JDBC

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

What is Driver?

Driver is software written in any language which converts one form of function call into another form of function call.

There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver-Type1
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

Type 1 – JDBC-ODBC Bridge Driver

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

Disadvantages

- Because this driver is using ODBC software, JDBC programs connecting through this Driver are portable to all Windows OS but not on non-windows OSs

- With this driver we can connect only to local DB (same machine DB) but not DBs running in other Systems. (Not suitable for network DBs)

Type 2 – JDBC-Native API

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

Disadvantage:

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

Network Protocol driver

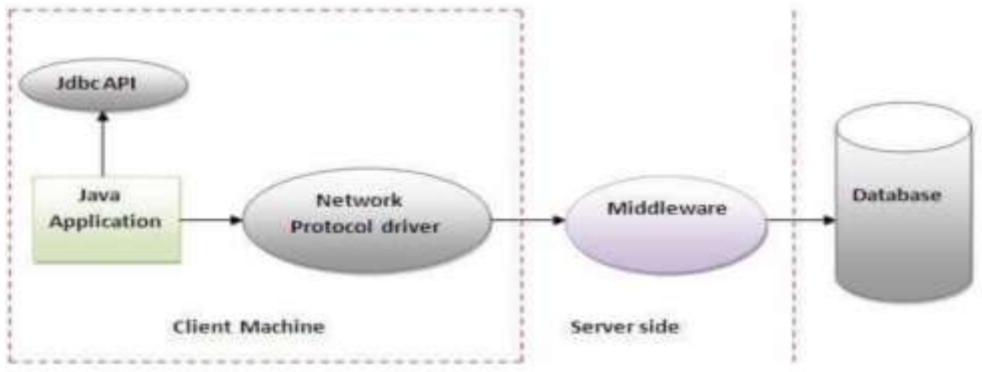
The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

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.



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.

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.
- We can connect to remote database as well.

Disadvantage:

- Drivers depend on the Database.

Below are the Driver names for various Databases

DataBase Name	JDBC driver Name
MySQL	com.mysql.jdbc.Driver
Oracle	oracle.jdbc.driver.OracleDriver
Microsoft SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver
MS Access	net.ucanaccess.jdbc.UcanaccessDriver
PostgreSQL	org.postgresql.Driver
SyBase	com.sybase.jdbcSybDriver

Javax.sql.*:

This package is a JDBC extension API which provides server-side data access and processing in Java applications.

Classes/ Interfaces	Description
CommonDataSource	It is an interface which define the common methods between DataSource, XADataSource and ConnectionPoolDataSource
ConnectionPoolDataSource	It is a factory of PooledConnection objects
DataSource	It is a factory of Connections to a physical datasource
PooledConnection	Used to manage the Connection pool
RowSet	Provides support to the JDBC API for Java Components bean model
RowSetMetaData	It specifies the information about the columns in the RowSet object
ConnectionEvent	Provides the details about the occurrence about the connection-related events
ConnectionEventListener	Used to register the PooledConnection object events
RowSetEvent	It generates when an event occurs to the RowSet object
StaementEvent	It is sent to all the StatementEventListeners which are registered when PooledConnection is generated

Connection Interface:

- A Connection is nothing but a session between Java application and the Database.
- Its available in the java.sql package.
- It is an interface with a factory of Statement, PreparedStatement, MetaData etc.,
- It also provides methods related to the Transaction Management like commit(), rollback() etc.,

Driver Manager class:

- The Driver Manager class acts as an interface between the user and the drivers.
- It keeps track of all the set of drivers and maintains the registered drivers and also handles the establishment of connection between the database and the driver.

Statement:

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

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

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

Statement vs PreparedStatement:

1) Performance

-Statement carries new SQL statement to DB everytime. Takes more time to execute each SQL statement

-PreparedStatement carries only one SQL statement once to DB, and carries arguments of the execution plan for every executeUpdate() method request. SQL execution time reduces, performance increases

2) Statement doesnot associate with any SQL statement, each time it takes new SQL statement and executes

-PreparedStatement is associated with only one SQL statement.s

3) We must use Statement object if every time we want to execute new SQL statement

If the SQL statement want to be executed for many times, PreparedStatement is preferable.

4) Especially for INSERT operation Statement would be confusing and error prone

PreparedStatement makes INSERT operation syntax simple

Main methods in Jdbc:

There are four important methods for executing the SQL query.

- ResultSetexecuteQuery (String sql)

This is the method in the Statement interface and is used to retrieve the results from the database. It is similar to the SELECT query in SQL.

- int executeUpdate (String sql)

This is used for specified operations like INSERT, UPDATE, DELETE (DML statements) and DDL statements that return nothing. It is mostly used to insert and update the record entries in the database.

- boolean execute (String sql)

This is used to execute the SQL query. It returns TRUE if it is a SELECT query. It returns FALSE if t s UPDATE or INSERT query.

- int[] executeBatch ()

This method is used to execute the batch of SQL queries. If all the queries are successful then, it returns the array of update counts. It is mostly used to insert or update the records in bulk.

- Retrieving the Results:

The executeQuery() method in the Statement interface will return the results in the form of a ResultSet Object. The returned object will never be NULL even if there are no matching records in the database.

ResultSetrs = statement1.executeQuery(QUERY);

This method is only used for SELECT query. If any INSERT/UPDATE query is given as input, then it throws the SQLException saying that ‘this method cannot be used for UPDATE’. ResultSet is an Enumerator Object.

- Close the Connection:

Finally, after performing various operations and manipulations on the database. Now, we need to close the connection.

The resources opened should be closed as we may end-up getting out of connection exceptions and errors at a later point of time. If we close the connection object, then the Statement and ResultSet objects are going to be automatically closed.

conn.close();

Types opf Resultsets:

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

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

Concurrency of ResultSet

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

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

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including −

S.N.	Methods & Description

1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException** Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException**

	Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Example:

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

 public class TYPE_FORWARD_ONLY_ResultSetType {

    public static void main(String... arg) {

          Connection con = null;

          PreparedStatement prepStmt = null;

          ResultSet rs = null;

          try {

                // registering Oracle driver class

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

                 // getting connection

                con = DriverManager.getConnection(

                            "jdbc:oracle:thin:@localhost:1521:orcl",

                            "srini", "Oracle123");

                System.out.println("Connection established successfully!");

               //ResultSet type = TYPE_FORWARD_ONLY

                prepStmt = con.prepareStatement("select NAME from EMPLOYEE",

                            ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);



                rs = prepStmt.executeQuery();
```

```
            while (rs.next()) {

                System.out.println(rs.getString("NAME"));

            }

            System.out.println("\nTYPE_FORWARD_ONLY - "+rs.getType()); //1003 is
TYPE_FORWARD_ONLY

        } catch (ClassNotFoundException e) {

            e.printStackTrace();

        } catch (SQLException e) {

            e.printStackTrace();

        }

        finally{

            try {

                if(rs!=null) rs.close(); //close resultSet

                if(prepStmt!=null) prepStmt.close(); //close PreparedStatement

                if(con!=null) con.close(); // close connection

            } catch (SQLException e) {

                e.printStackTrace();

            }

        }

    }

}
```