**SQL API**

* **JDBC Architecture :-**

The JDBC API supports both two-tier and three-tier processing models for database access.

1. **Two-tier Architecture for Data Access:-**



In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server.

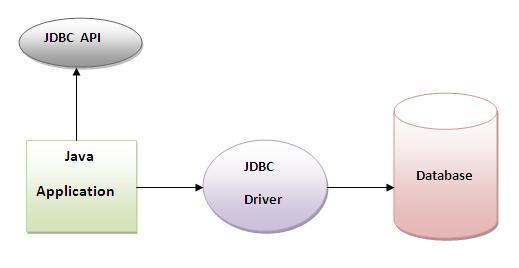
**2) Three-tier Architecture for Data Access :-**



In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

* **JDBC Drivers :-**

JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



|  |
| --- |
| 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. |



**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. |



**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.



**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. |



* **Connect to the Database :-**

There are 5 Steps to connect to the database in java

* 1. [Register the driver class](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step1)
  2. [Create the connection object](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step2)
  3. [Create the Statement object](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step3)
  4. [Execute the query](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step4)
  5. [Close the connection object](http://www.javatpoint.com/steps-to-connect-to-the-database-in-java#step5)

|  |
| --- |
|  |

**1) Register the driver class :-**

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

**Syntax:**

public static void forName(String className)throws ClassNotFoundException

**Example:**

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 :**

1. public static Connection getConnection(String url)throws SQLException
2. public static Connection getConnection(String url,String name,String password)

throws SQLException

**Example :**

Connection con=DriverManager.getConnection

("jdbc:oracle:thin:@localhost:1521:ORCL","system","dcs");

**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** :

public Statement createStatement()throws SQLException

**Example** :

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 :**

public ResultSet executeQuery(String sql)throws SQLException

**Example :**

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** :

public void close()throws SQLException

**Example** :

con.close();

* **Example to Connect Java Application with 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: ORCL","system","dcs");

//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);

}

}

}

* **To connect java application with the oracle database ojdbc14.jarfile is required to be loaded.**

Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

|  |
| --- |
| Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here. |

2) set classpath:

|  |
| --- |
| There are two ways to set the classpath:   * temporary * permanent |

*How to set the temporary classpath:*

|  |
| --- |
| Firstly, search the ojdbc14.jar file then open command prompt and write: |
|  |

C:>set classpath=c:\folder\ojdbc14.jar;.;

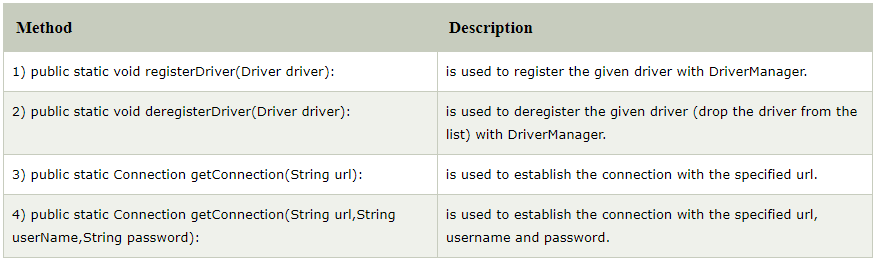
*How to set the permanent classpath:*

Go to environment variable then click on new tab. In variable name write classpath and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

* **DriverManager class:-**

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Commonly used methods of DriverManager class:



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Connection Interface:- A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(),rollback() etc. By default, connection commits the changes after executing queries.Commonly used methods of Connection interface:  |  | | --- | | **1) public Statement createStatement():** creates a statement object that can be used to execute SQL queries. | | **2) public Statement createStatement(int resultSetType,int resultSetConcurrency):** Creates a Statement object that will generate ResultSet objects with the given type and concurrency. | | **3) public void setAutoCommit(boolean status):** is used to set the commit status.By default it is true. | | **4) public void commit():** saves the changes made since the previous commit/rollback permanent. | | **5) public void rollback():** Drops all changes made since the previous commit/rollback. | | **6) public void close():** closes the connection and Releases a JDBC resources immediately. | |

# Statement interface :-

The Statement interface provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

|  |
| --- |
| **1) public ResultSet executeQuery(String sql):** is used to execute SELECT query. It returns the object of ResultSet. |
| **2) public int executeUpdate(String sql):** is used to execute specified query, it may be create, drop, insert, update, delete etc. |
| **3) public boolean execute(String sql):** is used to execute queries that may return multiple results. |
| **4) public int[] executeBatch():** is used to execute batch of commands. |

**Example of Statement interface**

Let’s see the simple example of Statement interface to insert, update and delete the record.

import java.sql.\*;

class FetchRecord

{

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

{

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

Connection con=DriverManager.getConnection

("jdbc:oracle:thin:@localhost:1521: ORCL","system","dcs");

Statement stmt=con.createStatement();

int result=stmt.executeUpdate("delete from emp where id=33");

System.out.println(result+" records deleted");

con.close();

}

}

# ResultSet interface :-

The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

#### By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE\_SCROLL\_INSENSITIVE or TYPE\_SCROLL\_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,

                     ResultSet.CONCUR\_UPDATABLE);

### Commonly used methods of ResultSet interface

|  |  |
| --- | --- |
| **1) public boolean next():** | is used to move the cursor to the one row next from the current position. |
| **2) public boolean previous():** | is used to move the cursor to the one row previous from the current position. |
| **3) public boolean first():** | is used to move the cursor to the first row in result set object. |
| **4) public boolean last():** | is used to move the cursor to the last row in result set object. |
| **5) public boolean absolute(int row):** | is used to move the cursor to the specified row number in the ResultSet object. |
| **6) public boolean relative(int row):** | is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative. |
| **7) public int getInt(int columnIndex):** | is used to return the data of specified column index of the current row as int. |
| **8) public int getInt(String columnName):** | is used to return the data of specified column name of the current row as int. |
| **9) public String getString(int columnIndex):** | is used to return the data of specified column index of the current row as String. |
| **10) public String getString(String columnName):** | is used to return the data of specified column name of the current row as String. |

|  |
| --- |
| [ResultSet example - records with columns.](http://tutorials.jenkov.com/images/java-jdbc/result-set-1.png) |
| ResultSet example - records with columns |

This ResultSet has 3 different columns (Name, Age, Gender), and 3 records with different values for each column.

You create a ResultSet by executing a Statement or PreparedStatement, like this:

1. Statement statement = connection.createStatement();
2. ResultSet result = statement.executeQuery("select \* from people");
3. Or like this:
4. String sql = "select \* from people";
5. PreparedStatement statement = connection.prepareStatement(sql);
6. ResultSet result = statement.executeQuery();

* **Iterating the ResultSet :-**

To iterate the ResultSet you use its next() method. The next() method returns true if the ResultSet has a next record, and moves the ResultSet to point to the next record. If there were no more records, next()returns false, and you can no longer. Once the next() method has returned false, you should not call it anymore. Doing so may result in an exception.

Here is an example of iterating a ResultSet using the next() method:

while(result.next()) {

// ... get column values from this record

}

* **Accessing Column Values :**

When iterating the ResultSet you want to access the column values of each record. You do so by calling one or more of the many getXXX() methods. You pass the name of the column to get the value of, to the manygetXXX() methods. For instance:

while(result.next()) {

result.getString ("name");

result.getInt ("age");

result.getBigDecimal("coefficient");

// etc.

}

# javax.sql:-

javax.sql is a JDBC API for the server side for accessing and processing the data from the databases typically a relational database using java. It is the essential part for J2EE. This API provides the facilities such as connection pooling, distributed transactions and row sets for the enterprise applications. An interface by name DataSource is provided in this API as an alternative to DriverManager to establish the connection.

DataSource and RowSet usage is direct, for the applications, where as connection pooling, distributed transactions are implemented by an infrastructure called middle-tier.

The javax.sql package provides for the following:

1. The DataSource interface as an alternative to the DriverManager for establishing a connection with a data source
2. Connection pooling and Statement pooling
3. Distributed transactions
4. Rowsets

* **Connection Pooling :-**
* Connections made via a DataSource object that is implemented to work with a middle tier connection pool manager will participate in connection pooling. This can improve performance dramatically because creating new connections is very expensive. Connection pooling allows a connection to be used and reused, thus cutting down substantially on the number of new connections that need to be created.
* Connection pooling is totally transparent. It is done automatically in the middle tier of a Java EE configuration, so from an application's viewpoint, no change in code is required. An application simply uses theDataSource.getConnection method to get the pooled connection and uses it the same way it uses any Connection object.
* **Rowsets :-**

The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

#### Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use
2. It is Scrollable and Updatable bydefault

The RowSet interface works with various other classes and interfaces behind the scenes. These can be grouped into three categories.

1. Event Notification :-
   * RowSetListener:  
     The RowSetListener interface is implemented by a component that wants to be notified about events that occur to a particular RowSet object. Such a component registers itself as a listener with a rowset via the RowSet.addRowSetListener method.

When the RowSet object changes one of its rows, changes all of it rows, or moves its cursor, it also notifies each listener that is registered with it. The listener reacts by carrying out its implementation of the notification method called on it.

* + RowSetEvent:  
    As part of its internal notification process, a RowSet object creates an instance of RowSetEvent and passes it to the listener. The listener can use this RowSetEvent object to find out which rowset had the event.

1. Metadata :-
   * RowSetMetaData  
     This interface, derived from the ResultSetMetaData interface, provides information about the columns in a RowSet object. An application can use RowSetMetaData methods to find out how many columns the rowset contains and what kind of data each column can contain.
2. The Reader/Writer Facility :-  
   A RowSet object that implements the RowSetInternal interface can call on the RowSetReader object associated with it to populate itself with data. It can also call on the RowSetWriter object associated with it to write any changes to its rows back to the data source from which it originally got the rows. A rowset that remains connected to its data source does not need to use a reader and writer because it can simply operate on the data source directly.