**JDBC (Java Database Connectivity)**

If you want to interact with the database from C and C++ program then your program has to use database specific libraries directly inside the program because of this migration of the database become complicated i.e. if you want to change the database then your application has to be modified to make use of new database. This gives you the maintenance problem.

Microsoft has introduced an ODBC driver which can solve the above described problem. With ODBC, your program can interact with ODBC driver which internally interacts with underlying database. In this architecture yourprogram is not using database specific libraries directly. If you want to migrate the database then you can just change the ODBC configuration.

But the problem for ODBC is that ODBC is limited to windows operating system only and has the acceptable performance.

**Types of JDBC Driver**

**Type 1: JDBC-ODBC Bridge Driver**

**Type 2: Partial Native and Partial Java Driver**

**Type 3: Net Protocol Driver**

**Type 4: Pure Java Driver**

**Steps to write JDBC program:**

**(LECPEC)**

Step1: Load the driver class

Step2: Establish the connection

Step3: Create the Statement

Step4: Prepare the SQL Statement and Submit the DB engine

Step5: Get the Result and process.

Step6: Close all the resources properly

**Type 1: JDBC-ODBC Bridge Driver**

1. Name: JDBC ODBC Bridge Driver
2. Vender: Sun
3. Software: JDK
4. Path: E:\j2sdk1.4.2\bin;
5. Classpath: E:\j2sdk1.4.2\lib;
6. DriverClass: sun.jdbc.odbc.JdbcOdbcDriver
7. Url: jdbc:odbc:dsn
8. Username: DB Username
9. Password: DB password
10. Architecture:

Picture………..

Steps to configure ODBC data sources:

1. Open the Control Panel => and then click on Administrator Tools
2. Click on data Source (odbc)
3. Click on add button to create a new dsn.
4. When you click on add, a new window will be opened with list of database drivers.
5. Select the driver called Oracle in XE and Click on Finish button.
6. Fill the following:

Data Source Name: EXDSN

TNS Service Name: XE

UserID: SYSTEM

And then click on Test Connection button.

1. Provide password and click on OK button.
2. Click on OK of pop up window.
3. Click on OK of remaining windows.
4. Example of program:

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.Statement;

**publicclass** JdbcTest1 {

**publicstaticvoid** main(String[] args) {

Connection con = **null**;

Statement st = **null**;

**try**{

//Step1:load the driver

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

//Step2: get the connection

con = DriverManager.*getConnection*("jdbc:odbc:EXDSN","SYSTEM","suraj");

//Step3: create the statement

st = con.createStatement();

//Step4: prepare and submit the SQL statement

String sql = "insert into customer values('C-107','aa','aa@gmail.com','12345','Bangalore')";

**int** x = st.executeUpdate(sql);

//Step5: process result

**if**(x == 1){

System.*out*.println("Custome is inserted.");

} **else** {

System.*out*.println("Sorry Customer is not inserted.");

}

}**catch**(Exception e){

e.printStackTrace();

} **finally** {

**try**{

//Step6: Resource Cleanup

**if**(st != **null**){

st.close();

}

**if**(con != **null**){

con.close();

}

} **catch**(Exception e){

e.printStackTrace();

}

}

}

}

**Type2:**

1. Name: Partial Native and Partial Java Driver
2. Vender: Database veder (class111.jar for oracle)
3. Software: JDK, DB Software and libraries by db vendor
4. Path: E:\j2sdk1.4.2\bin;
5. Classpath: set the classpath to class111.jar for oracle
6. DriverClass: oracle.jdbc.driver.OracleDriver
7. Url: jdbc:oracle:oci8:@hostname:portNumber:serviceName
8. Username: DB Username
9. Password: DB password
10. Architecture:

Picture………..

Not for MySQL and Oracle10g because DB specific client libraries.

**Type3:**

1. Name: Net Protocol Driver
2. Vender: Java Soft(IDS)
3. Software: IDS Software
4. Path: ------------
5. Classpath: -----------
6. DriverClass: com.ids.driver.IdsDriver
7. Url: jdbc.ids://hostname/…….
8. Username: DB Username
9. Password: DB password
10. Architecture:

Picture………..

**Type4:**

1. Name: Pure Java Driver
2. Vender: Database Vender

Oracle:

1. Software: Oracle and class111.jar
2. Path: -------------
3. Classpath: set the classpath to class111.jar
4. Driver Class: oracle.jdbc.driver.OracleDriver
5. Url: jdbc:oracle:thin:@hostname:portNumber:serviceName

Ex: jdbc:oracle:thin:@localhost:1521:XE

1. Username: DB Username
2. Password: DB password
3. Architecture:

Picture………..

Mysql

1. Software: Oracle and class111.jar
2. Path: -------------
3. Classpath: set the classpath to mysql.jar
4. Driver Class: com.mysql.jdbc.driver
5. Url: jdbc:mysql:thin://hostname:portNumber/databaseName

Ex: jdbc:mysql://localhost/EXDB

1. Username: DB Username
2. Password: DB password
3. Architecture:

Picture………..

**JDBC Statements**

There are three types of jdbc statements:

1. Statement
2. PreparedStatement
3. CallableStatement
4. **Statement:**

* Statement is an interface available in “java.sql” package.
* Statement Object can be created using following methods of connection interface;

1. st = con.createStatement();

Creates a Statement object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it may be more efficient to use a PreparedStatement object.

Result sets created using the returned Statement object will by default be type TYPE\_FORWARD\_ONLY and have a concurrency level of CONCUR\_READ\_ONLY. The holdability of the created result sets can be determined by calling [getHoldability](eclipse-javadoc:%E2%98%82=TestJdbc/C:%5C/apps%5C/Java%5C/jdk1.6.0_37%5C/jre%5C/lib%5C/rt.jar%3Cjava.sql(Connection.class%E2%98%83Connection~createStatement%E2%98%82%E2%98%82getHoldability).

1. st = con.createStatement(int,int);

Creates a Statement object that will generate ResultSet objects with the given type and concurrency. This method is the same as the createStatement method above, but it allows the default result set type and concurrency to be overridden. The holdability of the created result sets can be determined by calling [getHoldability](eclipse-javadoc:%E2%98%82=TestJdbc/C:%5C/apps%5C/Java%5C/jdk1.6.0_37%5C/jre%5C/lib%5C/rt.jar%3Cjava.sql(Connection.class%E2%98%83Connection~createStatement~I~I%E2%98%82%E2%98%82getHoldability).

**Parameters:**

**resultSetType** a result set type; one of

ResultSet.TYPE\_FORWARD\_ONLY,

ResultSet.TYPE\_SCROLL\_INSENSITIVE, or

ResultSet.TYPE\_SCROLL\_SENSITIVE

**resultSetConcurrency** a concurrency type; one of

ResultSet.CONCUR\_READ\_ONLY or

ResultSet.CONCUR\_UPDATABLE

1. st = con.createStatement(int,int,int);

Creates a Statement object that will generate ResultSet objects with the given type, concurrency, and holdability. This method is the same as the createStatement method above, but it allows the default result set type, concurrency, and holdability to be overridden.

**Parameters:**

**resultSetType** one of the following ResultSet constants:

ResultSet.TYPE\_FORWARD\_ONLY,

ResultSet.TYPE\_SCROLL\_INSENSITIVE, or

ResultSet.TYPE\_SCROLL\_SENSITIVE

**resultSetConcurrency** one of the following ResultSet constants: ResultSet.CONCUR\_READ\_ONLY or

ResultSet.CONCUR\_UPDATABLE

**resultSetHoldability** one of the following ResultSet constants: ResultSet.HOLD\_CURSORS\_OVER\_COMMIT or

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT

* Once Jdbc Statement object is created you can use one of the following methods defined in Statement interface to submit SQL statement to the database engine.

1. boolean execute(String sql)
2. int executeUpdate(String sql)
3. ResultSet executeQuery(String sql)
4. **boolean execute(String sql):**

* You can use execute() method to submit insert, update, delete sql statements.
* execute() method returns boolean value which represents whether the operation completed successfully or not.

**Returns:**

true if the first result is a ResultSet object; false if it is an update count or there are no results

1. **int excuteUpdate(String sql)**

* You can use executeUpdate() to submit insert, update, delete statements.
* executeUpdate() method returns integer which represents number of rows affected by the given SQL statements i.e. number of rows inserted or deleted or updated.

1. **ResultSet executeQuery(String sql)**

* You can use executeQuery() to execute the select SQL statements.
* executeQuery() returns the ResultSet which can store multiple records return by select statement.

Using one Statement object you can submit different types of SQL statements.

For eg:

Statement st = con.createStatement();

int x = st.executeUpdate(sql1);

int y = st.executeUpdate(sql2);

ResultSet rs = st.executeQuery(sql3);

boolean b = st.execute(sql4);

When you submit SQL statement using JDBC statement then submitted query will be compiled and executed every time.

Request time = 5ms

Compile = 5ms

Run = 5ms

Response time = 5ms

1 sql = 20ms

1000 sql = 20000 ms

Example1:

**createtable** usr(

userid number(20,0),

username **varchar**(20)**constraint** username\_uk **unique**,

password **varchar**(20) **notnull**,

**constraint** userid\_pk **primarykey**(userid)

);

**insertinto** usr **values**(1001,'username1','password1');

**package** com.jdbc;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

/\*\*

\* **@author**surajkumar

\*

\* This class shows the example of Statement

\* for inserting, updating, fetching and deleting the record.

\*

\*/

**publicclass** Example1 {

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

Connection con = **null**;

Statement st = **null**;

ResultSet rs = **null**;

**try**{

// load jdbc Driver

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

// establishing the connection using DriverManger class.

con = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe", "example", "example");

System.*out*.println(con);

//create a Statement using connection object

st = con.createStatement();

//inserting a record into table

//preparing query

String insert = "insert into usr values(1006,'username6','password6')";

//execute statement

**int** i = st.executeUpdate(insert);

**if**(i == 1){

System.*out*.println("One record inserted.");

} **else**{

System.*out*.println("Error in insertion....");

}

// updating the record

String update = "update usr set password = 'password@1' where username = 'username1'";

**int** result = st.executeUpdate(update);

**if**(result > 0){

System.*out*.println("one record updated.");

} **else**{

System.*out*.println("error in update the record");

}

// fetching the record

String fetch = "select \* from usr";

rs = st.executeQuery(fetch);

**while**(rs.next()){

//System.out.println("userId = " + rs.getInt("userid") + " username = " + rs.getString("username") + " password = " + rs.getString("password"));

System.*out*.println("userId = " + rs.getInt(1) + " username = " + rs.getString(2) + " password = " + rs.getString(3));

}

// delete the record

String delete = "delete from usr where username <> 'username1'";

**boolean**d = st.execute(delete);

rs = st.executeQuery(fetch);

**while**(rs.next()){

//System.out.println("userId = " + rs.getInt("userid") + " username = " + rs.getString("username") + " password = " + rs.getString("password"));

System.*out*.println("userId = " + rs.getInt(1) + " username = " + rs.getString(2) + " password = " + rs.getString(3));

}

}**catch**(Exception e){

e.printStackTrace();

}**finally**{

**try** {

st.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**try** {

con.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

1. **PreparedStatement:**

* PreparedStatement is an interface available in “java.sql” package and is extending “Statement” interface.
* You can create the “PreparedStatement” object using, one of the following methods of Connection interface.

1. PreparedStatement prepareStatment(String s)
2. PreparedStatement prepareStatment(String s, int i, int j)
3. PreparedStatement prepareStatment(String s, int i, int j, int k)

* Once Jdbc PreparedStatement Object is created, you can use one of the following methods defined in PreparedStatement to submit SQL statement to SQL engine.

1. boolean execute()
2. int executeUpdate()
3. ResutlSet executeQuery()

* Using one PreparedStatement object, you can submit only one type of SQL statement to database engine, because in this case SQL Statement is prepared at the time of PreparedStatement object’s creation.

For eg:

PreparedStatement ps1 = con.prepareStatement(sql1);

ps1.executeUpdate();

PreparedStatement ps2 = con.prepareStatement(sql2);

ps1.executeUpdate();

* When you submit SQL statement using PreparedStatement object, then Query will be compiled only once and the same will be executed directly every time.

**Placeholder mechanism:**

ps = con.prepareStatement(“select \* from customer where cname = ? and phone = ?”);

ps.setString(1, “aaaa”);

ps.setLong(2, “11111111”);

rs = ps.executeQuery();

ps = con.prepareStatement(“insert into customer values(?,?,?,?)”);

ps.setInt(1, 1000);

ps.setString(2, “aaaa”);

ps.setString(3, “aaaa@gmail.com”);

ps.setLong(4, 99999999);

int x = ps.excuteUpdate();

ps = con.prepareStatement(“update customer set cname = ?, email = ?, phone = ? where cid = ?”);

ps.setString(1, “bbbb”);

ps.setString(2, “bbbb@gmail.com”);

ps.setLong(3, 123456789);

ps.setInt(4, 1000);

int x = ps.exeuteUpdate();

ps = con.prepareStatement(“delete from customer where cid = ?”);

ps.setInt(1, 1000);

int x = ps.executeUpdate();

Example:

**package** com.jdbc;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**publicclass** Example2 {

**publicstaticvoid** main(String[] args) {

Connection con = **null**;

ResultSet rs = **null**;

con = *getConnection*();

System.*out*.println(con);

// insert record

PreparedStatement ps = **null**;

**try** {

ps = con.prepareStatement("insert into usr values(?,?,?)");

ps.setInt(1, 1004);

ps.setString(2, "username4");

ps.setString(3, "password4");

**int** i = ps.executeUpdate();

**if**(i>0){

System.*out*.println("record inserted.");

} **else**{

System.*out*.println("Error in record insertion.......");

}

// update record

ps = con.prepareStatement("update usr set password = 'password@1' where username = 'username1'");

**int** j = ps.executeUpdate();

**if**(j > 0){

System.*out*.println("record updated.");

}**else**{

System.*out*.println("Error while record update........");

}

//fetch the record

ps = con.prepareStatement("select \* from usr");

rs = ps.executeQuery();

**while**(rs.next()){

System.*out*.println("userId = " + rs.getInt(1) + " username = " + rs.getString(2) + " password = " + rs.getString(3));

}

//delete the record

ps = con.prepareStatement("delete from usr where username <> ?");

ps.setString(1,"username1");

ps.execute();

ps = con.prepareStatement("select \* from usr");

rs = ps.executeQuery();

**while**(rs.next()){

System.*out*.println("userId = " + rs.getInt(1) + " username = " + rs.getString(2) + " password = " + rs.getString(3));

}

} **catch** (SQLException e) {

e.printStackTrace();

}**finally**{

*close*(**rs**, ps, con);

}

}

**privatestatic** Connection getConnection(){

Connection con = **null**;

**try** {

Class.*forName*("oracle.jdbc.driver.OracleDriver");// ojd OD

**try** {

con = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe","example","example"); // jot ips

} **catch** (SQLException e) {

e.printStackTrace();

}

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

**return** con;

}

**privatestaticvoid** close(ResultSet rs,Statement st, Connection con){

**if**(rs != **null**){

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(st != **null**){

**try** {

st.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(con != **null**){

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

1. **CallableStatement:**

* CallableStatementis an interface available in “java.sql” package and is extending “PreparedStatement” interface.
* It is mainly designed to invoke stored procedures running in the database.
* If you want to write business logic inside database, you can choose stored procedures. When stored procedure is created, it will be compiled and stored in database, so that when you invoke the procedure, it will be executed directly without completion.
* You can create CallableStatement object by using one of the following methods of Connection interface.

1. CallableStatement prepareCall(String)
2. CallableStatement prepareCall(String, int, int)
3. CallableStatement prepareCall(String, int, int, int)

* Once CallableStatement object is created, you can call following methods:

1. int executeUpdate()
2. ResultSet executeQuery()
3. boolean execute()

* With one CallableStatement object you can make a call to one stored procedure only.

Example:

**package** com.jdbc;

**import** java.sql.CallableStatement;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** java.sql.Types;

**publicclass** Example3 {

**publicstaticvoid** main(String[] args) {

Connection con = **null**;

CallableStatement cs = **null**;

ResultSet rs = **null**;

**try** {

con = *getConnection*();

cs = con.prepareCall("{call doMaths(?,?,?,?)}");

cs.setInt(1, 45);

//cs.setInt("num2", 50); //java.sql.SQLException: operation not allowed: Ordinal binding and Named binding cannot be combined!

cs.setInt(2, 50);

cs.setString(3, "add");

cs.registerOutParameter(4, Types.*INTEGER*);

cs.executeUpdate();

System.*out*.println("result = " + cs.getInt(4));

} **catch** (SQLException e) {

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **finally**{

*close*(rs,cs,con);

}

}

**privatestatic** Connection getConnection() **throws** SQLException,

ClassNotFoundException {

Connection con = **null**;

Class.*forName*("oracle.jdbc.driver.OracleDriver"); // ojd OD

con = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe", "example", "example"); // jot ips

**return** con;

}

**privatestaticvoid** close(ResultSet rs, Statement st, Connection con) {

**if** (rs != **null**) {

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (st != **null**) {

**try** {

st.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (con != **null**) {

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

-- example with in

**createor**replace**procedure** doMaths(num1 **in** number, num2 **in** number, operation **in** varchar2)

**as**

res number(10);

**begin**

if operation = 'add'**then**

res:= num1+num2;

**end** if;

dbms\_output.put\_line('result =' || res);

**end**;

/

-- to execute the above procedure

call doMaths(7,3,'add');

-----------------------------------

-- Example with OUT

**createor**replace**procedure** doMaths(num1 **in** number, num2 **in** number, operation **in** varchar2, res out number)

**as**

--res number(10);

**begin**

if operation = 'add'**then**

res:= num1+num2;

**end** if;

dbms\_output.put\_line('result =' || res);

**end**;

/

**declare**

res number;

**begin**

doMaths(8,3,'add',res);

dbms\_output.put\_line('result =' || res);

**end**;

/

**Question:** How to retrieve INOUT parameter in the java program?

**Answer:**

String sql = "{ call abc(?, ?, ?) }";

CallableStatement cs = conn.prepareCall(sql);

cs.setInt(1, 20); // setting "a" in parameter to 1

cs.registerOutParameter(2, Types.VARCHAR); // setting "b" as out parameter

cs.setString(3, "Some String"); // setting "c" as in parameter

cs.registerOutParameter(3, Types.VARCHAR); // setting "c" as out parameter

// then execute

cs.executeUpdate();

// and retrieve out parameters

String bout = cs.getString(2);

String cout = cs.getString(3);

The following are the syntax (with and without return value, respectively):

{? = call <procedure-name> [(arg1, arg2, ...)]}

{call <procedure-name> [(arg1, arg2, ...)]}

**Batch Updates**

* JDBC batch update concept will help you to submit bulk of insert or update or delete Statements once as a batch.
* When you are submitting insert or update or delete Statements one by one to database, it consumes a lot of time.

Normal – 1Q = 5(request)+ 5(compile)+ 5(run)+ 5(response) = 20 ms

1000 Q = 20,000 ms

Batch Update: 5(request) + 5000(compile) + 5000(run) + 5(response) = 10,010 ms

* Because of using batch update, we can reduce network traffic between java application & database server. The application performance will increase.

You can batch both SQL inserts, updates and deletes. It does not make sense to batch select statements.  
There are two ways to execute batch updates:

1. Using a Statement
2. Using a PreparedStatement

Statement Batch Updates

You can use a Statement object to execute batch updates. You do so using the addBatch() andexecuteBatch() methods.

Statement statement = **null**;

**try**{

statement = connection.createStatement();

statement.addBatch("update people set firstname='John' where id=123");

statement.addBatch("update people set firstname='Eric' where id=456");

statement.addBatch("update people set firstname='May' where id=789");

**int**[] recordsAffected = statement.executeBatch();

} **finally** {

**if**(statement != **null**) statement.close();

}

PreparedStatement Batch Updates

The PreparedStatementenables you to reuse the same SQL statement, and just insert new parameters into it, for each update to execute.

String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement = **null**;

**try**{

preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");

preparedStatement.setString(2, "Larson");

preparedStatement.setLong (3, 123);

preparedStatement.addBatch();

preparedStatement.setString(1, "Stan");

preparedStatement.setString(2, "Lee");

preparedStatement.setLong (3, 456);

preparedStatement.addBatch();

**int**[] affectedRecords = preparedStatement.executeBatch();

}**finally** {

**if**(preparedStatement != **null**) {

preparedStatement.close();

}

}

executeBatch()

Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts. The int elements of the array that is returned are ordered to correspond to the commands in the batch, which are ordered according to the order in which they were added to the batch. The elements in the array returned by the method executeBatch may be one of the following:

1. A number greater than or equal to zero -- indicates that the command was processed successfully and is an update count giving the number of rows in the database that were affected by the command's execution
2. A value of SUCCESS\_NO\_INFO -- indicates that the command was processed successfully but that the number of rows affected is unknown

If one of the commands in a batch update fails to execute properly, this method throws a BatchUpdateException, and a JDBC driver may or may not continue to process the remaining commands in the batch. However, the driver's behavior must be consistent with a particular DBMS, either always continuing to process commands or never continuing to process commands. If the driver continues processing after a failure, the array returned by the method BatchUpdateException.getUpdateCounts will contain as many elements as there are commands in the batch, and at least one of the elements will be the following:

1. A value of EXECUTE\_FAILED -- indicates that the command failed to execute successfully and occurs only if a driver continues to process commands after a command fails

The possible implementations and return values have been modified in the Java 2 SDK, Standard Edition, version 1.3 to accommodate the option of continuing to proccess commands in a batch update after a BatchUpdateException obejct has been thrown.

**Returns:**

An array of update counts containing one element for each command in the batch. The elements of the array are ordered according to the order in which commands were added to the batch.

Example 4 :

**package** com.jdbc;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**publicclass** Example4 {

**publicstaticvoid** main(String[] args) {

Connection con = **null**;

Statement st = **null**;

ResultSet rs = **null**;

**try** {

con = JdbcUtil.*getConnection*();

st = con.createStatement();

st.addBatch("insert into usr values(1002,'username2','password2')");

st.addBatch("insert into usr values(1003,'username3','password3')");

st.addBatch("insert into usr values(1004,'username4','password4')");

st.addBatch("update usr set password = 'password@1' where username = 'username1'");

st.addBatch("delete from usr where username <> 'username1'");

**int**[] countArr = st.executeBatch();

**for** (**int** x : countArr){

System.*out*.println(x);

}

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}**finally**{

JdbcUtil.*clean*(rs,ps,con);

}

}

}

**class** JdbcUtil{

**publicstatic** Connection getConnection() **throws** ClassNotFoundException, SQLException{

Connection con = **null**;

Class.*forName*("oracle.jdbc.driver.OracleDriver"); // ojd OD

con = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe", "example", "example"); //jot ips

**return** con;

}

**publicstaticvoid** clean(ResultSet rs, Statement st, Connection con) {

**if**(rs != **null**){

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(st != **null**){

**try** {

st.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(con != **null**){

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

Output:

1

1

1

1

3

**Example5:**

**package** com.jdbc;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**publicclass** Example5 {

**publicstaticvoid** main(String[] args) {

Connection con = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

**try** {

con = JdbcUtil.*getConnection*();

ps = con.prepareStatement("insert into usr values(?,?,?)");

ps.setInt(1, 1002);

ps.setString(2, "username2");

ps.setString(3, "password2");

ps.addBatch();

//ps = con.prepareStatement("insert into usr values(?,?,?)");

ps.setInt(1, 1003);

ps.setString(2, "username3");

ps.setString(3, "password3");

ps.addBatch();

//ps = con.prepareStatement("insert into usr values(?,?,?)");

ps.setInt(1, 1004);

ps.setString(2, "username4");

ps.setString(3, "password4");

ps.addBatch();

// PreparedStatement can add only one type of sql query into batch with multiple parameters set.

// ps = con.prepareStatement("update usr set password = ? where username = ?");

// ps.setString(1, "password@1");

// ps.setString(2, "username1");

// ps.addBatch();

//

// ps = con.prepareStatement("delete from usr where username<> ?");

// ps.setString(1, "username1");

// ps.addBatch();

**int**[] countArr = ps.executeBatch();

**for** (**int** x : countArr){

System.*out*.println(x);

}

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

} **finally**{

JdbcUtil.*clean*(rs,ps,con);

}

}

}

**class** JdbcUtil{

**publicstatic** Connection getConnection() **throws** ClassNotFoundException, SQLException{

Connection con = **null**;

Class.*forName*("oracle.jdbc.driver.OracleDriver"); // ojd OD

con = DriverManager.*getConnection*("jdbc:oracle:thin:@localhost:1521:xe", "example", "example"); //jot ips

**return** con;

}

**publicstaticvoid** clean(ResultSet rs, PreparedStatement ps, Connection con) {

**if**(rs != **null**){

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(ps != **null**){

**try** {

ps.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if**(con != **null**){

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

**Question:** I have both mysql and oracle driver, which database connection will be established when I’m trying to get connection as below:

Class.forName(“oracle.jdbc.driver.OracleDriver”);

Class.forName(“mysql.Driver”);

con = DriverManager.getConnection(…………);

**Answer:** The Connection you are going get will depend on url, username, password which you pass to getConnection(…….).

**Question:** What will happen if I try to get connection without loading Driver class?

**Answer:** Exception will be thrown at runtime.

java.sql.SQLException: No suitable driver

Question: Difference between Statement, PreparedStatement, CallableStatement:

|  |  |  |
| --- | --- | --- |
| **Statement** | **PreparedStatement** | **CallableStatement** |
| It is suitable to use  Statement only when we know that we will not need to execute the SQL query multiple times. | This interface is useful to use the same **SQL command multiple times,**like inserting number of records. This gives a **better performance**. | This interface is used for executing the SQL stored procedures.   It adds a level of abstraction, so the execution of stored procedures does not have to be DBMS-specific. |
| the Statement doesn’t offer support for the parameterized SQL queries. Parameterized SQL queries is an important protection from SQL injection attacks. | **PreparedStatement pst = con.prepareStatement(“insert into Employee (empid, empname, empsal) values (?, ?, ?)”);**  The **questions marks** in the above statement are called as **parameters**. The values are represented as question marks as their values are not known at compile time but known as runtime only. | the output parameters need to be explicitly defined through the corresponding registerOutParameter() methods; whereas the input parameters are provided in the same manner as with the PreparedStatement. |
| Statement would be suitable for the execution of the DDL (Data Definition Language) statements, such as CREATE, ALTER, DROP. | No SQL injection attacks problem |  |

**Connection Pool**

1. To get the connection with the database you need to use DriverManager class as follows:

Connection con = DriverManager.getConnection(url, username, password);

1. Connection which you are getting with the DriverManager class is called as DriverManager connection.

**Following are the drawbacks with DriverManager Connection**

1. You need to hard code url, username and password in various program whenever you are getting connection using DriverManager.
2. If you migrate the database later you need to modify all the programs. This gives lot maintenance.
3. When we call getConnection() method a new connection will be established and when you call close() method on the connection then connection will be destroyed permanently.
4. Creating and destroying the connection is expensive and hit the application performance.

To solve above set of problems:

1. You can use data source connections.
2. Data source connections are pooled connections and will run inside a server.

Example of some server: Weblogic, JBoss , Websphere etc

Configure connection pool:

1. Pool name
2. Data base name
3. Driver type
4. url
5. user name
6. password
7. driver class
8. minimum number of connection
9. maximum number of connection

Configure Data Source:

1. data source name
2. pool name
3. data source JNDI name

**Jdbc program**

//set the properties related to your application server.

Context ctx = new InitialContex();

DataSource ds = (DataSource)ctx.lookup(“oracleJNDI”);

Connection con = ds.getConnection();

con.close();

* Enterprise application contains two modules.

1. Web module
2. EJB module

* To run web module, we need web container.
* To run EJB module, we need EJB container.
* Web server contains only web container.
* Application server contains both web container and EJB container.
* Connection pooling mechanism increases scalability of your application by providing the facility to reuse the data base connection.
* When we configure data source in application server following tasks will happen:

1. Pool of database connection will be created.
2. Data source object will be created.
3. Pool will be bound with data source object.
4. Data source object will be bound with the JNDI registry with the given JNDI name.

* The server can increase or decrease the pool size in the peak and non-peak hours.
* When it is increased in size means more users are using the application. We can increase up to the maximum limit.
* When it is decreased in size means less users are using the application. We can decrease up to minimum limit.

**JDBC Transaction Management**

**Question:** What is transaction?

**Answer:** Transaction is a process of executing set of data base operations with all or nothing criteria.

i.e. when all the operations are successful in a unit of all operation than transaction should be ended by calling the commit();

When any one operation is failed in that unit of all operation then transaction should be ended by calling the rollback();

**Question:** What does a transaction guarantee?

**Answer:** If you implement the transaction in your application properly it will guarantee ACID properties.

A =>Atomicity

C => Consistency

I => Isolation

D => Durability

**Question:** How to start and end the transaction in JDBC?

**Answer:**

try{

con.autoCommit(false); //beginning of the transaction

op1;

op2;

op3;

op4;

con.commit();//end of transaction after successfully execution of all operation.

} catch(){

con.rollback();//end of transaction after failure execution of anyone operation.

}

op5;

op6;

When transaction is ended by calling the commit() and rollback() auto commit will be enable immediately.

**Example of Jdbc transaction:**

Fund Transfer scenario:

Steps,

1. Set auto commit false
2. Get the balance of FromAcc
3. Deduct the txAmnt for FromAcc or return exception
4. Update the FromAcc
5. Add txAmnt to ToAcc
6. Update the ToAcc
7. Commit all changes

**createtable** account\_detail(

account\_id number **primarykey**,

account\_type varchar2(2) **notnull**,

cust\_name varchar2(20),

balance number

);

**insertinto** account\_detail **values**(10001,'SA','customer1',50000);

**insertinto** account\_detail **values**(10002,'SA','customer2',30000);

**Example 6:**

**package** com.jdbc;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

/\*\*

\* **@author**surajkumar

\*

\* This class shows the example of Jdbc transaction(Fund transfer scenario)

\*

\*/

**publicclass** Example6 {

**publicstaticvoid** main(String[] args) {

**try** {

*fundTransfer*(10001,10002,5000);

System.*out*.println("Fund Transferred successfully!");

} **catch** (ClassNotFoundException e) {

System.*out*.println(e.getMessage());

} **catch** (SQLException e) {

System.*out*.println(e.getMessage());

} **catch** (AccountException e) {

System.*out*.println(e.getMessage());

}

}

**publicstaticvoid** fundTransfer(**int** fromAcc, **int** toAcc, **int** amount) **throws** AccountException, ClassNotFoundException, SQLException {

Connection con = **null**;

PreparedStatement ps = **null**;

PreparedStatement ps2 = **null**;

PreparedStatement ps3 = **null**;

ResultSet rs = **null**;

**try** {

con = JdbcUtil.*getConnection*();

// set auto commit as false

con.setAutoCommit(**false**);

// get FromAcc balance

ps = con.prepareStatement("select balance from account\_detail where account\_id = ?");

ps.setInt(1,fromAcc);

rs = ps.executeQuery();

**int** fromBalance = 0;

**while**(rs.next()){

fromBalance = rs.getInt("balance");

}

// verify the sufficient balance

**if**(fromBalance < amount || fromBalance - amount < 5000){

**thrownew** AccountException("Insufficient Balance in the account...." + fromAcc);

}

// get toAcc balance

ps2 = con.prepareStatement("select balance from account\_detail where account\_id = ?");

ps2.setInt(1, toAcc);

rs = ps2.executeQuery();

**int** toBalance = 0;

**while**(rs.next()){

toBalance = rs.getInt("balance");

}

// transfer fund

fromBalance = fromBalance - amount;

toBalance = toBalance + amount;

// update the database

ps3 = con.prepareStatement("update account\_detail set balance = ? where account\_id = ?");

ps3.setInt(1, fromBalance);

ps3.setInt(2, fromAcc);

ps3.addBatch();

ps3.setInt(1, toBalance);

ps3.setInt(2, toAcc);

ps3.addBatch();

ps3.executeBatch();

// commit all changes to database

con.commit();

} **catch** (SQLException sqle){

System.*out*.println("inside catch block");

con.rollback();

**throw** sqle;

} **finally**{

JdbcUtil.*clean*(rs, ps, con);

JdbcUtil.*clean*(rs, ps2, con);

JdbcUtil.*clean*(rs, ps3, con);

}

}

}

**class** AccountException **extends** Exception{

/\*\*

\*

\*/

**privatestaticfinallong***serialVersionUID* = 3694697889749689603L;

**public** AccountException(){}

**public** AccountException(String message){

**super**(message);

}

}

**Note:**  To produce SQLException make one line change as below

ps = con.prepareStatement("selecta balance from account\_detail where account\_id = ?");

Output will be:

inside catch block

Invalid SQL type: sqlKind = UNINITIALIZED

**ResultSet**

1. ResultSet is an interface which is available in “java.sql” package.
2. ResultSet’s objects are used to store the set of records which are fetched from the database table using “select” statement.
3. When a ResultSet object is created newly, ResultSet cursor will point to before first record.

Before to first record

|  |
| --- |
| 1. |
| 2. |
| 3. |
|  |
|  |

First record

Last record  
 after last record

**rs.next()**

If next record found then return true and move the cursor to next record. Otherwise return false and make the cursor to “after last”.

**rs.previous()**

If previous record found then return true and moves the cursor to previous record. Otherwise return false and make the cursor to “before to first”.

**Types of ResultSet:**

Depending on the scrollability of the ResultSet you can divided the ResultSet into two types:

1. Forward only ResultSet
2. Scrollable ResultSet
3. **Forward only ResultSet:**
4. When ResultSet is forward only, we can move the cursor toward forward direction only.
5. When ResultSet is forward only, we can call only the following methods on the ResultSet object.
6. next()
7. close()
8. getXXXXX()

3. previous is not allowed.

4. By default ResultSet are forward only.

5. ResultSet type creation is depending on the statement which you are using.

st = con.createStatement();

rs = st.executeQuery(sql);

Here rs type will be forward only and read only.

st = con.createStatement(i, j, k ); // createStatement(int, int, int);

rs = st.executeQuery(sql);

Here rs type will depend on i, j, k.

i => TYPE\_FORWARDONLY

TYPE\_SCROLL\_SENSITIVE

TYPE\_SCROLL\_INSENSITIVE

j => CONCUR\_READ\_ONLY

CONCUR\_UPDATABLE

k => HOLD\_CURSORS\_OVER\_COMMIT

CLOSE\_CURSORS\_AT\_COMMIT

rs.getInt(int) // int parameter will start from 0

rs.getInt(string) // string parameter will be the column name

**Returns:**

the column value; if the value is SQL NULL, the value returned is 0

rs.getString(int)

rs.getString(String)

**Returns:**

the column value; if the value is SQL NULL, the value returned is null

1. **Scrollable ResultSet:**
2. When ResultSet is scrollable then we can move the cursor in forward direction and also in reverse direction.
3. When ResultSet is scrollable then we can call the following methods on the ResultSet object:
4. next()
5. previous()
6. close()
7. getXXXXX()
8. absolute()
9. relative()
10. first()
11. last()
12. isFirst()
13. isLast()
14. afterLast()
15. isAfterLast()
16. beforeFirst()
17. isBeforeFirst()
18. Following is the way to create scrollable ResultSet.

st = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_READ\_ONLY);

rs = st.executeQuery(sql);

1. Example of use of scrollable:

rs.relative(-3); // just go to 3 records previous from the present record.

rs.relative(3); // just go to 3 records next to the present record.

rs.absolute(7);//go directly to 7th record.

**Example using scrollable ResultSet**

We can also divide the ResultSet into following types:

1. Read only ResultSet (Static ResultSet)
2. Updatable ResultSet (Dynamic ResultSet)
3. **Read only ResultSet:**
4. When ResultSet is Read only you can read the data from ResultSet by calling “getXXXX” methods, but you cannot update the data.
5. By default the ResultSet is read only.
6. Read only ResultSet is also called as static ResultSet.
7. **Updatable ResultSet:**
8. When ResultSet is updatable you can do the following operations on the ResultSet.

* You can read the data.
* You can insert the new record in the ResultSet.
* You can delete the records from the ResultSet.
* You can delete the records from the ResultSet.
* You can update the existing ResultSet record.

1. When we do update, delete, insert in the record the same operation will be done in the underlying table immediately.
2. Following are the methods which you can invoke only on updatable ResultSet.

updateXXXXX();

insertRow();

deleteRow();

updateRow();

refreshRow();

moveToInsertRow();

moveToCurrentRow();

rowDeleted();

rowInserted();

rowUpdated();

1. Following is the way to create Updatable ResultSet:

st = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

rs = st.executeQuery(sql);

Here rs is scrollable and updatable.

Example of Updatable ResultSet.

**RowSet**

1. RowSet is an interface which is available in “javax.sql” package and is extending “java.sql.ResultSet”.
2. Like ResultSet RowSet is also used to store set of records return from the db.
3. Problems with the ResultSet:

|  |  |  |
| --- | --- | --- |
|  | ResultSet Usage | RowSet Usage |
|  | Load the driver class  con = ….;  st = con.createStatement();  rs = st.executeQuery(sql);  while(rs.next()){  } | CachedRowSet crs = new CachedRowSetImpl();  crs.setUrl(jdbc:mysql:\\localhost\EXDB);  crs.setUsername(“root”);  crs.setPassword(“root”);  crs.setCommand(“select \* from customer”);  crs.execute();  while(crs.next()){  } |
|  | To use ResultSet you need to create three objects connection, statement and ResultSet. | With on RowSet object we can perform the entire database operation. |
|  | ResultSet is connection oriented object i.e. without connection we can’t access ResultSet object. | RowSet object is connectionless, without connection you can access data inside RowSet. |
|  | When you are using ResultSet you are responsible to clean all the resources like con, st, rs. | With RowSet automatic cleanup process will happen. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Statement** | **Create/prepare** | **execute** | **Return after execution** | **operations** |
| Statement | con.createStatement()  con.createStatement(int, int)  con.createStatement(int, int, int)  String sql = “insert into table values(‘fds’,123,443)”; | st.excute(sql) | boolean | Insert, update, Delete |
| st.executeUpdate(sql) | int | Insert, update, Delete |
| st.executeQuery(sql) | ResultSet | Select |
| PreparedStatement | con.prepareStatement(String)  con.prepareStatement(String,int,int)  con.prepareStatement(String,int,int,int)  string = “insert into table values(?,?,?)”;  ps.setString(1,”XXXX”);  ps.setLong(2,23243443);  ps.setInt(3, 4343); | ps.execute() | boolean | Insert, update, Delete |
| ps.executeUpdate() | int | Insert, update, Delete |
| ps.executeQuery() | ResultSet  rs.getInt(1);/rs.getInt(“column\_name”);  rs.getString(2);/rs.getString(“column\_name”); | select |
| CallableStatement | con.prepareCall(String)  con.prepareCall(String, int, int)  con.prepareCall(String, int,int,int)  string =  “{call doMaths(?,?)}”;  cs.setInt(1,23443);//IN parameter  cs.registerOutParameter(2, Types.VARCHAR);//OUT Parameter  cs.setString(2,”xyz”);//IN Parameter  or,  cs.setInt(“param\_name”,123);  cs.registerOutParameter(“param\_name”, Types.VARCHAR);  cs.setString(“param\_name”, “dfsd”); | cs.execute() | boolean | To call stored procedure.  cs.getString(2);  or,  cs.getString(“param\_name”);  if we combine both number and param\_name for place holder value we’ll get runtime exception:  java.sql.SQLException: operation not allowed: Ordinal binding and Named binding cannot be combined! |
| cs.executeUpdate() | int |
| cs.executeQuery() | ResultSet |

ResultSet in Detail:

# Retrieving and Modifying Values from Result Sets

The following method, [CoffeesTable.viewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) outputs the contents of the COFFEES tables, and demonstrates the use of ResultSet objects and cursors:

public static void viewTable(Connection con, String dbName)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL " +

"from " + dbName + ".COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

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

}

}

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. For example, the[CoffeeTables.viewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) method creates a ResultSet, rs, when it executes the query through the Statement object, stmt. Note that a ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method ResultSet.next moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

This page covers the following topics:

* [ResultSet Interface](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_interface)
* [Retrieving Column Values from Rows](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#retrieve_rs)
* [Cursors](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#cursors)
* [Updating Rows in ResultSet Objects](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_update)
* [Using Statement Objects for Batch Updates](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#batch_updates)
* [Inserting Rows in ResultSet Objects](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_insert)

## ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are type, concurrency, and cursor holdability.

### ResultSet Types

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

* TYPE\_FORWARD\_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
* TYPE\_SCROLL\_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
* TYPE\_SCROLL\_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE\_FORWARD\_ONLY.

**Note**: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData.supportsResultSetType returns true if the specified ResultSet type is supported and false otherwise.

### ResultSet Concurrency

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

* CONCUR\_READ\_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
* CONCUR\_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR\_READ\_ONLY.

**Note**: Not all JDBC drivers and databases support concurrency. The method DatabaseMetaData.supportsResultSetConcurrency returns true if the specified concurrency level is supported by the driver and false otherwise.

The method [CoffeesTable.modifyPrices](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) demonstrates how to use a ResultSet object whose concurrency level is CONCUR\_UPDATABLE.

### Cursor Holdability

Calling the method Connection.commit can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior. The ResultSet property holdability gives the application control over whether ResultSet objects (cursors) are closed when commit is called.

The following ResultSet constants may be supplied to the Connection methods createStatement, prepareStatement, and prepareCall:

* HOLD\_CURSORS\_OVER\_COMMIT: ResultSet cursors are not closed; they are holdable: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.
* CLOSE\_CURSORS\_AT\_COMMIT: ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.

The default cursor holdability varies depending on your DBMS.

**Note**: Not all JDBC drivers and databases support holdable and non-holdable cursors. The following method, JDBCTutorialUtilities.cursorHoldabilitySupport, outputs the default cursor holdability of ResultSet objects and whether HOLD\_CURSORS\_OVER\_COMMIT and CLOSE\_CURSORS\_AT\_COMMIT are supported:

public static void cursorHoldabilitySupport(Connection conn)

throws SQLException {

DatabaseMetaData dbMetaData = conn.getMetaData();

System.out.println("ResultSet.HOLD\_CURSORS\_OVER\_COMMIT = " +

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT);

System.out.println("ResultSet.CLOSE\_CURSORS\_AT\_COMMIT = " +

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

System.out.println("Default cursor holdability: " +

**dbMetaData.getResultSetHoldability()**);

System.out.println("Supports HOLD\_CURSORS\_OVER\_COMMIT? " +

**dbMetaData.supportsResultSetHoldability(**

**ResultSet.HOLD\_CURSORS\_OVER\_COMMIT)**);

System.out.println("Supports CLOSE\_CURSORS\_AT\_COMMIT? " +

**dbMetaData.supportsResultSetHoldability(**

**ResultSet.CLOSE\_CURSORS\_AT\_COMMIT)**);

}

## Retrieving Column Values from Rows

The ResultSet interface declares getter methods (for example, getBoolean and getLong) for retrieving column values from the current row. You can retrieve values using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For example, the following method, [CoffeesTable.alternateViewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), retrieves column values by number:

public static void alternateViewTable(Connection con)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL from COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString(1);

int supplierID = rs.getInt(2);

float price = rs.getFloat(3);

int sales = rs.getInt(4);

int total = rs.getInt(5);

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

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

}

}

Strings used as input to getter methods are case-insensitive. When a getter method is called with a string and more than one column has the same alias or name as the string, the value of the first matching column is returned. The option to use a string as opposed to an integer is designed to be used when column aliases and names are used in the SQL query that generated the result set. For columns that are not explicitly named in the query (for example, select \* from COFFEES) it is best to use column numbers. If column names are used, the developer should guarantee that they uniquely refer to the intended columns by using column aliases. A column alias effectively renames the column of a result set. To specify a column alias, use the SQL ASclause in the SELECT statement.

The getter method of the appropriate type retrieves the value in each column. For example, in the method [CoffeeTables.viewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), the first column in each row of the ResultSet rs isCOF\_NAME, which stores a value of SQL type VARCHAR. The method for retrieving a value of SQL type VARCHAR is getString. The second column in each row stores a value of SQL typeINTEGER, and the method for retrieving values of that type is getInt.

Note that although the method getString is recommended for retrieving the SQL types CHAR and VARCHAR, it is possible to retrieve any of the basic SQL types with it. Getting all values withgetString can be very useful, but it also has its limitations. For instance, if it is used to retrieve a numeric type, getString converts the numeric value to a Java String object, and the value has to be converted back to a numeric type before it can be operated on as a number. In cases where the value is treated as a string anyway, there is no drawback. Furthermore, if you want an application to retrieve values of any standard SQL type other than SQL3 types, use the getString method.

## Cursors

As mentioned previously, you access the data in a ResultSet object through a cursor, which points to one row in the ResultSet object. However, when a ResultSet object is first created, the cursor is positioned before the first row. The method [CoffeeTables.viewTable](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) moves the cursor by calling the ResultSet.next method. There are other methods available to move the cursor:

* next: Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.
* previous: Moves the cursor backward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.
* first: Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.
* last:: Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.
* beforeFirst: Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.
* afterLast: Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.
* relative(int rows): Moves the cursor relative to its current position.
* absolute(int row): Positions the cursor on the row specified by the parameter row.

Note that the default sensitivity of a ResultSet is TYPE\_FORWARD\_ONLY, which means that it cannot be scrolled; you cannot call any of these methods that move the cursor, except next, if your ResultSet cannot be scrolled. The method [CoffeesTable.modifyPrices](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), described in the following section, demonstrates how you can move the cursor of a ResultSet.

## Updating Rows in ResultSet Objects

You cannot update a default ResultSet object, and you can only move its cursor forward. However, you can create ResultSet objects that can be scrolled (the cursor can move backwards or move to an absolute position) and updated.

The following method, [CoffeesTable.modifyPrices](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), multiplies the PRICE column of each row by the argument percentage:

public void modifyPrices(float percentage) throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement();

stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName + ".COFFEES");

while (uprs.next()) {

float f = uprs.getFloat("PRICE");

uprs.updateFloat( "PRICE", f \* percentage);

uprs.updateRow();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

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

}

}

The field ResultSet.TYPE\_SCROLL\_SENSITIVE creates a ResultSet object whose cursor can move both forward and backward relative to the current position and to an absolute position. The field ResultSet.CONCUR\_UPDATABLE creates a ResultSet object that can be updated. See the ResultSet Javadoc for other fields you can specify to modify the behavior ofResultSet objects.

The method ResultSet.updateFloat updates the specified column (in this example, PRICE with the specified float value in the row where the cursor is positioned. ResultSet contains various updater methods that enable you to update column values of various data types. However, none of these updater methods modifies the database; you must call the methodResultSet.updateRow to update the database.

## Using Statement Objects for Batch Updates

Statement, PreparedStatement and CallableStatement objects have a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as CREATE TABLE and DROP TABLE. It cannot, however, contain a statement that would produce a ResultSet object, such as a SELECT statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a Statement object at its creation, is initially empty. You can add SQL commands to this list with the method addBatch and empty it with the methodclearBatch. When you have finished adding statements to the list, call the method executeBatch to send them all to the database to be executed as a unit, or batch.

For example, the following method [CoffeesTable.batchUpdate](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) adds four rows to the COFFEES table with a batch update:

public void batchUpdate() throws SQLException {

Statement stmt = null;

try {

this.con.setAutoCommit(false);

stmt = this.con.createStatement();

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto\_decaf', 49, " +

"10.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut\_decaf', 49, " +

"10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();

this.con.commit();

} catch(BatchUpdateException b) {

JDBCTutorialUtilities.printBatchUpdateException(b);

} catch(SQLException ex) {

JDBCTutorialUtilities.printSQLException(ex);

} finally {

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

this.con.setAutoCommit(true);

}

}

The following line disables auto-commit mode for the Connection object con so that the transaction will not be automatically committed or rolled back when the method executeBatch is called.

this.con.setAutoCommit(false);

To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.

The method Statement.addBatch adds a command to the list of commands associated with the Statement object stmt. In this example, these commands are all INSERT INTOstatements, each one adding a row consisting of five column values. The values for the columns COF\_NAME and PRICE are the name of the coffee and its price, respectively. The second value in each row is 49 because that is the identification number for the supplier, Superior Coffee. The last two values, the entries for the columns SALES and TOTAL, all start out being zero because there have been no sales yet. (SALES is the number of pounds of this row's coffee sold in the current week; TOTAL is the total of all the cumulative sales of this coffee.)

The following line sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

int [] updateCounts = stmt.executeBatch();

Note that stmt uses the method executeBatch to send the batch of insertions, not the method executeUpdate, which sends only one command and returns a single update count. The DBMS executes the commands in the order in which they were added to the list of commands, so it will first add the row of values for Amaretto, then add the row for Hazelnut, then Amaretto decaf, and finally Hazelnut decaf. If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts that indicate how many rows were affected by each command are stored in the array updateCounts.

If all four of the commands in the batch are executed successfully, updateCounts will contain four values, all of which are 1 because an insertion affects one row. The list of commands associated with stmt will now be empty because the four commands added previously were sent to the database when stmt called the method executeBatch. You can at any time explicitly empty this list of commands with the method clearBatch.

The Connection.commit method makes the batch of updates to the COFFEES table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.

The following line enables auto-commit mode for the current Connection object.

this.con.setAutoCommit(true);

Now each statement in the example will automatically be committed after it is executed, and it no longer needs to invoke the method commit.

### Performing Parameterized Batch Update

It is also possible to have a parameterized batch update, as shown in the following code fragment, where con is a Connection object:

con.setAutoCommit(false);

PreparedStatement pstmt = con.prepareStatement(

"INSERT INTO COFFEES VALUES( " +

"?, ?, ?, ?, ?)");

pstmt.setString(1, "Amaretto");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

pstmt.setString(1, "Hazelnut");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

**// ... and so on for each new**

**// type of coffee**

int [] updateCounts = pstmt.executeBatch();

con.commit();

con.setAutoCommit(true);

### Handling Batch Update Exceptions

You will get a BatchUpdateException when you call the method executeBatch if (1) one of the SQL statements you added to the batch produces a result set (usually a query) or (2) one of the SQL statements in the batch does not execute successfully for some other reason.

You should not add a query (a SELECT statement) to a batch of SQL commands because the method executeBatch, which returns an array of update counts, expects an update count from each SQL statement that executes successfully. This means that only commands that return an update count (commands such as INSERT INTO, UPDATE, DELETE) or that return 0 (such asCREATE TABLE, DROP TABLE, ALTER TABLE) can be successfully executed as a batch with the executeBatch method.

A BatchUpdateException contains an array of update counts that is similar to the array returned by the method executeBatch. In both cases, the update counts are in the same order as the commands that produced them. This tells you how many commands in the batch executed successfully and which ones they are. For example, if five commands executed successfully, the array will contain five numbers: the first one being the update count for the first command, the second one being the update count for the second command, and so on.

BatchUpdateException is derived from SQLException. This means that you can use all of the methods available to an SQLException object with it. The following method,[JDBCTutorialUtilities.printBatchUpdateException](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html) prints all of the SQLException information plus the update counts contained in a BatchUpdateException object. Because BatchUpdateException.getUpdateCounts returns an array of int, the code uses a for loop to print each of the update counts:

public static void printBatchUpdateException(BatchUpdateException b) {

System.err.println("----BatchUpdateException----");

System.err.println("SQLState: " + b.getSQLState());

System.err.println("Message: " + b.getMessage());

System.err.println("Vendor: " + b.getErrorCode());

System.err.print("Update counts: ");

int [] updateCounts = b.getUpdateCounts();

for (int i = 0; i < updateCounts.length; i++) {

System.err.print(updateCounts[i] + " ");

}

}

## Inserting Rows in ResultSet Objects

**Note**: Not all JDBC drivers support inserting new rows with the ResultSet interface. If you attempt to insert a new row and your JDBC driver database does not support this feature, aSQLFeatureNotSupportedException exception is thrown.

The following method, [CoffeesTable.insertRow](https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html), inserts a row into the COFFEES through a ResultSet object:

public void insertRow(String coffeeName, int supplierID,

float price, int sales, int total)

throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_SENSITIVE

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName +

".COFFEES");

uprs.moveToInsertRow();

uprs.updateString("COF\_NAME", coffeeName);

uprs.updateInt("SUP\_ID", supplierID);

uprs.updateFloat("PRICE", price);

uprs.updateInt("SALES", sales);

uprs.updateInt("TOTAL", total);

uprs.insertRow();

uprs.beforeFirst();

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

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

}

}

This example calls the Connection.createStatement method with two arguments, ResultSet.TYPE\_SCROLL\_SENSITIVE and ResultSet.CONCUR\_UPDATABLE. The first value enables the cursor of the ResultSet object to be moved both forward and backward. The second value, ResultSet.CONCUR\_UPDATABLE, is required if you want to insert rows into aResultSet object; it specifies that it can be updatable.

The same stipulations for using strings in getter methods also apply to updater methods.

The method ResultSet.moveToInsertRow moves the cursor to the insert row. The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row can be constructed by calling the updater methods prior to inserting the row into the result set. For example, this method calls the method ResultSet.updateString to update the insert row's COF\_NAME column to Kona.

The method ResultSet.insertRow inserts the contents of the insert row into the ResultSet object and into the database.

**Note**: After inserting a row with the ResultSet.insertRow, you should move the cursor to a row other than the insert row. For example, this example moves it to before the first row in the result set with the method ResultSet.beforeFirst. Unexpected results can occur if another part of your application uses the same result set and the cursor is still pointing to the insert row.

**NamedParameterStatement**

package com.crimsonlogic.datacleansing;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.sql.Timestamp;

import java.util.HashMap;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Map.Entry;

/\*\*

\* This class wraps around a {@link PreparedStatement} and allows the programmer

\* to set parameters by name instead of by index. This eliminates any confusion

\* as to which parameter index represents what. This also means that rearranging

\* the SQL statement or adding a parameter doesn't involve renumbering your

\* indices. Code such as this: Connection con=getConnection(); String

\* query="select \* from my\_table where name=? or address=?"; PreparedStatement

\* p=con.prepareStatement(query); p.setString(1, "bob"); p.setString(2,

\* "123 terrace ct"); ResultSet rs=p.executeQuery();

\*

\* can be replaced with: Connection con=getConnection(); String query="select \*

\* from my\_table where name=:name or address=:address"; NamedParameterStatement

\* p=new NamedParameterStatement(con, query); p.setString("name", "bob");

\* p.setString("address", "123 terrace ct"); ResultSet rs=p.executeQuery();

\*\*

\* @author adam\_crume

\*/

public class NamedParameterStatement {

/\*\* The statement this object is wrapping. \*/

private final PreparedStatement statement;

/\*\* Maps parameter names to arrays of ints which are the parameter indices. \*/

private final Map indexMap;

/\*\*

\* Creates a NamedParameterStatement. Wraps a call to c.

\* {@link Connection#prepareStatement(java.lang.String) prepareStatement}.

\*

\* @param connection

\* the database connection

\* @param query

\* the parameterized query

\* @throws SQLException

\* if the statement could not be created

\*/

public NamedParameterStatement(final Connection connection, final String query) throws SQLException {

indexMap = new HashMap();

String parsedQuery = parse(query, indexMap);

statement = connection.prepareStatement(parsedQuery);

}

/\*\*

\* Parses a query with named parameters. The parameter-index mappings are put

\* into the map, and the parsed query is returned. DO NOT CALL FROM CLIENT

\* CODE. This method is non-private so JUnit code can test it.

\*

\* @param query

\* query to parse

\* @param paramMap

\* map to hold parameter-index mappings

\* @return the parsed query

\*/

static final String parse(final String query, final Map paramMap) {

// I was originally using regular expressions, but they didn't work well for ignoring

// parameter-like strings inside quotes.

int length = query.length();

StringBuffer parsedQuery = new StringBuffer(length);

boolean inSingleQuote = false;

boolean inDoubleQuote = false;

int index = 1;

for (int i = 0; i < length; i++) {

char c = query.charAt(i);

if (inSingleQuote) {

if (c == '\'') {

inSingleQuote = false;

}

} else if (inDoubleQuote) {

if (c == '"') {

inDoubleQuote = false;

}

} else {

if (c == '\'') {

inSingleQuote = true;

} else if (c == '"') {

inDoubleQuote = true;

} else if (c == ':' && i + 1 < length && Character.isJavaIdentifierStart(query.charAt(i + 1))) {

int j = i + 2;

while (j < length && Character.isJavaIdentifierPart(query.charAt(j))) {

j++;

}

String name = query.substring(i + 1, j);

c = '?'; // replace the parameter with a question mark

i += name.length(); // skip past the end if the parameter

List indexList = (List) paramMap.get(name);

if (indexList == null) {

indexList = new LinkedList();

paramMap.put(name, indexList);

}

indexList.add(new Integer(index));

index++;

}

}

parsedQuery.append(c);

}

// replace the lists of Integer objects with arrays of ints

for (Iterator itr = paramMap.entrySet().iterator(); itr.hasNext();) {

Map.Entry entry = (Map.Entry) itr.next();

List list = (List) entry.getValue();

int[] indexes = new int[list.size()];

int i = 0;

for (Iterator itr2 = list.iterator(); itr2.hasNext();) {

Integer x = (Integer) itr2.next();

indexes[i++] = x.intValue();

}

entry.setValue(indexes);

}

return parsedQuery.toString();

}

/\*\*

\* Returns the indexes for a parameter.

\*

\* @param name

\* parameter name

\* @return parameter indexes

\* @throws IllegalArgumentException

\* if the parameter does not exist

\*/

private int[] getIndexes(final String name) {

int[] indexes = (int[]) indexMap.get(name);

return indexes;

}

/\*\*

\* Sets a parameter.

\*

\* @param name

\* parameter name

\* @param value

\* parameter value

\* @throws SQLException

\* if an error occurred

\* @throws IllegalArgumentException

\* if the parameter does not exist

\* @see PreparedStatement#setObject(int, java.lang.Object)

\*/

public void setObject(final String name, final Object value) throws SQLException {

int[] indexes = getIndexes(name);

if (indexes == null) {

return;

}

for (int i = 0; i < indexes.length; i++) {

statement.setObject(indexes[i], value);

}

}

/\*\*

\* Sets a parameter.

\*

\* @param name

\* parameter name

\* @param value

\* parameter value

\* @throws SQLException

\* if an error occurred

\* @throws IllegalArgumentException

\* if the parameter does not exist

\* @see PreparedStatement#setString(int, java.lang.String)

\*/

public void setString(final String name, final String value) throws SQLException {

int[] indexes = getIndexes(name);

if (indexes == null) {

return;

}

for (int i = 0; i < indexes.length; i++) {

statement.setString(indexes[i], value);

}

}

public void setParamMap(final Map<String, Object> paramMap) throws SQLException {

for (Entry<String, Object> entry : paramMap.entrySet()) {

setObject(entry.getKey(), entry.getValue());

}

}

/\*\*

\* Sets a parameter.

\*

\* @param name

\* parameter name

\* @param value

\* parameter value

\* @throws SQLException

\* if an error occurred

\* @throws IllegalArgumentException

\* if the parameter does not exist

\* @see PreparedStatement#setInt(int, int)

\*/

public void setInt(final String name, final int value) throws SQLException {

int[] indexes = getIndexes(name);

if (indexes == null) {

return;

}

for (int i = 0; i < indexes.length; i++) {

statement.setInt(indexes[i], value);

}

}

/\*\*

\* Sets a parameter.

\*

\* @param name

\* parameter name

\* @param value

\* parameter value

\* @throws SQLException

\* if an error occurred

\* @throws IllegalArgumentException

\* if the parameter does not exist

\* @see PreparedStatement#setInt(int, int)

\*/

public void setLong(final String name, final long value) throws SQLException {

int[] indexes = getIndexes(name);

if (indexes == null) {

return;

}

for (int i = 0; i < indexes.length; i++) {

statement.setLong(indexes[i], value);

}

}

/\*\*

\* Sets a parameter.

\*

\* @param name

\* parameter name

\* @param value

\* parameter value

\* @throws SQLException

\* if an error occurred

\* @throws IllegalArgumentException

\* if the parameter does not exist

\* @see PreparedStatement#setTimestamp(int, java.sql.Timestamp)

\*/

public void setTimestamp(final String name, final Timestamp value) throws SQLException {

int[] indexes = getIndexes(name);

if (indexes == null) {

return;

}

for (int i = 0; i < indexes.length; i++) {

statement.setTimestamp(indexes[i], value);

}

}

/\*\*

\* Returns the underlying statement.

\*

\* @return the statement

\*/

public PreparedStatement getStatement() {

return statement;

}

/\*\*

\* Executes the statement.

\*

\* @return true if the first result is a {@link ResultSet}

\* @throws SQLException

\* if an error occurred

\* @see PreparedStatement#execute()

\*/

public boolean execute() throws SQLException {

return statement.execute();

}

/\*\*

\* Executes the statement, which must be a query.

\*

\* @return the query results

\* @throws SQLException

\* if an error occurred

\* @see PreparedStatement#executeQuery()

\*/

public ResultSet executeQuery() throws SQLException {

return statement.executeQuery();

}

/\*\*

\* Executes the statement, which must be an SQL INSERT, UPDATE or DELETE

\* statement; or an SQL statement that returns nothing, such as a DDL

\* statement.

\*

\* @return number of rows affected

\* @throws SQLException

\* if an error occurred

\* @see PreparedStatement#executeUpdate()

\*/

public int executeUpdate() throws SQLException {

return statement.executeUpdate();

}

/\*\*

\* Closes the statement.

\*

\* @throws SQLException

\* if an error occurred

\* @see Statement#close()

\*/

public void close() throws SQLException {

statement.close();

}

/\*\*

\* Adds the current set of parameters as a batch entry.

\*

\* @throws SQLException

\* if something went wrong

\*/

public void addBatch() throws SQLException {

statement.addBatch();

}

/\*\*

\* Executes all of the batched statements.

\*

\* See {@link Statement#executeBatch()} for details.

\*

\* @return update counts for each statement

\* @throws SQLException

\* if something went wrong

\*/

public int[] executeBatch() throws SQLException {

return statement.executeBatch();

}

}