JDBC - Quick Guide

What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

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

Pre-Requisite:

You need to have good understanding on the following two subjects to learn JDBC:

- Core JAVA Programming
- SQL or MySQL Database

JDBC - Environment Setup:

Make sure you have done following setup:

- Core JAVA Installation
- SQL or MySQL Database Installation

Apart from the above you need to setup a database which you would use for your project. Assuming this is EMP and you have created on table Employees within the same database.

Creating JDBC Application:

There are six steps involved in building a JDBC application which I'm going to brief in this tutorial:

Import the packages:

This requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice as follows:

```
//STEP 1. Import required packages
import java.sql.*;
```

Register the JDBC driver:

This requires that you initialize a driver so you can open a communications channel with the database. Following is the code snippet to achieve this:

```
//STEP 2: Register JDBC driver
Class.forName("com.mysql.jdbc.Driver");
```

Open a connection:

This requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database as follows:

```
//STEP 3: Open a connection
//  Database credentials
static final String USER = "username";
static final String PASS = "password";
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

Execute a query:

This requires using an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database as follows:

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
```

If there is an SQL UPDATE,INSERT or DELETE statement required, then following code snippet would be required:

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "DELETE FROM Employees";
ResultSet rs = stmt.executeUpdate(sql);
```

Extract data from result set:

This step is required in case you are fetching data from the database. You can use the appropriate ResultSet.getXXX() method to retrieve the data from the result set as follows:

```
//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id  = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");

    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
```

Clean up the environment:

You should explicitly close all database resources versus relying on the JVM's garbage collection as follows:

```
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();
```

First JDBC Program:

Based on the above steps, we can have following consolidated sample code which we can use as a template while writing our JDBC code:

This sample code has been written based on the environment and database setup done in Environment chapter.

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DB_URL = "jdbc:mysql://localhost/EMP";
```

```
//  Database credentials
static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
   //STEP 2: Register JDBC driver
   Class.forName("com.mysql.jdbc.Driver");

   //STEP 3: Open a connection
   System.out.println("Connecting to database...");
   conn = DriverManager.getConnection(DB_URL,USER,PASS);

   //STEP 4: Execute a query
   System.out.println("Creating statement...");
   stmt = conn.createStatement();
   String sql;
   sql = "SELECT id, first, last, age FROM Employees";
   ResultSet rs = stmt.executeQuery(sql);

   //STEP 5: Extract data from result set
   while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
   }
   //STEP 6: Clean-up environment
   rs.close();
   stmt.close();
   conn.close();
}catch(SQLException se){
   //Handle errors for JDBC
   se.printStackTrace();
}catch(Exception e){
   //Handle errors for Class.forName
   e.printStackTrace();
```

```
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile above example as follows:

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

SQLException Methods:

A SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.

The passed SQLException object has the following methods available for retrieving additional information about the exception:

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and its backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and its backtrace to the print writer you specify.

By utilizing the information available from the Exception object, you can catch an exception and continue your program appropriately. Here is the general form of a try block:

```
try {
   // Your risky code goes between these curly braces!!!
}
catch(Exception ex) {
   // Your exception handling code goes between these
   // curly braces, similar to the exception clause
   // in a PL/SQL block.
}
finally {
   // Your must-always-be-executed code goes between these
   // curly braces. Like closing database connection.
}
```

JDBC - Data Types:

The following table summarizes the default JDBC data type that the Java data type is converted to when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

SQL	JDBC/Java	setXXX	updateXXX
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB(), updateCLOB(), updateArray(), and updateRef() methods that

enable you to directly manipulate the respective data on the server.

The setXXX() and updateXXX() methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject() and updateObject(), enable you to map almost any Java type to a JDBC data type.

ResultSet object provides corresponding getXXX() method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

SQL	JDBC/Java	setXXX	getXXX
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

JDBC - Batch Processing:

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

- JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

JDBC - Streaming Data:

A PreparedStatement object has the ability to use input and output streams to supply parameter data. This enables you to place entire files into database columns that can hold large values, such as CLOB and BLOB data types.

There are following methods which can be used to stream data:

- **setAsciiStream():** This method is used to supply large ASCII values.
- **setCharacterStream():** This method is used to supply large UNICODE values.
- **setBinaryStream():** This method is used to supply large binary values.

The setXXXStream() method requires an extra parameter, the file size, besides the parameter placeholder. This parameter informs the driver how much data should be sent to the database using the stream.

For a detail on all these concept, you need to go through the complete tutorial.