API is contained in two packages named java.sql and javax.sql. The java.sql package contains core Java objects of JDBC API. There are two distinct layers within the JDBC API: the application layer, which database application developers use and driver layer which the drivers vendors implement. The connection between application and driver layers is illustrated in figure below:

Driver Mana

g

e

r

Drive

r

Connection

PreparedStatement

Statement

CallableStatement

ResultSet

ResultSet

ResultSet

Fig. The JDBC API

There are four main interfaces that every driver layer must implement and one class that bridges the Application and driver layers. The four interfaces are Driver, Connection, Statement and ResultSet. The Driver interface implementation is where the connection to the database is made. In most applications, Driver is accessed through DriverManager class.

**The JDBC process**

# Accessing JDBC / ODBC Bridge with the database

Before actual performing the Java database application, we associate the connection of database source using JDBC – ODBC Bridge. The steps are as follows:

1. Go to Control Panel -> Administrative Tools -> Data Sources.
2. Open Data Sources ODBC icon.



1. Select the tab with heading “User DSN”.
2. Click on ‘Add’ button.
3. Select the appropriate driver as per the database to be used. (e.g. Microsoft ODBC driver for Oracle to access Oracle Database
4. Click finish button and the corresponding ODBC database setup window will appear.
5. Type DSN name and provide the required information such as user name and password for the database (.mdb files) of Microsoft Access Database etc. and click on OK button.
6. Our DSN name will get appeared in user data sources.

There are six different steps to use JDBC in our Java application program. These can be shown diagrammatically as below:

**Phase Task Relevant java.sql classes**

Load driver DriverManager

Create connection Connection

Generate SQL statements Statement

Process result data ResultSet

Terminate connection Connection

Release data structures Statement

1. Load the driver

Initialisation

Processing

Termination

1. Define and establish the Connection
2. Create a Statement object
3. Execute a query
4. Process the results
5. Close the connection

# Loading the JDBC driver

The JDBC drivers must be loaded before the Java application connects to the DBMS. The Class.forName() is used to load the JDBC driver. The developer must write routine that loads the JDBC / ODBC Bridge. The bridge driver called sun.jdbc.odbc.JdbcOdbcDriver. It is done in following way:

Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);

# Connect to the DBMS

After loading the driver the application must get connected to DBMS. For this we use DriverManager.getConnection() method. The DriverManager is highest class in Java.sql hierarchy and is responsible for managing driver related information.

The DriverManager.getConncetion() method is passed the URL of the database and user ID and password required by the database. The URL is the string object that contains the driver name that is being accessed by the Java program.

The DriverManager.getConncetion() method returns Connection interface that is used throughout the process to reference the database. The signature of this method is:

Connection DriverManager.getConncetion(String url,

String userID, String password);

Here, the URL format is specified as follows:

<protocol>:<subprotocol>:<dsn-name>

The ‘protocol’ is a JDBC protocol that is used to read the URL. The ‘subprotocol’ is JDBC driver name and ‘dsn-name’ is the name of the database that we provided while creating JDBC Bridge though control panel. We use the following URL for our application:

jdbc:odbc:*customer*

here, ‘customer’ is an example of DSN name given to our database. The user name and password are also provided at the time of creating DSN. It is not compulsory to provide the username and password. For example:

Conncetion con; con = DriverManager.getConnection(“jdbc:odbc:customer”,

“micro”, “pitch”);

# Create Statement object

The createStatement( ) method of Connection interface is used to create the Statement object which is then used to execute the query. For example:

Statement st = con.createStatement();

# Execute the query

The executeQuery() method of Statement object is used execute and process the query which returns the ResultSet object. ResultSet is the object which actually contains the result returned by the query. For example:

ResultSet rs = st.executeQuery(“select \* from customer”);

Here, the ‘customer’ is neither database name nor DSN name but it is a table name.

# Process the results

The ResultSet object is assigned the results received from the DBMS after the query is processed. The ResultSet object consists of methods used to interact with data that is returned by the DBMS to Java application program. For example, the next() method is used to proceed throughout the result set. It returns true, if the data is available in result set to read.

The ResultSet also contains several getXxx( ) methods to read the value from particular column of current row. For example, getString(“name”) will read the value from column ‘name’ in the form of string. Instead of passing column name as parameter, we can pass column as parameter also. Such as, getString(1). For example:

String name;

int age;

do

{

name = rs.getString(“name”);

age = rs.getInt(“age”);

System.out.println(name+“=”+age);

} while(rs.next());

# Terminate the Connection

The Connection to the DBMS is terminated by using the close() method of the Connection object once Java program has finished accessing the DBMS. The close( ) method throws as exception if problem is encountered when disengaging the DBMS. For example:

con.close();

The close() method of Statement object is used to close the statement object to stop the further processing.

Statement Objects

Once the connection to the database is opened, the Java application creates and sends a query to access data contained in the database. One of three type of statement objects is used to execute the query immediatelt. A PreparedStatement is used to execute the compiled query and

CallableStetement is used to execute the stored procedure.

# Statement object

The Statement object is used whenever a Java program needs to immediately execute a query without first having query compiled. The Statement contains three different methods depending upon the type of query these will be used.

1. **executeQuery()**

This method returns the ResultSet object that contains rows, columns and metadata that represent data requested by the query. Its signature is:

ResultSet executeQuery(String query);

Generally, this method is used to execute only the ‘SELECT’ query of the SQL.

1. **executeUpdate()**

This method is used to execute the queries that contain INSERT, DELETE and UPDATE statements. This method returns integer indicating the number of rows that were updated by the query. Its signature is:

int executeUpdate(String query);

For example:

int rows = st.executeUpdate("DELETE FROM EMPLOYEES

WHERE STATUS=0");

1. **execute()**

It executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts we must then use the methods getResultSet( ) or getUpdateCount( ) to retrieve the result, and getMoreResults( ) to move to any subsequent result(s). Signature is as follows:

public boolean execute(String sql)

For example:

if(st.execute()) rs = st.getResultSet(); Signatures of other methods:

public ResultSet getResultSet() public int getUpdateCount()

public boolean getMoreResults()

# PreparedStatement object

A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object’s execution method is called. Compiling a query is an overhead that is acceptable if the query is called once. However, compiling process can become an expensive overhead if the query is executed several times by the same program during the same session.

A SQL query can be precompiled and executed by using the

PreparedStatement object. In such cases a query is created similar to other queries. However, a question mark is given on the place for the value that is inserted into the query after it is compiled. It is the value that changes each time the query is executed.

For doing this process, we need to construct the query with question marks such as,

“select \* from nation where population > ?”

Such type of the query is passed as the parameter to the prepareStatement( ) method of the Connection object which then returns the PreparedStatement object. For example:

String query = “select \* from nation where population > ?”;

PreparedStatement ps = prepareStatement(query);

Once the PreparedStatement object is obtained, the setXxx( ) methods of it can be used to replace question mark with the value passed to setXxx() method. There are a number of setXxx() methods available in

PreparedStatement object, each of which specifies the data type of value that is being passed to setXxx() method. For example, considering the above query again,

ps.setInt(1, 100000);

This method requires two parameters. First parameter is an integer that identifies position of the question mark placeholder and second is the value that replaces the question mark. If the query contains two question marks we have to pass second value also using setXxx() method.

Now, we need to use appropriate execute method depending upon type of the query without any parameters. Such as,

ResultSet rs = ps.executeQuery();

This will generate the ResultSet object as the execution of the query. The PreparedStatement contain all three execute methods but without any parameters as given below:

ResultSet executeQuery( ) int executeUpdate( )

boolean execute( )

The setXxx( ) methods:

void setBoolean(int index, boolean value); void setByte(int index, byte value); void setDate(int index, Date value); void setDouble(int index, double value); void setFloat(int index, float value); void setInt(int index, int value); void setLong(int index, long value); void setObject(int index, Object value); void setShort(int index, short value);

void setString(int index, String value);

Example:

Consider the following database:



import java.sql.\*; class StudentData

{

public static void main(String args[])

{ try

{

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

Connection con =

DriverManager.getConnection("jdbc:odbc:stud");

PreparedStatement ps = con.prepareStatement("select \*

from Student where Marks > ?");

ps.setInt(1,70); //set question marks place holder

ResultSet rs = ps.executeQuery(); //execute System.out.println("Students having marks > 70 are:");

while(rs.next())

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

con.close();

}

catch(Exception e){ }

}

}

Output:

Students having marks > 70 are:

Rakhee

Rahul

Karthik

# CallableStatement Object

The CallableStatement is used to call the stored procedures from within a JDBC application program. A stored procedure is a block of code and is identified by a unique name. The type style of code depends upon the DBMS vendor and can be written in PL/SQL, Transact-SQL, C or another programming language. The stored procedure is executed by invoking name of the stored procedure. For example, a stored procedure written in PL/SQL as given below:

create or replace procedure "INSERTR"

id IN NUMBER,

name IN VARCHAR2)

is

begin

insert into user420 values(id,name);

end;

The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are IN, OUT, INOUT. The IN parameter contains the data that needs to be passed to the stored procedure whose value is assigned using setXxx() method. Whereas, OUT parameter contains the value returned by the stored procedure, if any. The OUT parameter must be registered using registerOutParameter() method and afterwards this is retrieved by using getXxx() method. The INOUT parameter is a single parameter that is used to both pass information and retrieve information from a stored procedure.

**import** java.sql.\*;

**public** **class** Proc {

**public** **static** **void** main(String[] args) **throws** Exception{

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

Connection con=DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:xe",

"system","oracle");

CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");

stmt.setInt(1,1011);

stmt.setString(2,"Amit");

stmt.execute();

System.out.println("success");

}

}

After establishing the connection, the prepareCall() method of the Connection object is passed

with query of stored procedure call. It returns the object of CallableStatement.

**ResultSet:**

When we execute a query to retrieve data from a table using a java application , the output of the query is stored in a ResultSet object in a tabular format.A ResultSet objects maintains a cursor that enables you to move through the rows stored in a ResultSet object.By Default , the ResultSet object maintains a cursor that moves in the forward direction only. As a result , it moves from the first row to the last row in the ResultSet. We cannot update the default ResultSet object.The cursor in the ResultSet object initially points before the first row.The *java.sql.ResultSet* interface represents the result set of a database query.

**ResultSet Types:**

Read only : Allows you to only read the rows in a ResultSet object.

Forward only : Allows you to move the result set cursor from first row to last row in forward direction only.

Scrollable : Allows you to move the result set cursor forward or backward through the result set.

Updatable : Allows you to update the result set rows retrieved from a database table.We can specify the type of a ResultSet object using the createStatement () of the Connection interface.

The createStatement() accepts ResultSet fields as parameters to create different types of the ResultSet objects.

**Result Fields:**

**TYPE\_SCROLL\_SENSITIVE :** Specifies that the cursor of the ResultSet object is scrollable and it reflects the changes in the data made by the other users.

**TYPE\_SCROLL\_INSENSITIVE :** Specifies that the cursor of the ResultSet object is scrollable and it does not reflect changes in the data made by the other users.

**TYPE\_FORWARD\_ONLY :** Specifies that the cursor of the ResultSet object moves in forward direction only from the first row to the last row.

**CONCUR\_READ\_ONLY :** Specifies the concurrency mode that does not allow you to update the ResultSet object.

**CONCUR\_UPDATABLE :** specifies the concurrency mode that allows you to update the ResultSet object.

**HOLD\_CURSORS\_OVER\_COMMIT :** Specifies that a ResultSet object should not be closed after data is committed to the database.

**CLOSE\_CURSORS\_AT\_COMMIT :** Specifies that a Result Set object should be closed after data is committed to the database.

**Methods of ResultSet:**

boolean first () : Shifts the control of a result set cursor to the first row of the result set.

boolean isFirst() : Determines whether the result set cursors points to the first row of the result set.

boolean beforeFirst() : Shifts the control of a result set cursorbefore the first row of the result set.

boolean isBeforeFirst () : Determines whether the result set cursor points before the first row of the result set.

boolean last() : Shifts the control of a result set cursor to the last row of the result set.

**boolean isLast () :** Determines whether the result set cursor points to the last row of the result set.

**boolean afterLast() :** Shifts the control of a result set cursor after the last row of the result set.

**boolean isAfterLast () :** Determines whether the result set cursor points after the last row of the result set.

**boolean previous () :** Shifts the control of a result set cursoe to the previous row of the result set.

**boolean absolute (int i) :** Shifts the control of a result set cursor to the row number that you specify as a paramter.

**boolean relative (int i) :** Shifts the control of a result set cursor , forward or backward , relative to he row number that you specify as a paramter .This method accepts either a positive or a negative values as a paramter

**Example of ResultSet:**

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,ResultSet.CONCUR\_READ\_ONLY);

ResultSet rs = stmt.executeQuery(“SELECT \* FROM authors”);

if(rs.isBeforeFirst()== true)

System.out.println(“Result set cursor id before first row in result set”);

if (rs.first() == true)

System.out.println(rs.getString(1) + “,” + rs.getString(2)+”,”+rs.getString(3));

System.out.println(“using absolute() method”); rs.absolute(4);

int rowcount = rs.getRow() ;

System.out.println(“rowNum should be 4 ” + rowcount);

JDBC allows you to create an updatable result set that enables you to modify the rows in the result set.

**Updatable ResultSet:**

**The following methods used with updatable result set:**

voidupdateRow () : Updates a row of the current ResultSet object and the underlying database tables.

void insertRow () : Inserts a row in the current ResultSet object and underlying database table.

void deleteRow () : Deletes a row from the current ResultSet object and underlying database table.

void updateString () : Updates the specified column with the given string value.

void updateInt() : Updates the specified column with the given int value.

Example: Statement stmt = con.createStatement ( ResultSet.TYPE\_SCROLL\_SENSITIVE , ResultSet.CONCUR\_UPDATABLE);

ResultSet rs = stmt.executeQuery( “SELECT au\_id , city , state FROM authors WHERE au\_id = ‘123’ ”) ;

rs.next () ;

rs.updateString(“state” , “MAHARASHTRA”) ;

rs.updateString(“city” , “Pune”);

**Metadata:**

Metadata is data about data. A J2EE component can access metadata by using the DatabaseMetaData interface. The DatabaseMetaData interface is used to retrieve information about databases, tables, columns and indexes among other information about the DBMS. A J2EE component retrieves metadata about the database by calling the getMetaData() method of the Connection object. The getMetaData() method returns a DatabaseMetaData object that contains information about the database and its components.

Some of the commonly used DatabaseMetaData object method:

getDatabaseProductName(): returns the product name of the database

getUserName(): returns the username

getURL(): returns the URL of the database

getSchemas(): returns all the schema names available in this database

getPrimaryKeys(): returns the primary keys

getProcedures(): returns stored procedure names

getTables: returns names of tables in the database

**ResultSet Metadata:**

There are two types of metadata that can be retrieved from the DBMS. Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of the ResultSet object.

ResultSetMetaData rm =Result.getMetaData

The more commonly used methods are as follows:

getColumnCount() Returns the name of the columns contained in the ResultSet

getColumnName(int number): returns the data type of the column specified by the column number