## JDBC

Java DataBase Connectivity

Used to communicate between java app and any other database

Common tasks associated are:

* Making connection with database
* Make sql or mysql statements
* Executing the sql or mysql queries in the database
* Viewing and modifying the resulting records

JDBC architecture:

Java app

JDBC API

JDBC Driver manager

JDBC Driver 

JDBC Driver 

Oracle

Mysql server

**Common JDBC components:**

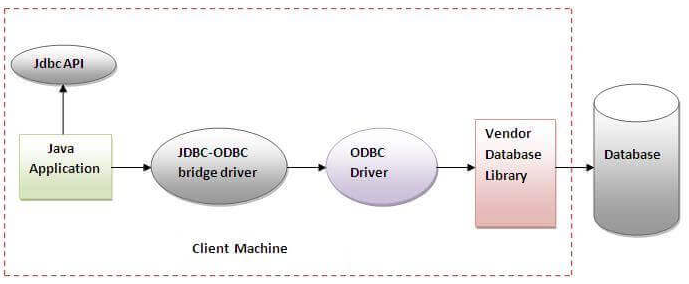
The JDBC API provides the following interfaces and classes −

* **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
* **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
* **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
* **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
* **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
* **SQLException:** This class handles any expception that occur in a database application.

**Types of jdbc drivers**

**1) JDBC-ODBC bridge driver(non java)**

|  |
| --- |
| 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 (type 4). |



Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**Advantages:**

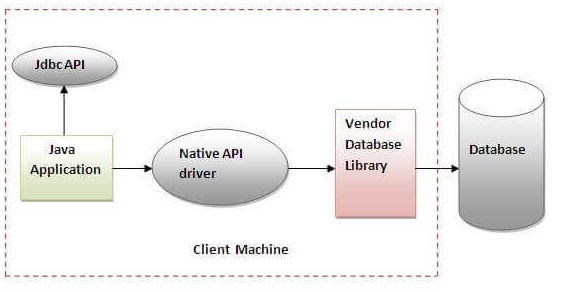
* easy to use.
* can be easily connected to any database.

**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 (partially written in java)(written in c/c++)**

|  |
| --- |
| 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:**

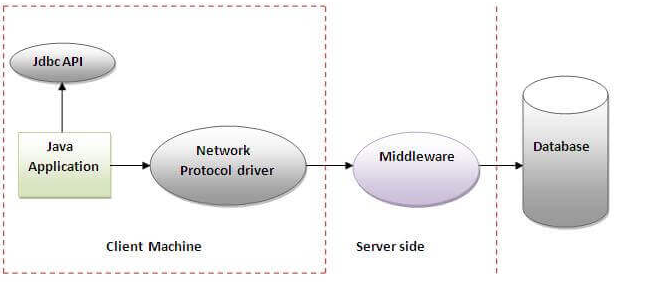
* Performance upgraded than JDBC-ODBC bridge driver.

**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:**

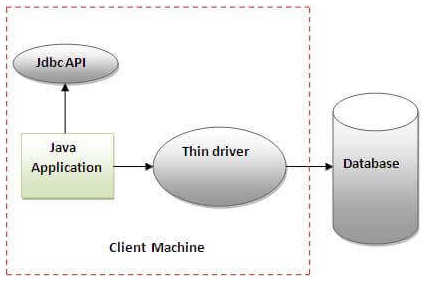
* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

**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 depend on the Database.

**Which Driver to use When?**

* If you are accessing one type of database, such as Oracle, mysql, the preferred driver type is type-4.
* If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
* Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
* The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

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

* 1. **Add the driver library**

**Use IDE**

**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.

Class.*forName*(**"com.mysql.jdbc.Driver"**);

### 2) Create the connection object

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

Connection con = DriverManager.  
 *getConnection*(**"jdbc:mysql://localhost:3306/jdbc"**,  
 *//dburl/dbname  
 //username password* **"root"**, **"root"**);

### 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. |

Statement stmnt = 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.

String sql = **"select** *\** **from test"**;  
ResultSet rs = stmnt.executeQuery(sql);

Note:

Use **executeQuery(string)** method to retrieve data from db, returns ResultSet object

Use **executeUpdate(string)** method to insert,update,delete data in db, returns int for the number of affected rows

### 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.

con.close();

**PrepareStatement Interface**

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query. The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

Eg:

String sql=**"insert into test(name,address,price,elective) values(?,?,?)"**;

PreparedStatement stmt = con.prepareStatement(sql);

*// parameterNo value*stmt.setInt( 1, 1 );

stmt.executeUpdate();

The important methods of PreparedStatement interface are given below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void setInt(int paramIndex, int value) | sets the integer value to the given parameter index. |
| public void setString(int paramIndex, String value) | sets the String value to the given parameter index. |
| public void setFloat(int paramIndex, float value) | sets the float value to the given parameter index. |
| public void setDouble(int paramIndex, double value) | sets the double value to the given parameter index. |
| public int executeUpdate() | executes the query. It is used for create, drop, insert, update, delete etc. |
| public ResultSet executeQuery() | executes the select query. It returns an instance of ResultSet. |

**SQL Exceptions**

Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion. An SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.

**try**{  
 *//sql codes*}**catch** (SQLException sqlException){  
 *//handle exception*}

**DDL and DML operation**

Data Definition Language (DDL) and Data Manipulation Language (DML) together forms a Database Language. The basic difference between DDL and DML is that **DDL** (Data Definition Language) is used to Specify the database schema database structure.

On the other hand, **DML** (Data Manipulation Language) is used to access, modify or retrieve the data from the database.

DDL query :

String createTable=**"CREATE TABLE "** +  
 **"test( "** +  
 **"id INT(10), "** +  
 **"name VARCHAR(100),"** +  
 **"price FLOAT(100),"** +  
 **"elective VARCHAR(100),"**;

stmnt.execute(sql);

DML query:

String sql = **"insert into test(name,address,price,elective) values"** + **"('Nick Fury','Everywhere',10000000,'SHIELD')"**;

stmnt.executeUpdate(sql);

stmnt.executeQuery(sql);

# Scrollable and updatable result sets

A scrollable updatable result set maintains a cursor which can both scroll and update rows.

Use scrollable insensitive result sets. To create a scrollable insensitive result set which is updatable, the statement has to be created with concurrency mode ResultSet.CONCUR\_UPDATABLE and type ResultSet.TYPE\_SCROLL\_INSENSITIVE.

Eg:

Statement stmnt = con.createStatement(ResultSet.***TYPE\_SCROLL\_INSENSITIVE***,  
 ResultSet.***CONCUR\_UPDATABLE***);

String sql = **"SELECT** *\** **"** +  
 **"FROM test"**;  
sql = **"select** *\** **from test"**;  
ResultSet rs = stmnt.executeQuery(sql);  
System.***out***.println(**"before"**);  
**while** (rs.next()) {  
 System.***out***.print(rs.getInt(**"tid"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"name"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"address"**) + **"\t"**);  
 System.***out***.print(rs.getInt(**"price"**) + **"\t"**);  
 System.***out***.println(rs.getString(**"elective"**) + **"\t"**);  
  
}  
rs.absolute(5); *// update the fifth row***int** newBonus = rs.getInt(**"price"**) + 100;  
rs.updateInt(**"price"**, newBonus);  
rs.updateRow();  
rs = stmnt.executeQuery(sql);  
System.***out***.println(**"after"**);  
**while** (rs.next()) {  
 System.***out***.print(rs.getInt(**"tid"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"name"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"address"**) + **"\t"**);  
 System.***out***.print(rs.getInt(**"price"**) + **"\t"**);  
 System.***out***.println(rs.getString(**"elective"**) + **"\t"**);  
  
}

**JDBC Transactions**

A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.

The classic example of when transactions are necessary is the example of bank accounts. You need to transfer $100 from one account to the other.

You do so by subtracting $100 from the first account, and adding $100 to the second account.

If this process fails after you have subtracted the $100 from the first bank account, the $100 are never added to the second bank account. The money is lost in cyber space.

To solve this problem the subtraction and addition of the $100 are grouped into a transaction. If the subtraction succeeds, but the addition fails, you can "rollback" the first subtraction. That way the database is left in the same state as before the subtraction was executed.

You start a transaction by this invocation:

con.setAutoCommit(**false**);

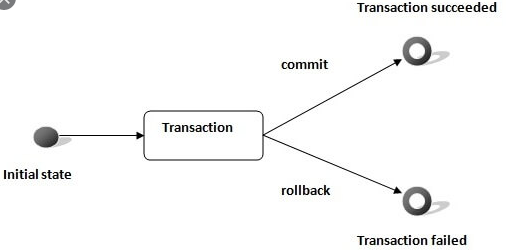
Now you can continue to perform database queries and updates. All these actions are part of the transaction.

If any action attempted within the transaction fails, you should rollback the transaction. This is done like this:

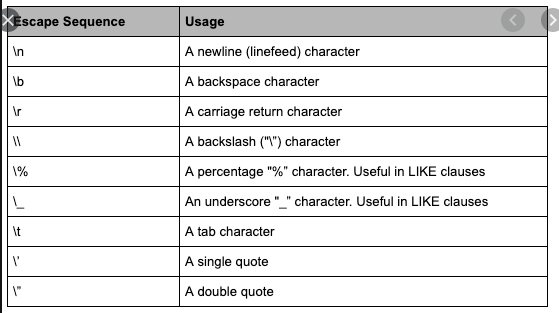
con.rollback();

If all actions succeed, you should commit the transaction. Committing the transaction makes the actions permanent in the database. Once committed, there is no going back. Committing the transaction is done like this:

con.commit();



**SQL Escapes**



**RowSet**

We have use ResultSet, which have certain limitations as it explicitly requires Connection and Statement explicitly. All those extra efforts can be reduced by using RowSet, which simplifies our program.

Eg:   
 Class.*forName*(**"com.mysql.jdbc.Driver"**);  
 JdbcRowSet rs = RowSetProvider.*newFactory*().createJdbcRowSet();  
 rs.setUrl(**"jdbc:mysql://localhost:3306/jdbc"**);  
 rs.setUsername(**"root"**);  
 rs.setPassword(**"root"**);  
  
 rs.setCommand(**"select \* from test"**);  
 rs.execute();  
  
 **while** (rs.next()) {  
 System.***out***.print(rs.getInt(**"tid"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"name"**) + **"\t"**);  
 System.***out***.print(rs.getString(**"address"**) + **"\t"**);  
 System.***out***.print(rs.getInt(**"price"**) + **"\t"**);  
 System.***out***.println(rs.getString(**"elective"**) + **"\t"**);  
 }

## CachedRowSet

A CachedRowSet object is a container for rows of data that caches its rows in memory, which makes it possible to operate (scroll and update) without keeping the database connection open all the time.

A CachedRowSet object makes use of a connection to the database only briefly: while it is reading data to populate itself with rows, and again while it is committing changes to the underlying database. So the rest of the time, a CachedRowSet object is disconnected, even while its data is being modified. Hence it is called disconnected row set.

**Writing LOB (Large Object)**

**LOB:**

* **BLOB: Byte Large Object**
* **CLOB: Character Large Object**

**Writing to database blob**

Class.*forName*(**"com.mysql.jdbc.Driver"**);  
  
Connection con = DriverManager.  
 *getConnection*(**"jdbc:mysql://localhost:3306/jdbc"**,  
 **"root"**, **"root"**);  
String sql=**"INSERT INTO "** +  
 **"files(name, image) "** +  
 **"VALUES( ?,?)"**;  
  
FileInputStream fis = **new** FileInputStream(**"pic.jpg"**);  
**byte**[] buffer = **new byte**[1024];  
ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  
**for** (**int** len; (len = fis.read(buffer)) != -1;) {  
 bos.write(buffer, 0, len);  
}  
PreparedStatement stmt = con.prepareStatement(sql);  
stmt.setString(1, **"test"**);  
stmt.setBytes(2, bos.toByteArray());  
stmt.executeUpdate();

**Reading FROM database Blob**

Class.*forName*(**"com.mysql.jdbc.Driver"**);  
  
 Connection con = DriverManager.  
 *getConnection*(**"jdbc:mysql://localhost:3306/jdbc"**,  
 **"root"**, **"root"**);  
  
 String sql=**"SELECT** *\** **"** +  
 **"FROM files "** +  
 **"WHERE id=2"**;  
 PreparedStatement stmt= con.prepareStatement(sql);  
ResultSet rs = stmt.executeQuery();  
 **while** (rs.next()) {  
 InputStream input = rs.getBinaryStream(**"image"**);  
 **byte**[] buffer = **new byte**[1024];  
 FileOutputStream fos= **new** FileOutputStream(**"result.jpg"**);  
 **while** (input.read(buffer) > 0) {  
 fos.write(buffer);  
 }  
 con.close();