JDBC

# API

[Driver](https://docs.oracle.com/javase/8/docs/api/java/sql/Driver.html). Provides the API for registering and connecting drivers based on JDBC (every driver class must implement it). Generally used only by the DriverManager class

[DriverManager](https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html). To make a connection with a driver.

[Connection](https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html). Provides methods for creating statements and managing connections with a data source and their properties.

[Statement](https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html). Object used for executing a static SQL statement and returning the results it produces.

[ResultSet](https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html). Object used for retrieving and updating the results of a query.

To use the JDBC API with a particular DBMS, you need a JDBC driver to establish the communication between Java and the database. JDBC drivers are divided into four types:

Type 1: JDBC-ODBC (Open Database Connectivity) Bridge  
Type 2: Native-API, partly Java driver  
Type 3: Network-protocol, all-Java driver  
Type 4: Native-protocol, all-Java driver

All JDBC drivers implement four JDBC interfaces:

* Driver
* Connection
* Statement
* ResultSet.

The DriverManager class tracks the loaded JDBC drivers and creates the database connections.The Driver class IS loaded with this code:

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

# COMPONENTS

DriverManager connects an application to a data source by using a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the classpath.

A database URL varies depending on the DBMS used. For example, for ORACLE it looks like this:

jdbc:oracle:thin:hr/hr@localhost:1521/XE

Where localhost is the name of the server that is hosting the database, 1521 is the port number and hr is the user.

To connect to a data source using DriverManager, we need to get a Connection object with something like the following snippet:

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

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:hr/hr@localhost:1521/XE","system","PASSWORD");

Statement stmt=con.createStatement();

# PROCESS

To process an SQL statement with JDBC you have to:

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the ResultSet object.
5. Close the connection.

In code, it looks like this:

try{

//Establishing a connection

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

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:hr/hr@localhost:1521/XE","system","PASSWORD");

//Create a statemnent

Statement stmt=con.createStatement();

//Execute a query

ResultSet rs=stmt.executeQuery("select USERNAME from DBA\_USERS");

//Process the query

while(rs.next())

System.out.println(rs.getString(1));

//Close connection

stmt.close();

con.close();

}catch(Exception e){ System.out.println(e);}First, a connection with a MySQL database is established.

With a Connection object, you create a Statement object. There are three different kinds of statements:

* [Statement](https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html). Used to implement simple SQL statements with no parameters.
* [PreparedStatement](https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html). Used for precompiling SQL statements that might contain input parameters. It extends Statement.
* [CallableStatement](https://docs.oracle.com/javase/8/docs/api/java/sql/CallableStatement.html). Used to execute stored procedures that may contain both input and output parameters. It extends PreparedStatement.

To execute a query, call an execute method from Statement such as:

* execute(). Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
* executeQuery(). Returns one ResultSet object.
* executeUpdate(). Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.

With a ResultSet object, you can access the data. It acts as a cursor, pointing to one row of data and positioned before the first row at the beginning. Then you call, for example, the method next() to move the cursor forward by one row and you can get the data with getter methods that either take the column index (the first column is 1) or the column name. There are getter methods for a lot of types, for example:

int getInt(int columnIndex);

int getInt(String columnName);

long getLong(int columnIndex);

long getLong(String columnName);

String getString(int columnIndex);

String getString(String columnName);

BigDecimal getBigDecimal(int columnIndex);

BigDecimal getBigDecimal(String columnName);

When you are finished, call the method Statement.close() to immediately release the resources it's using. When you call this method, its ResultSet objects are also closed. If you're not going to need the connection anymore, you should also close the connection with Connection.close(). You can use a try-with-resources statement to automatically close Connection, Statement, and ResultSet objects, regardless of whether an SQLException has been thrown. Or you can close them manually in the finally block like this:

try {

Connection con = ...

Statement stmt = ...

// Do something with stmt

} catch(Exception e) {

e.printStackTrace();

} finally {

if (stmt != null) { stmt.close(); }

if (con != null) { con.close(); }

}