IMPORTANT JDBC CORE API CLASSES, INTERFACES AND EXCEPTIONS

The JDBC API is comprised of two Java packages: java.sql and javax.sql. The following are core JDBC classes, interfaces, and exceptions in the java.sql package:

DriverManager

The DriverManager class (java.sql.DriverManager) is one of main components of JDBC. DriverManager manages database drivers, load database specific drivers and select the most appropriate database specific driver from the previously loaded drivers when a new connection is established. Managing, loading and selecting drivers are done automatically by the DriverManager from JDBC 4.0, when a new connection is created.

The DriverManager can also be considered as a connection factory class as it uses database specific drivers to create connection (java.sql.Connection) objects. DriverManager consist of only one private constructor and hence it cannot be inherited or initialized directly. All other members of DriverManager are static. DriverManager maintains a list of DriverInfo objects that hold one Driver object each.

Connection

Connection interface provides a standard abstraction to access the session established with a database server. JDBC driver provider should implement the connection interface. We can obtain a Connection object using the getConnection() method of the DriverManager as:

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

Before JDBC 4.0, we had to first load the Driver implementation before getting the connection from the DriverManager as:

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

We can also use DataSource for creating a connection for better data source portability and is the preferred way in production applications.

Statement

The Statement interface provides a standard abstraction to execute SQL statements and return the results using the ResultSet objects.

A Statement object contains a single ResultSet object at a time. The execute method of a Statement implicitly close its current ResultSet if it is already open. PreparedStatement and CallableStatement are

two specialized Statement interfaces. You can create a Statement object by using the createStatement() method of the Connection interface as:

```
Statement st = con.createStatement();
```

You can then execute the statement, get the ResultSet and iterate over it:

```
ResultSet rs = st.executeQuery("select * from employeeList");

while (rs.next()) {

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

}
```

PreparedStatement

PreparedStatement is a sub interface of the Statement interface. PreparedStatements are pre-compiled and hence their execution is much faster than that of Statements. You get a PreparedStatement object from a Connection object using the prepareStatement() method:

```
PreparedStatement ps1 = con.prepareStatement("insert into employeeList values ('Heartin',2)");

ps1.executeUpdate();
```

I can use any sql that I use in a Statement. One difference here is that in a Statement you pass the sql in the execute method, but in PreparedStatement you have to pass the sql in the prepareStatement() method while creating the PreparedStatement and leave the execute method empty. You can even override the sql statement passed in prepareStatement() by passing another one in the execute method, though it will not give the advantage of precompiling in a PreparedStatement.

PreparedStatement also has a set of setXXX() methods, with which you can parameterize a PreparedStatement as:

```
PreparedStatement ps1 = con.prepareStatement("insert into employeeList values (?,?)");

ps1.setString(1, "Heartin4");

ps1.setInt(2, 7);

ps1.executeUpdate();
```

This is very useful for inserting SQL 99 data types like BLOB and CLOB. Note that the index starts from 1 and not 0 and the setXXX() methods should be called before calling the executeUpdate() method.

CallableStatement

CallableStatement extends the capabilities of a PreparedStatement to include methods that are only appropriate for stored procedure calls; and hence CallableStatement is used to execute SQL stored procedures. Whereas PreparedStatement gives methods for dealing with IN parameters, CallableStatement provides methods to deal with OUT parameters as well.

Consider a stored procedure with signature as 'updateID(oldid in int,newid in int,username out varchar2)'.*Code to create this stored proc is given in the end.* We can use a CallableStatement for executing it as:

```
CallableStatement cs=con.prepareCall("{call updateID(?,?,?)}");

cs.setInt(1, 2);

cs.setInt(2, 4);

cs.registerOutParameter(3, java.sql.Types.VARCHAR);

cs.executeQuery();

System.out.println(cs.getString(3));
```

If there are no OUT parameters, we can even use PreparedStatement. But using a CallableStatement is the right way to go for stored procedures. A CallableStatement can return one ResultSet object or multiple ResultSet objects. Multiple ResultSet objects are handled using operations inherited from Statement. You can use getResultSet() to get a result as a ResultSet. The getResultSet() method should be called only once per result. So we should use getMoreResults() to move to this Statement's next result, which returns true if it is a ResultSet object. If true, then we can call getResultSet() again to get the next result as a ResultSet object.

For maximum portability, a call's ResultSet objects and update counts should be processed prior to getting the values of output parameters. *Statement vs PreparedStatement vs CallableStatement is a favorite JDBC topic for many interviewers.*

ResultSet

This interface represents a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

ParameterMetaData

ParameterMetaData retrieves the type and properties of the parameters used in the PreparedStatement interface. For some queries and driver implementations, the data that would be returned by a ParameterMetaData object may not be available until the PreparedStatement has been executed. Some driver implementations may even not be able to provide information about the types and properties for

each parameter marker in a CallableStatement object. The below example however worked fine for me with Oracle thin driver and Oracle XE database and printed 3.

```
CallableStatement cs=con.prepareCall("{call updateID(?,?,?)}");

cs.setInt(1, 2);

cs.setInt(2, 4);

cs.registerOutParameter(3, java.sql.Types.VARCHAR);

ParameterMetaData paramMetaData = cs.getParameterMetaData();

System.out.println("Count="+paramMetaData.getParameterCount());

cs.executeQuery();
```

We can use ParameterMetaData with CallableStatement as well as in this example because CallableStatement is also a PreparedStatement.

ResultSetMetaData

ResultSetMetaData is used to retrieve information about the count, types and the properties of columns used in a ResultSet object. Consider an example where you are passed a ResultSet to a method and you don't know the number or type of columns in the ResultSet. You can use ResultSetMetaData to get the column details and then find the corresponding row values using the column details.

```
public static void printColumnNames(ResultSet resultSet) throws SQLException {

    if (resultSet != null) {

      ResultSetMetaData rsMetaData = resultSet.getMetaData();

      int numberOfColumns = rsMetaData.getColumnCount();

      for (int i = 1; i < numberOfColumns + 1; i++) {

        String columnName = rsMetaData.getColumnName(i);

        System.out.println("column name=" + columnName);

      }

  }

}
```

RowId

RowId maps a java object with a RowId. The RowId is a built-in datatype and is used as the identification key of a row in a database table, especially when there are duplicate rows.

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

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

  oracle.sql.ROWID rowid = (ROWID)rs.getRowId("columnName");

  System.out.println(rowid);

}
```

We need to provide columnName or column index to getRowId same as rs.getString().

SQLException

This class is an exception class that provides information on a database access error or other errors. JDBC 4.0 introduced following refined subclasses of SQLException:

- java.sql.SQLClientInfoException
- java.sql.SQLDataException
- java.sql.SQLFeatureNotSupportedException
- java.sql.SQLIntegrityConstraintViolationException
- java.sql.SQLInvalidAuthorizationSpecException
- java.sql.SQLSyntaxErrorException
- java.sql.SQLTransactionRollbackException
- java.sql.SQLTransientConnectionException

There are still more interfaces which are part of JDBC core API (java.sql package). You can look through the documentation and try them all as part of learning or as required.

Database Code: updateID stored procedure used in examples

```
create or replace procedure updateID(oldid in int,newid in int,username out varchar2 )

is

begin

update student set id=newid where id=oldid;

select firstname into username from student where id=newid;
```

```
commit;

end updateID;
```

Source : http://javajee.com/important-jdbc-core-api-classes-interfaces-and-exceptions