CHAPTER 2

JDBC-JAVA DATABASE CONNECTIVITY

1. Introduction:

Java DataBase Connectivity (JDBC) is one of the most widely used API in enterprise applications, because most of the applications use some sort of database connectivity. I have recently posted a lot of JDBC tutorials related to basic JDBC, DataSource and it's integration with Spring Framework. This is a summary post where all those articles are listed, if you are new to JDBC then you should go through them in order for better understanding.

Java Database Connectivity (**JDBC**) API provides industry-standard and database-independent connectivity between the java applications and database servers. Just like java programs that we can "write once and run everywhere", JDBC provides framework to connect to relational databases from java programs.
JDBC API is used to achieve following tasks:

- Establishing a connection to relational Database servers like Oracle, MySQL etc. JDBC API doesn't provide framework to connect to NoSQL databases like MongoDB.
- Send SQL queries to the Connection to be executed at database server.
- Process the results returned by the execution of the query.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as :

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

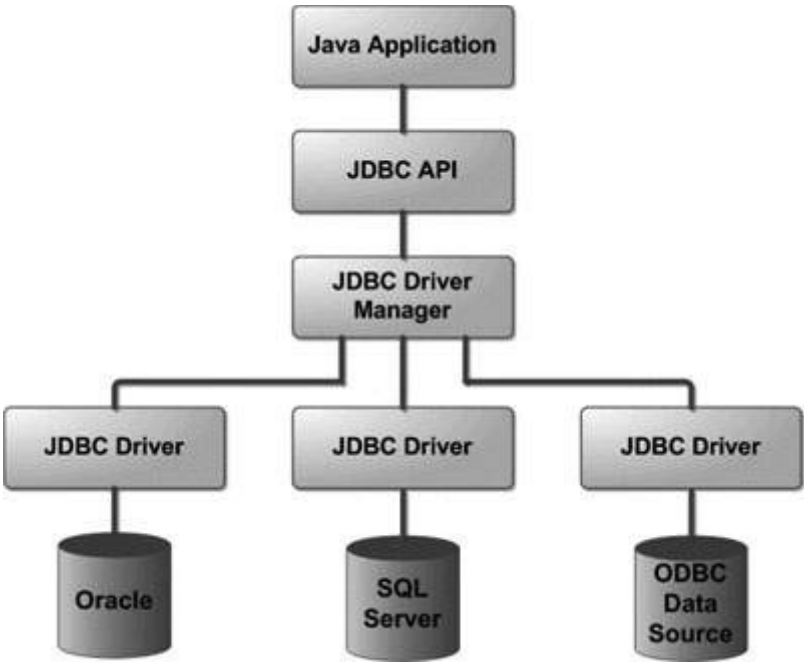
All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.
JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

2. JDBC Architecture:

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers :

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.
Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application :



3. 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 errors that occur in a database application.

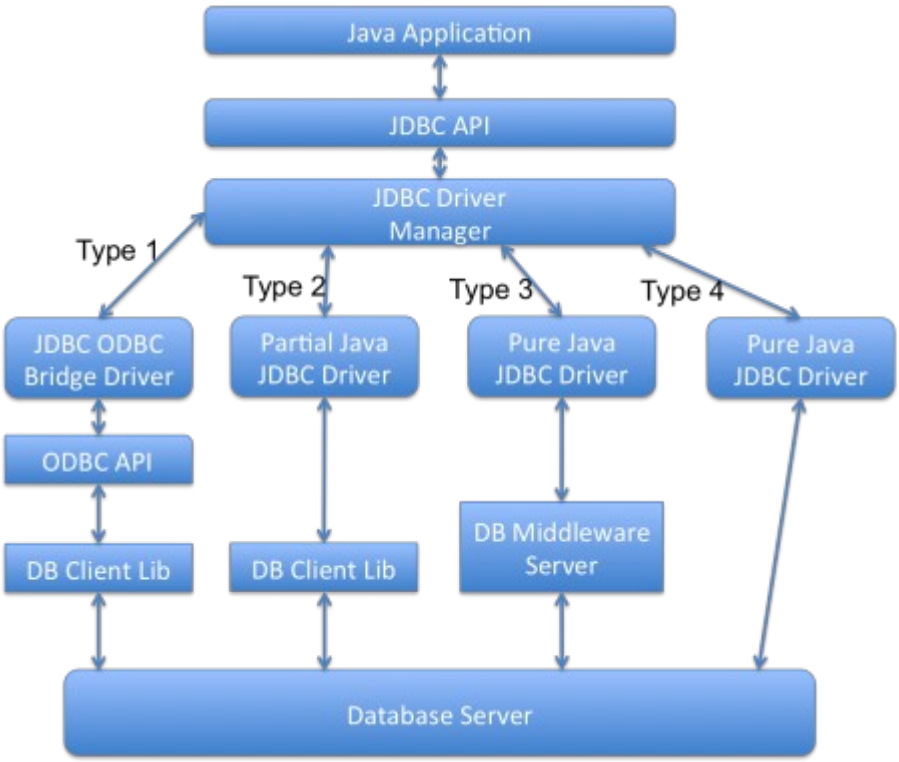
4. Creating JDBC Application:

There are following six steps involved in building a JDBC application −

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the DriverManager.getConnection() method to create a connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate ResultSet.getXXX() method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

5. JDBC Drivers:

JDBC API consists of two parts – first part is the JDBC API to be used by the application programmers and second part is the low-level API to connect to database. First part of JDBC API is part of standard java packages in java.sql package.
For second part there are four different types of JDBC drivers:



1. **JDBC-ODBC Bridge plus ODBC Driver** (Type 1): This driver uses ODBC driver to connect to database servers. We should have ODBC drivers installed in the machines from where we want to connect to database, that's why this driver is almost obsolete and should be used only when other options are not available.
2. **Native API partly Java technology-enabled driver** (Type 2): This type of driver converts JDBC class to the client API for the RDBMS servers. We should have database client API installed at the machine from which we want to make database connection. Because of extra dependency on database client API drivers, this is also not preferred driver.
3. **Pure Java Driver for Database Middleware** (Type 3): This type of driver sends the JDBC calls to a middleware server that can connect to different type of databases. We should have a middleware server installed to work with this kind of driver. This adds to extra network calls and slow performance. Hence this is also not widely used JDBC driver.
4. **Direct-to-Database Pure Java Driver** (Type 4): This is the preferred driver because it converts the JDBC calls to the network protocol understood by the database server. This solution doesn't require any extra APIs at the client side and suitable for database connectivity over the network. However for this solution, we should use database specific drivers, for example OJDBC jars provided by Oracle for Oracle DB and MySQL Connector/J for MySQL databases.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type

is 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.
After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

6. JDBC Connection:

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:

1. **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
2. **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
3. **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
4. **Create Connection Object:** Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.
5. **Close Connections :** Finally, we have to explicitly close the connections that we have opened.

1. Import JDBC Packages:
The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.
To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code :

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

2. Register JDBC Driver:
You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.
You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I : Class.forName():
The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method

is preferable because it allows you to make the driver registration configurable and portable.
The following example uses Class.forName( ) to register the Oracle driver :

```
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
    }
catch(ClassNotFoundException ex) {
          System.out.println("Error: unable to load driver class!");
          System.exit(1);
     }
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows :

```
try {
                                Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
         }
catch(ClassNotFoundException ex) {
          System.out.println("Error: unable to load driver class!");
          System.exit(1);
         }
catch(IllegalAccessException ex) {
          System.out.println("Error: access problem while loading!");
          System.exit(2);
         }
catch(InstantiationException ex) {
          System.out.println("Error: unable to instantiate driver!");
         System.exit(3);
         }
```

Approach II : DriverManager.registerDriver():
The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.
You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver :

```
try {
         Driver myDriver = new oracle.jdbc.driver.OracleDriver();
         DriverManager.registerDriver( myDriver );
         }
catch(ClassNotFoundException ex) {
         System.out.println("Error: unable to load driver class!");
         System.exit(1);
         }
```

3. Database URL Formulation:

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods :

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database. Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

4. Create Connection Object:

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password:

The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be :

jdbc:oracle:thin:@amrood:1521:EMP

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows :

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password";
Connection conn = DriverManager.getConnection(URL, USER, PASS) ;
```

Using Only a Database URL:

A second form of the DriverManager.getConnection( ) method requires only a database URL :

DriverManager.getConnection(String url);

However, in this case, the database URL includes the username and password and has the following general form :

jdbc:oracle:driver:username/password@database

So, the above connection can be created as follows :

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object:

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object :

DriverManager.getConnection(String url, Properties info);

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code :

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );
Connection conn = DriverManager.getConnection(URL, info);
```

5. Close JDBC Connections:

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows :

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database

administrator happy.

7. JDBC Statements:

Therr are three types of Statements available as given below.

1. Statement:
It can be used for general-purpose access to database
It can be used for general-purpose access to the database. It is useful when you are using static SQL statements at runtime.

2. PreparedStatement:
It can be used when you plan to use the same SQL statements many times. The PreparedStatement interface accepts input parameters at run time.

3. CallableStatement:
CallableStatement can be used when you want to access database stored procedures.

JDBC Prepared Statement:
JDBC PreparedStatement can be used when you plan to use the same SQL statement many times. It is used to handle precompiled query. If we want to execute same query with different values for more than one time then precompiled queries will reduce the no of compilations. Connection.prepareStatement() method can provide you PreparedStatment object. This object provides setXXX() methods to provide query values. Below example shows how to use PreparedStatement.

```
String query = "insert into emp(name,salary) values(?,?)";
PreparedStatement prSt = con.prepareStatement(query);
prSt.setString(1, "John");
prSt.setInt(2, 10000);
//count will give you how many records got updated
int count = prSt.executeUpdate();
//Run the same query with different values
prSt.setString(1, "Cric");
prSt.setInt(2, 5000);
count = prSt.executeUpdate();
```

JDBC prepared statement with ResultSet:

```
ResultSet rs=null;
PreparedStatement prSt=null;
String query = "select * from emp where empid=?";
prSt = con.prepareStatement(query);
prSt.setInt(1, 1016);
rs = prSt.executeQuery();
```

```
while(rs.next()){
    System.out.println(rs.getString("name")+" -- "+rs.getInt("salary"));
}
rs.close();
prSt.setInt(1, 1416);
rs = prSt.executeQuery();
while(rs.next()){
     System.out.println(rs.getString("name")+" -- "+rs.getInt("salary"));
}
rs.close();
```

How to get primary key value (auto-generated keys) from inserted queries using JDBC?

When we are inserting a record into the database table and the primary key is an auto-increment or auto-generated key, then the insert query will generate it dynamically. The below example shows how to get this key after insert statement. After perfoming executeUpdate() method on PreparedStatement, call getGeneratedKeys() method on PreparedStatement. It will return you ResultSet, from which you can get auto increment column values.

```
ResultSet rs=null;
PreparedStatement pstmt=null;
String query = "insert into emps (name, dept, salary) values (?,?,?)";
pstmt = con.prepareStatement(query,Statement.RETURN_GENERATED_KEYS);
pstmt.setString(1, "John");
pstmt.setString(2, "Acc Dept");
pstmt.setInt(3, 10000);
pstmt.executeUpdate();
rs = pstmt.getGeneratedKeys();
if(rs != null && rs.next()){
        System.out.println("Generated Emp Id: "+rs.getInt(1));
}
```

JDBC Callable Statement:

A CallableStatement object provides a way to call stored procedures using JDBC. Connection.prepareCall() method provides you CallableStatement object. Below example shows how to call stored procedure.

```
CallableStatement callSt=null;
callSt = con.prepareCall("{call myprocedure(?,?)}");
callSt.setInt(1,200);
callSt.setDouble(2, 3000);
callSt.execute();
System.out.println("Executed stored procedure!!!");
```

Write a program for CallableStatement statement with stored procedure returns OUT parameters.

A CallableStatement object provides a way to call stored procedures using JDBC. Connection.prepareCall() method provides you CallableStatement object. Below example shows how to call stored procedure with out parameters. You can regester output parameters data types by using registerOutParameter() method.

```
CallableStatement callSt=null;
callSt = con.prepareCall("{call myprocedure(?,?)}");
callSt.setInt(1,200);
//below method used to register data type of the out parameter
callSt.registerOutParameter(2, Types.DOUBLE);
callSt.execute();
Double output = callSt.getDouble(2);
System.out.println("The output returned from stored procedure: "+output);
```

Example : CallableStatement statement with batch execution.

We can do batch execution with CallableStatement. Below example shows how to do batch execution using addBatch() method and executeBatch() method

```
CallableStatement callSt=null;
callSt = con.prepareCall("{call myprocedure(?,?)}");
callSt.setInt(1,200);
callSt.setDouble(2, 3000);
callSt.addBatch();
callSt.setInt(1,130);
callSt.setDouble(2, 2000);
callSt.addBatch();
int[] updateCounts = callSt.executeBatch();
```

Example : to execute SQL function using CallableStatement:

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value. Below example shows how to execute a SQL function using CallableStatement. Below example shows how to call a SQL function using CallableStatement. The query syntax should be {?= call myfunction(?,?)}

```
CallableStatement callSt=null;
callSt = con.prepareCall("{?= call myfunction(?,?)}");
callSt.setInt(1,200);
//below method used to register data type of the out parameter
callSt.registerOutParameter(2, Types.DOUBLE);
callSt.execute();
Double output = callSt.getDouble(2);
System.out.println("The output returned from sql function: "+output);
```

8. ResultSetMetaData:

ResultSetMetaData is an object that can be used to get information about the types and properties of the columns in a ResultSet object. Below example shows how to get ResultSet column properties using ResultSetMetaData object.

```
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
for(int i=0;i<=columnCount;i++){
        System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnType(i));
}
```

9. DatabaseMetaData:

DatabaseMetaData is used to know which type of driver we are using and whether is it compatable or JDBC complaint or not. It is used to know all details about database provider as well.

```
DatabaseMetaData dm = con.getMetaData();
System.out.println(dm.getDriverVersion());
System.out.println(dm.getDriverName());
System.out.println(dm.getDatabaseProductName());
System.out.println(dm.getDatabaseProductVersion());
```

10. Batch Update :

Batch update is nothing but executing set of queries at a time. Batch updates reduces number of database calls. In batch processing, batch should not contain select query. You can add queries by calling addBatch() method and can execute the bunch of queries by calling executeBatch() method.

1. Using Statement:

When using batch updates with Statement object, you can use multiple types of queries which can be acceptable in executeUpdate() method.

```
Statement st = null;
....
con.setAutoCommit(false);
st = con.createStatement();
st.addBatch("update emp set sal=3000 where empid=200");
st.addBatch("insert into emp values (100,4000)");
st.addBatch("update emp set emp name='Ram' where empid=345");
int count[] = st.executeBatch();
for(int i=1;i<=count.length;i++){
      System.out.println("Query "+i+" has effected "+count[i]+" times");
}
con.commit();
```

2. Using PreparedStatement:

Batch update is nothing but executing set of queries at a time. Batch updates reduces number of database calls. In batch processing, batch should not contain select query. When we are using PreparedStatement to execute batch update, we have to run the same query multiple times. Below examples shows how to do batch updates with PreparedStatement.

```
PreparedStatement pst = null;
.....
con.setAutoCommit(false);
pst = con.prepareStatement("update emp set sal=? where empid=?");
pst.setInt(1, 3000);
pst.setInt(2, 200);
pst.addBatch();
pst.setInt(1, 4000);
pst.setInt(2, 230);
pst.addBatch();
int count[] = pst.executeBatch();
for(int i=1;i<=count.length;i++){
        System.out.println("Query "+i+" has effected "+count[i]+" times");
}
con.commit();
```

11. Types of ResultSets in JDBC(Updatable Resultset):

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:

1. 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.

2.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.

3.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 retrived.

The default ResultSet type is **TYPE_FORWARD_ONLY.**

Scrollable Result Set With Read Only Mode:

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.

```
Statement st = null;
....
st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
rs = st.executeQuery("select accno, bal from bank");
System.out.println("ResultSet Curson is at before first: "+rs.isBeforeFirst());
while(rs.next()){
        System.out.println(rs.getInt(1)+" "+rs.getDouble(2));
}
//now result set cursor reached the last position
System.out.println("Is After Last: "+rs.isAfterLast());
while(rs.previous()){
        System.out.println(rs.getInt(1)+" "+rs.getDouble(2));
}
```

Scrollable Result Set With Updatable Mode:

```
Statement st = null;
....
st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
rs = st.executeQuery("select accno, bal from bank");
while(rs.next()){
        if(rs.getInt(1) == 100){
                rs.updateDouble(2, 2000);
                rs.updateRow();
                System.out.println("Record updated!!!");
        }
}
```

How to insert an image into database table? or Write an example for inserting BLOB into table:

BLOB is nothing bug Binary Large Object. BLOB is used to store large amount of binary data into database like images, etc. Below example shows how to store images into database rows.

```
PreparedStatement ps = null;
.....
```

```
ps = con.prepareCall("insert into student_profile values (?,?)");
ps.setInt(1, 101);
is = new FileInputStream(new File("Student_img.jpg"));
ps.setBinaryStream(2, is);
int count = ps.executeUpdate();
System.out.println("Count: "+count);
```

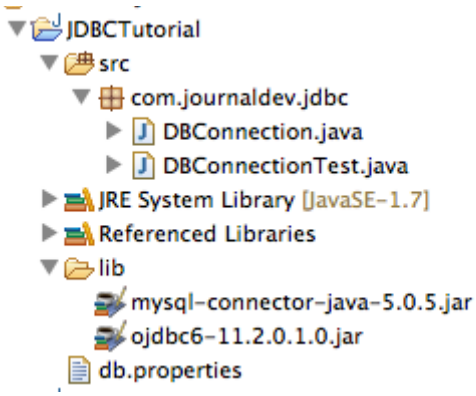
How to read an image from database table? or Write an example for reading BLOB from table.

BLOB is nothing bug Binary Large Object. BLOB is used to store and retrieve large amount of binary data from database like images, etc. Below example shows how to read images from database rows.

```
Statement st=null;
FileInputStream is=null;
FileOutputStream os=null;
st = con.createStatement();
rs = st.executeQuery("select student_img from student_profile where id=101");
if(rs.next()){
        is = rs.getBinaryStream(1);
}
is = new FileInputStream(new File("Student_img.jpg"));
os = new FileOutputStream("std_img.jpg");
byte[] content = new byte[1024];
int size = 0;
while((size = is.read(content)) != -1){
        os.write(content, 0, size);
}
```

12. JDBC Project:

Let's create a simple JDBC Example Project and see how JDBC API helps us in writing loosely-coupled code for database connectivity.



Before starting with the example, we need to do some prep work to have some data in the database servers to query. Installing the database servers is not in the scope of this tutorial, so I will assume that you have database servers installed.
We will write program to connect to database server and run a simple query and process the results. For showing how we can achieve loose-coupling in connecting to databases using JDBC API, I will use Oracle and MySQL database systems.

1. Run below SQL scripts to create the table and insert some dummy values in the table.

```
--mysql create table
create table Users(
id int(3) primary key,
name varchar(20),
email varchar(20),
country varchar(20),
password varchar(20)
);

--oracle create table
create table Users(
id number(3) primary key,
name varchar2(20),
email varchar2(20),
country varchar2(20),
password varchar2(20)
);

--insert rows
INSERT INTO Users (id, name, email, country, password)
VALUES (1, 'Pankaj', 'pankaj@apple.com', 'India', 'pankaj123');
INSERT INTO Users (id, name, email, country, password)
VALUES (4, 'David', 'david@gmail.com', 'USA', 'david123');
INSERT INTO Users (id, name, email, country, password)
VALUES (5, 'Raman', 'raman@google.com', 'UK', 'raman123');
commit;
```

Notice that datatypes in Oracle and MySQL databases are different, that's why I have provided two different SQL DDL queries to create Users table. However both the databases confirms to SQL language, so insert queries are same for both the database tables.

2. Database Drivers:
As you can see in the project image, I have both MySQL (mysql-connector-java-5.0.5.jar) and

Oracle (ojdbc6-11.2.0.1.0.jar) type-4 drivers in the lib directory and added to the project build path. Make sure you are using the correct version of the java drivers according to your database server installation version. Usually these jars shipped with the installer, so you can find them in the installation package.

3. Database Configurations:

We will read the database configuration details from the property files, so that we can easily switch from Oracle to MySQL database and vice versa, just by changing the property details.

```
#mysql DB properties
#DB_DRIVER_CLASS=com.mysql.jdbc.Driver
#DB_URL=jdbc:mysql://localhost:3306/UserDB
#DB_USERNAME=pankaj
#DB_PASSWORD=pankaj123

#Oracle DB Properties
DB_DRIVER_CLASS=oracle.jdbc.driver.OracleDriver
DB_URL=jdbc:oracle:thin:@localhost:1571:MyDBSID
DB_USERNAME=scott
DB_PASSWORD=tiger
```

Database configurations are the most important details when using JDBC API. The first thing we should know is the Driver class to use. For Oracle database, driver class is oracle.jdbc.driver.OracleDriver and for MySQL database, driver class is com.mysql.jdbc.Driver. You will find these driver classes in their respective driver jar files and both of these implement java.sql.Driver interface.

The second important part is the database connection URL string. Every database driver has it's own way to configure the database URL but all of them have host, port and Schema details in the connection URL. For MySQL connection String format is jdbc:mysql://<HOST>:<PORT>/<SCHEMA> and for Oracle database connection string format is jdbc:oracle:thin:@<HOST>:<PORT>:<SID>.

The other important details are database username and password details to be used for connecting to the database.

4. JDBC Connection Program:

Let's see a simple program to see how we can read above properties and create database connection.

DBConnection.java

```
public class DBConnection {
	public static Connection getConnection() {
		Properties props = new Properties();
		FileInputStream fis = null;
		Connection con = null;
```

```
                try {
                        fis = new FileInputStream("db.properties");
                        props.load(fis);

                        // load the Driver Class
                        Class.forName(props.getProperty("DB_DRIVER_CLASS"));

                        // create the connection now
                        con = DriverManager.getConnection(props.getProperty("DB_URL"),
                                        props.getProperty("DB_USERNAME"),
                                        props.getProperty("DB_PASSWORD"));
                        }
                catch (IOException | ClassNotFoundException | SQLException e) {
                        e.printStackTrace();
                }
                return con;
        }
}
```

The program is really simple, first we are reading database configuration details from the property file and then loading the JDBC driver and using DriverManager to create the connection. Notice that this code use only Java JDBC API classes and there is no way to know that it's connecting to which type of database. This is also a great example of *writing code for interfaces* methodology.
The important thing to notice is the *Class.forName()* method call, this is the Java Reflection method to create the instance of the given class. You might wonder why we are using Reflection and not *new* operator to create the object and why we are just creating the object and not using it.

The first reason is that using reflection to create instance helps us in writing loosely-coupled code that we can't achieve if we are using new operator. In that case, we could not switch to different database without making corresponding code changes.

The reason for not using the object is because we are not interested in creating the object. The main motive is to load the class into memory, so that the driver class can register itself to the DriverManager. If you will look into the Driver classes implementation, you will find that they have static block where they are registering themselves to DriverManager.

5. oracle.jdbc.OracleDriver

```
static
        {
                Driver defaultDriver=null;
                try
                {
                        if (defaultDriver == null)
                        {
                                defaultDriver = new oracle.jdbc.OracleDriver();
```

```
                    DriverManager.registerDriver(defaultDriver);
                }
                //some code omitted for clarity
            }
            catch (IOException | ClassNotFoundException | SQLException e) {
                e.printStackTrace();
            }
    }
```

6. com.mysql.jdbc.Driver

```
static
    {
        try
        {
            DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
```

This is a great example where we are making our code loosely-coupled with the use of reflection API. So basically we are doing following things using Class.forName() method call.

```
Driver driver = new OracleDriver();
DriverManager.registerDriver(driver);
```

DriverManager.getConnection() method uses the registered JDBC drivers to create the database connection and it throws java.sql.SQLException if there is any problem in getting the database connection.Now let's write a simple test program to use the database connection and run simple query.

7. JDBC Statement and ResultSet:

Here is a simple program where we are using the JDBC Connection to execute SQL query against the database and then processing the result set.

DBConnectionTest.java

```
class DBConnectionTest {
private static final String QUERY = "select id,name,email,country,password from Users";
public static void main(String[] args) {
//using try-with-resources to avoid closing resources (boiler plate code)
        try
        {
```

```
                Connection con = DBConnection.getConnection();
                Statement stmt = con.createStatement();
                ResultSet rs = stmt.executeQuery(QUERY);

                while(rs.next()){
                        int id = rs.getInt("id");
                        String name = rs.getString("name");
                        String email = rs.getString("email");
                        String country = rs.getString("country");
                        String password = rs.getString("password");
                        System.out.println(id + "," +name+ "," +email+ "," +country+ ","
+password);
                }
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        } catch (SQLException e) {

                e.printStackTrace();
        }
  }
}
```

We are using Java 7 try-with-resources feature to make sure that resources are closed as soon as we are out of try-catch block. JDBC Connection, Statement and ResultSet are expensive resources and we should close them as soon as we are finished using them.
Connection.createStatement() is used to create the Statement object and then executeQuery() method is used to run the query and get the result set object.
First call to ResultSet next() method call moves the cursor to the first row and subsequent calls moves the cursor to next rows in the result set. If there are no more rows then it returns false and come out of the while loop. We are using result set getXXX() method to get the columns value and then writing them to the console.

When we run above test program, we get following output.

```
1,Pankaj,pankaj@apple.com,India,pankaj123
4,David,david@gmail.com,USA,david123
5,Raman,raman@google.com,UK,raman123
```

Just uncomment the MySQL database configuration properties from db.properties file and comment the Oracle database configuration details to switch to MySQL database. Since the data is same in both Oracle and MySQL database Users table, you will get the same output.

8. Update a record in the database using JDBC:

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.SQLException;
import java.sql.Statement;

class MyQueryUpdate {
	public static void main(String[] args) {
		Connection con = null;
		Statement stmt = null;
		try {
			Class.forName("oracle.jdbc.driver.OracleDriver");
			con = DriverManager.getConnection("jdbc:oracle:thin:@<hostname>:<port
num>:<DB name>","user","password");
			stmt = con.createStatement();
			String query = "update table emp set salary=2000 where empid=200";
			//count will give you how many records got updated
			int count = stmt.executeUpdate(query);
			System.out.println("Updated queries: "+count);
		}
		catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		catch (SQLException e) {
			e.printStackTrace();
		}
		finally {
			try{
				if(stmt != null) stmt.close();
				if(con != null) con.close();
			} catch(Exception ex){}
		}
	}
}
```