JournalDev

Java Database Connectivity

Pankaj Kumar

Table of Content

1. JDBC Overview

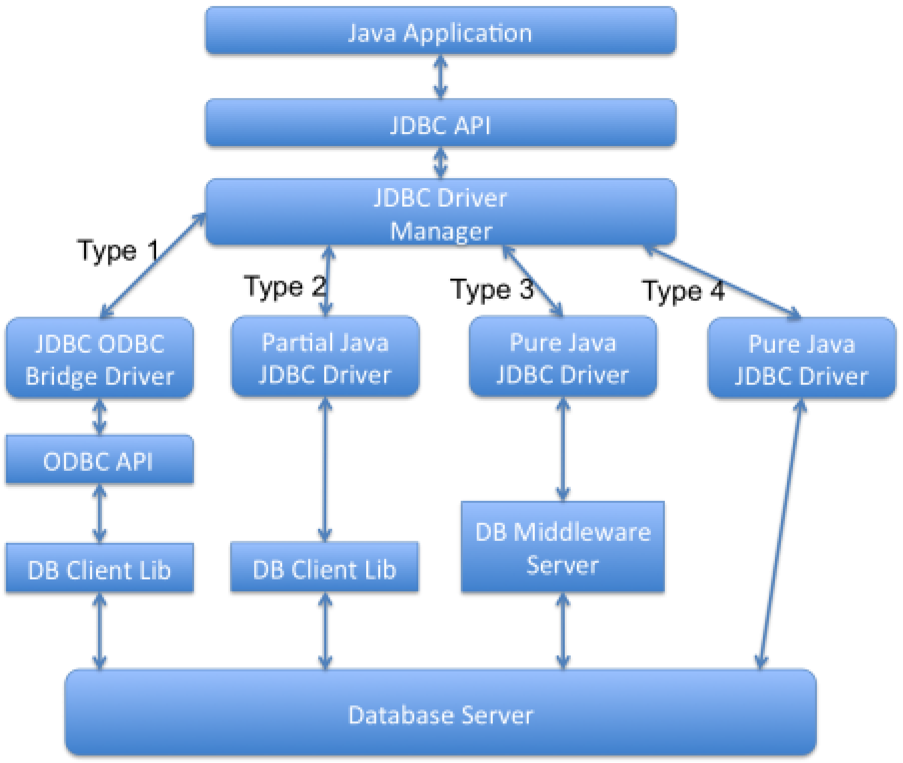
Java Database Connectivity (**JDBC**) API provides industry-standard and database-independent connectivity between the java applications and database servers. Just like java programs that we can "write once and run everywhere", JDBC provides framework to connect to relational databases from java programs.

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.

A. JDBC Drivers

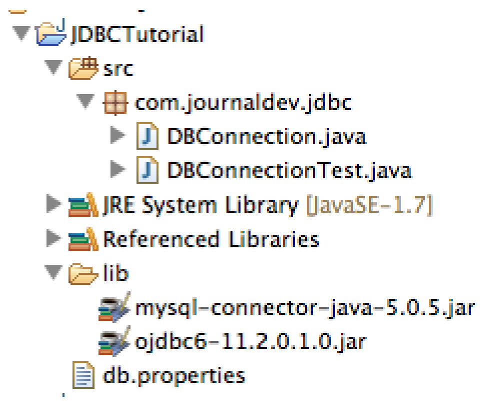
JDBC API consists of two parts – first part is the JDBC API to be used by the application programmers and second part is the low-level API to connect to database. First part of JDBC API is part of standard java packages in java.sql package. For second part there are four different types of JDBC drivers:



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.

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



```
JDBCTutorial
  src
    com.journaldev.jdbc
      DBConnection.java
      DBConnectionTest.java
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
  lib
    mysql-connector-java-5.0.5.jar
    ojdbc6-11.2.0.1.0.jar
  db.properties
```

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.

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.

B. 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.

C. 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.

D. JDBC Connection Program

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

```java
package com.journaldev.jdbc;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

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)
{
            // TODO Auto-generated catch block
            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.

```
static
  {
    try
    {
      if (defaultDriver == null)
      {
        defaultDriver = new oracle.jdbc.OracleDriver();
        DriverManager.registerDriver(defaultDriver);
      }
    //some code omitted for clarity
    }
}

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.

E. 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.

```
package com.journaldev.jdbc;
```

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

public 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 (SQLException e) {
            e.printStackTrace();
        }

    }

}
```

Notice that 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.

You can download the source code of "JDBC Example Project" from here.

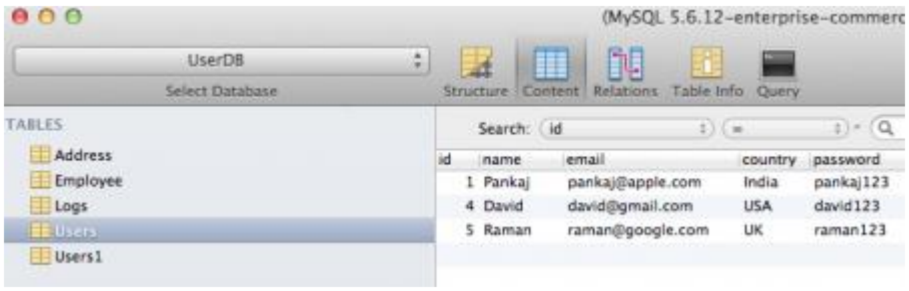
That's all for the JDBC example tutorial, as you can see that JDBC API helps us in writing driver independent code and makes it easier to switch to other relational databases if required. Download the project from above link and run different scenarios for better understanding.

2. JDBC Statement and PreparedStatement

While working with JDBC API for database connectivity, we can use Statement or PreparedStatement to execute queries. These queries can be CRUD operation queries or even DDL queries to create or drop tables.

JDBC Statement has some major issues and should be avoided in all cases, let's see this with a simple example.

I have **Users** table in my local MySQL database with following data.



id	name	email	country	password
1	Pankaj	pankaj@apple.com	India	pankaj123
4	David	david@gmail.com	USA	david123
5	Raman	raman@google.com	UK	raman123

Below script will create the table and insert the data for test use.

```
CREATE TABLE `Users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL DEFAULT '',
  `email` varchar(20) NOT NULL DEFAULT '',
  `country` varchar(20) DEFAULT 'USA',
  `password` varchar(20) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;

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

A utility class for creating JDBC Connection to our mysql database.

```
package com.journaldev.jdbc.statements;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnection {

    public final static String DB_DRIVER_CLASS =
"com.mysql.jdbc.Driver";
    public final static String DB_URL =
"jdbc:mysql://localhost:3306/UserDB";
    public final static String DB_USERNAME = "pankaj";
    public final static String DB_PASSWORD = "pankaj123";

    public static Connection getConnection() throws
ClassNotFoundException, SQLException {

        Connection con = null;

        // load the Driver Class
        Class.forName(DB_DRIVER_CLASS);

        // create the connection now
        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

        System.out.println("DB Connection created successfully");
        return con;
    }
}
```

Now let's say we have following class that asks user to enter the email id and password and if it matches, then prints the user details. I am using JDBC Statement for executing the query.

```
package com.journaldev.jdbc.statements;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

public class GetUserDetails {

    public static void main(String[] args) throws
ClassNotFoundException, SQLException {
        //read user entered data
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter email id:");
        String id = scanner.nextLine();
        System.out.println("User id="+id);
        System.out.println("Please enter password to get details:");
        String pwd = scanner.nextLine();
        System.out.println("User password="+pwd);
        printUserData(id,pwd);

    }

    private static void printUserData(String id, String pwd) throws
ClassNotFoundException, SQLException {

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try{
        con = DBConnection.getConnection();
        stmt = con.createStatement();
        String query = "select name, country, password from Users where
email = '"+id+"' and password='"+pwd+"'";
        System.out.println(query);
        rs = stmt.executeQuery(query);

        while(rs.next()){

System.out.println("Name="+rs.getString("name")+",country="+rs.getStrin
g("country")+",password="+rs.getString("password"));
        }
        }finally{
            if(rs != null) rs.close();
            stmt.close();
            con.close();
        }

    }

}
```

Let's see what happens when we pass different kinds of input to above program.

```
Please enter email id:
david@gmail.com
User id=david@gmail.com
Please enter password to get details:
david123
User password=david123
DB Connection created successfully
```

```
select name, country, password from Users where email =
'david@gmail.com' and password='david123'
Name=David,country=USA,password=david123
```

So our program works fine and a valid user can enter their credentials and get his details. Now let’s see how a hacker can get unauthorized access to a user because we are using Statement for executing queries.

```
Please enter email id:
david@gmail.com' or '1'='1
User id=david@gmail.com' or '1'='1
Please enter password to get details:

User password=
DB Connection created successfully
select name, country, password from Users where email =
'david@gmail.com' or '1'='1' and password=''
Name=David,country=USA,password=david123
```

As you can see that we are able to get the user details even without having password. The key point to note here is that query is created through String concatenation and if we provide proper input, we can hack the system, like here we did by passing user id as *david@gmail.com’ or ‘1’=’1*. This is an example of “**SQL Injection**” where poor programming is responsible for making our application vulnerable for unauthorized database access.

One solution is to read the user input and then escape all the special characters that are used by MySQL but that would be clumsy and error prone. That’s why JDBC API came up with *PreparedStatement* interface that extends *Statement* that automatically escape the special characters before executing the query. Let’s rewrite above class using PreparedStatement and try to hack the system.

```
package com.journaldev.jdbc.statements;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Scanner;

public class GetUserDetailsUsingPS {

    public static void main(String[] args) throws
ClassNotFoundException, SQLException {

        // read user entered data
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter email id:");
        String id = scanner.nextLine();
        System.out.println("User id=" + id);
        System.out.println("Please enter password to get details:");
        String pwd = scanner.nextLine();
        System.out.println("User password=" + pwd);
        printUserData(id, pwd);
    }

    private static void printUserData(String id, String pwd) throws
ClassNotFoundException,
            SQLException {

        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        String query = "select name, country, password from Users where
email = ? and password = ?";
        try {
            con = DBConnection.getConnection();
            ps = con.prepareStatement(query);

            //set the parameter
            ps.setString(1, id);
            ps.setString(2, pwd);
            rs = ps.executeQuery();

            while (rs.next()) {
                System.out.println("Name=" + rs.getString("name") +
",country="
                        + rs.getString("country") + ",password="
                        + rs.getString("password"));
            }
        } finally {
            if (rs != null)
                rs.close();
            ps.close();
            con.close();
        }

    }
}
```

```
Please enter email id:
david@gmail.com' or '1'='1
User id=david@gmail.com' or '1'='1
Please enter password to get details:

User password=
DB Connection created successfully
```

So we are not able to hack the database, it happened because the actual query that is getting executed is:

```
select name, country, password from Users where email = 'david@gmail.com\'
or \'1\'=\'1\' and password=''
```

When we fire a query to be executed for a relational database, it goes through following steps.

1. Parsing of SQL query
2. Compilation of SQL Query
3. Planning and optimization of data acquisition path
4. Executing the optimized query and return the resulted data

When we use Statement, it goes through all the four steps but with PreparedStatement first three steps are executed when we create the prepared statement. So execution of query takes less time and quicker that Statement.

Another benefit of using PreparedStatement is that we can use Batch Processing through *addBatch()* and *executeBatch()* methods. We can create a single prepared statement and use it to execute multiple queries.

Some points to remember about JDBC PreparedStatement are:

- PreparedStatement helps us in preventing SQL injection attacks because it automatically escapes the special characters.
- PreparedStatement allows us to execute dynamic queries with parameter inputs.
- PreparedStatement provides different types of setter methods to set the input parameters for the query.
- PreparedStatement is faster than Statement. It becomes more visible when we reuse the PreparedStatement or use its batch processing methods for executing multiple queries.
- PreparedStatement helps us in writing object Oriented code with setter methods whereas with Statement we have to use String Concatenation to create the query. If there are multiple parameters to set, writing Query using String concatenation looks very ugly and error prone.
- PreparedStatement returns *FORWARD_ONLY* ResultSet, so we can only move in forward direction.

- Unlike Java Arrays or List, the indexing of PreparedStatement variables starts with 1.
- One of the limitation of PreparedStatement is that we can’t use it for SQL queries with IN clause because PreparedStatement doesn’t allow us to bind multiple values for single placeholder (?). However there are few alternative approaches to use PreparedStatement for IN clause, read more at JDBC PreparedStatement IN clause.

That’s all for the comparison of JDBC Statement and PreparedStatement, you should always use PreparedStatement because it’s fast, object oriented, dynamic and more reliable.

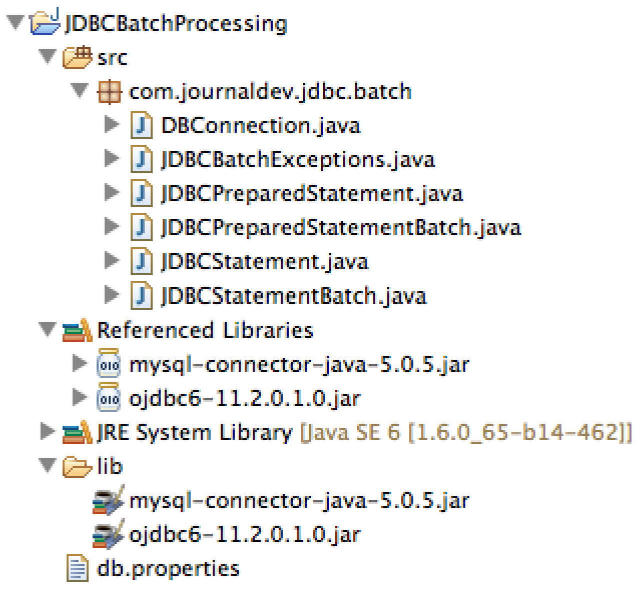
That’s all for the comparison of JDBC Statement and PreparedStatement, you should always use PreparedStatement because it’s fast, object oriented, dynamic and more reliable.

.3 JDBC Batch Processing

Sometimes we need to run bulk queries of similar kind for a database, for example loading data from CSV files to relational database tables. As we know that we have option to use Statement or PreparedStatement to execute queries. Apart from that JDBC API provides **Batch Processing** feature through which we can execute bulk of queries in one go for a database.

JDBC API supports batch processing through Statement and PreparedStatement addBatch() and executeBatch() methods. This tutorial is aimed to provide details about JDBC Batch API with example and how we should use it.

We will look into different programs so we have a project with structure as below image.



```
JDBCBatchProcessing
  src
    com.journaldev.jdbc.batch
      DBConnection.java
      JDBCBatchExceptions.java
      JDBCPreparedStatement.java
      JDBCPreparedStatementBatch.java
      JDBCStatement.java
      JDBCStatementBatch.java
  Referenced Libraries
    mysql-connector-java-5.0.5.jar
    ojdbc6-11.2.0.1.0.jar
  JRE System Library [Java SE 6 [1.6.0_65-b14-462]]
  lib
    mysql-connector-java-5.0.5.jar
    ojdbc6-11.2.0.1.0.jar
  db.properties
```

Notice that I have MySQL and Oracle DB JDBC Driver jars in the project build path, so that we can run our application across MySQL and Oracle DB both.

Let’s first create a simple table for our test programs. We will run bulk of JDBC insert queries and look at the performance with different approaches.

```
--Oracle DB
CREATE TABLE Employee (
  empId NUMBER NOT NULL,
  name varchar2(10) DEFAULT NULL,
  PRIMARY KEY (empId)
);

--MySQL DB
CREATE TABLE `Employee` (
  `empId` int(10) unsigned NOT NULL,
  `name` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`empId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We will read the Database configuration details from property file, so that switching from one database to another is quick and easy.

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

#Oracle DB Properties
#DB_DRIVER_CLASS=oracle.jdbc.driver.OracleDriver
#DB_URL=jdbc:oracle:thin:@localhost:1871:UserDB
#DB_USERNAME=scott
#DB_PASSWORD=tiger
```

Before we move into actual JDBC programming to insert bulk data into Employee table, let’s write a simple utility class to get the database connection.

```java
package com.journaldev.jdbc.batch;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

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 e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return con;
    }
}
```

Approach 1: Use Statement to execute one query at a time.

```java
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCStatement {

    public static void main(String[] args) {

        Connection con = null;
        Statement stmt = null;
        try {
            con = DBConnection.getConnection();
            stmt = con.createStatement();

            long start = System.currentTimeMillis();
            for(int i =0; i<10000;i++){
                String query = "insert into Employee values
("+i+",'Name"+i+"')";
                stmt.execute(query);
            }
            System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                stmt.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Approach 2: Use PreparedStatement to execute one query at a time.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCPreparedStatement {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name) values
(?,?)";
        try {
            con = DBConnection.getConnection();
            ps = con.prepareStatement(query);

            long start = System.currentTimeMillis();
            for(int i =0; i<10000;i++){
                ps.setInt(1, i);
                ps.setString(2, "Name"+i);
                ps.executeUpdate();
            }
            System.out.println("Time 
Taken="+(System.currentTimeMillis()-start));
            
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                ps.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

This approach is similar as using Statement but PreparedStatement provides performance benefits and avoid SQL injection attacks.

Approach 3: Using Statement Batch API for bulk processing.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCStatementBatch {

    public static void main(String[] args) {
        
        Connection con = null;
        Statement stmt = null;
        
        try {
            con = DBConnection.getConnection();
            stmt = con.createStatement();
            
            long start = System.currentTimeMillis();
            for(int i =0; i<10000;i++){
                String query = "insert into Employee values 
("+i+",'Name"+i+"')";
                stmt.addBatch(query);
                
                //execute and commit batch of 1000 queries
                if(i%1000 ==0) stmt.executeBatch();
            }
            //commit remaining queries in the batch
            stmt.executeBatch();

            System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                stmt.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

We are processing 10,000 records with batch size of 1000 records. Once the batch size reaches, we are executing it and continue processing remaining queries.

Approach 4: Using PreparedStatement Batch Processing API for bulk queries.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBCPreparedStatementBatch {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name) values
(?,?)";
        try {
            con = DBConnection.getConnection();
            ps = con.prepareStatement(query);

            long start = System.currentTimeMillis();
            for(int i =0; i<10000;i++){
                ps.setInt(1, i);
                ps.setString(2, "Name"+i);

                ps.addBatch();

                if(i%1000 == 0) ps.executeBatch();
            }
            ps.executeBatch();

            System.out.println("Time
Taken="+(System.currentTimeMillis()-start));

        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                ps.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Let's see how our programs work with MySQL database, I have executed them separately multiple times and below table contains the results.

MySQL DB	Statement	PreparedStatement	Statement Batch	PreparedStatement Batch
Time Taken (ms)	8256	8130	7129	7019

When I looked at the response time, I was not sure whether it's right because I was expecting some good response time improvements with Batch Processing. So I looked online for some explanation and found out that by default MySQL batch processing works in similar way like running without batch. To get the actual benefits of Batch Processing in MySQL, we need to pass **rewriteBatchedStatements** as TRUE while creating the DB connection. Look at the MySQL URL above in **db.properties** file for this.

With *rewriteBatchedStatements* as *true*, below table provides the response time for the same programs.

MySQL DB	Statement	PreparedStatement	Statement Batch	PreparedStatement Batch
Time Taken (ms)	5676	5570	3716	394

As you can see that PreparedStatement Batch Processing is very fast when rewriteBatchedStatements is true. So if you have a lot of batch processing involved, you should use this feature for faster processing.

A. Oracle DB Batch Processing

When I executed above programs for Oracle DB, the results were in line with MySQL processing results and PreparedStatement Batch processing was much faster than any other approach.

B. Batch Processing Exceptions

Let's see how batch programs behave in case one of the queries throw exceptions.

```java
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;

public class JDBCBatchExceptions {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name) values 
(?,?)";
        try {
            con = DBConnection.getConnection();

            ps = con.prepareStatement(query);

            String name1 = "Pankaj";
            String name2="Pankaj Kumar"; //longer than column length
            String name3="Kumar";

            ps.setInt(1, 1);
            ps.setString(2, name1);
            ps.addBatch();

            ps.setInt(1, 2);
            ps.setString(2, name2);
            ps.addBatch();
            ps.setInt(1, 3);
            ps.setString(2, name3);
            ps.addBatch();

            int[] results = ps.executeBatch();

            System.out.println(Arrays.toString(results));

        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
            try {
                ps.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }}
```

When I executed above program for MySQL database, I got below exception and none of the records were inserted in the table.

```
com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too long for
column 'name' at row 2
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2939)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
    at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java
:1268)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
541)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
455)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
440)
    at
com.mysql.jdbc.PreparedStatement.executeBatchedInserts(PreparedStatemen
t.java:1008)
    at
com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:90
8)
    at
com.journaldev.jdbc.batch.JDBCBatchExceptions.main(JDBCBatchExceptions.
java:37)
```

When executed the same program for Oracle database, I got below exception.

```
java.sql.BatchUpdateException: ORA-12899: value too large for column
"SCOTT"."EMPLOYEE"."NAME" (actual: 12, maximum: 10)

    at
oracle.jdbc.driver.OraclePreparedStatement.executeBatch(OraclePreparedS
tatement.java:10070)
    at
oracle.jdbc.driver.OracleStatementWrapper.executeBatch(OracleStatementW
rapper.java:213)
    at
com.journaldev.jdbc.batch.JDBCBatchExceptions.main(JDBCBatchExceptions.
java:38)
```

But the rows before exception were inserted into the database successfully. Although the exception clearly says what the error is but it doesn't tell us which query is causing the issue. So either we validate the data before adding them for batch processing or we should use JDBC Transaction Management to make sure all or none of the records are getting inserted incase of exceptions.

Same program with transaction management looks like below.

```
package com.journaldev.jdbc.batch;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Arrays;

public class JDBCBatchExceptions {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement ps = null;
        String query = "insert into Employee (empId, name) values
(?,?)";
        try {
            con = DBConnection.getConnection();
            con.setAutoCommit(false);

            ps = con.prepareStatement(query);

            String name1 = "Pankaj";
            String name2="Pankaj Kumar"; //longer than column length
            String name3="Kumar";
            ps.setInt(1, 1);
            ps.setString(2, name1);
            ps.addBatch();

            ps.setInt(1, 2);
            ps.setString(2, name2);
            ps.addBatch();

            ps.setInt(1, 3);
            ps.setString(2, name3);
            ps.addBatch();

            int[] results = ps.executeBatch();

            con.commit();
            System.out.println(Arrays.toString(results));

        } catch (SQLException e) {
            e.printStackTrace();
            try {
                con.rollback();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
        }finally{
            try {
                ps.close();
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

As you can see that I am rolling back the transaction if any SQL exception comes. If the batch processing is successful, I am explicitly committing the transaction.

That's all for JDBC Batch Processing feature, make sure to experiment with your data to get the optimal value of batch size for bulk queries. One of the limitation of batch processing is that we can't execute different type of queries in the batch.

4. JDBC DataSource and Apache DBCP

We have already seen that JDBC DriverManager can be used to get relational database connections. But when it comes to actual programming, we want more than just connections.

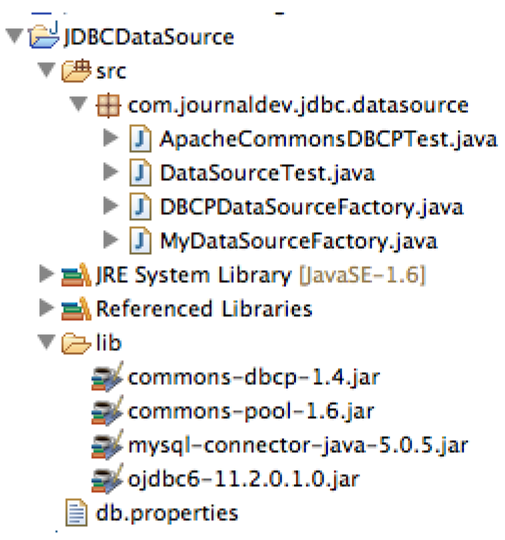
Most of the times we are looking for loose coupling for connectivity so that we can switch databases easily, connection pooling for transaction management and distributed systems support. **JDBC DataSource** is the preferred approach if you are looking for any of these features in your application. JDBC DataSource interface is present in javax.sql package and it only declare two overloaded methods getConnection() and getConnection(String str1,String str2).

It is the responsibility of different Database vendors to provide different kinds of implementation of DataSource interface. For example MySQL JDBC Driver provides basic implementation of DataSource interface with com.mysql.jdbc.jdbc2.optional.MysqlDataSource class and Oracle database driver implements it with oracle.jdbc.pool.OracleDataSource class.

These implementation classes provide methods through which we can provide database server details with user credentials. Some of the other common features provided by these DataSource implementation classes are

- Caching of PreparedStatement for faster processing
- Connection timeout settings
- Logging features
- ResultSet maximum size threshold

Let's create a simple JDBC project and learn how to use MySQL and Oracle DataSource basic implementation classes to get the database connection. Our final project will look like below image.



```
JDBCDataSource
  src
    com.journaldev.jdbc.datasource
      ApacheCommonsDBCPTest.java
      DataSourceTest.java
      DBCPDataSourceFactory.java
      MyDataSourceFactory.java
  JRE System Library [JavaSE-1.6]
  Referenced Libraries
  lib
    commons-dbcp-1.4.jar
    commons-pool-1.6.jar
    mysql-connector-java-5.0.5.jar
    ojdbc6-11.2.0.1.0.jar
  db.properties
```

A. Database Setup

Before we get into our example programs, we need some database setup with table and sample data. Installation of MySQL or Oracle database is out of scope of this tutorial, so I will just go ahead and setup table with sample data.

```
--Create Employee table
CREATE TABLE `Employee` (
  `empId` int(10) unsigned NOT NULL,
  `name` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`empId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- insert some sample data
INSERT INTO `Employee` (`empId`, `name`)
VALUES
    (1, 'Pankaj'),
    (2, 'David');

commit;

CREATE TABLE "EMPLOYEE"
  (
    "EMPID"   NUMBER NOT NULL ENABLE,
    "NAME"    VARCHAR2(10 BYTE) DEFAULT NULL,
    PRIMARY KEY ("EMPID")
  );
 Insert into EMPLOYEE (EMPID,NAME) values (10,'Pankaj');
Insert into EMPLOYEE (EMPID,NAME) values (5,'Kumar');
Insert into EMPLOYEE (EMPID,NAME) values (1,'Pankaj');
commit;
```

Now let’s move on to our java programs. For having database configuration loosely coupled, I will read them from property file.

```
#mysql DB properties
MYSQL_DB_DRIVER_CLASS=com.mysql.jdbc.Driver
MYSQL_DB_URL=jdbc:mysql://localhost:3306/UserDB
MYSQL_DB_USERNAME=pankaj
MYSQL_DB_PASSWORD=pankaj123

#Oracle DB Properties
ORACLE_DB_DRIVER_CLASS=oracle.jdbc.driver.OracleDriver
ORACLE_DB_URL=jdbc:oracle:thin:@localhost:1521:orcl
ORACLE_DB_USERNAME=hr
ORACLE_DB_PASSWORD=oracle
```

Make sure that above configurations match with your local setup. Also make sure you have MySQL and Oracle DB JDBC jars included in the build path of the project.

B. JDBC MySQL and Oracle DataSource Example

Let’s write a factory class that we can use to get MySQL or Oracle DataSource.

```
package com.journaldev.jdbc.datasource;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.SQLException;
import java.util.Properties;

import javax.sql.DataSource;

import oracle.jdbc.pool.OracleDataSource;

import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;

public class MyDataSourceFactory {

    public static DataSource getMySQLDataSource() {
        Properties props = new Properties();
        FileInputStream fis = null;
        MysqlDataSource mysqlDS = null;
        try {
            fis = new FileInputStream("db.properties");
            props.load(fis);
            mysqlDS = new MysqlDataSource();
            mysqlDS.setURL(props.getProperty("MYSQL_DB_URL"));
            mysqlDS.setUser(props.getProperty("MYSQL_DB_USERNAME"));

mysqlDS.setPassword(props.getProperty("MYSQL_DB_PASSWORD"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return mysqlDS;
    }

    public static DataSource getOracleDataSource(){
        Properties props = new Properties();
        FileInputStream fis = null;
        OracleDataSource oracleDS = null;
        try {
            fis = new FileInputStream("db.properties");
            props.load(fis);
            oracleDS = new OracleDataSource();
            oracleDS.setURL(props.getProperty("ORACLE_DB_URL"));
            oracleDS.setUser(props.getProperty("ORACLE_DB_USERNAME"));

oracleDS.setPassword(props.getProperty("ORACLE_DB_PASSWORD"));
        } catch (IOException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return oracleDS;
    }

}
```

Notice that both Oracle and MySQL DataSource implementation classes are very similar, let’s write a simple test program to use these methods and run some test.

```
package com.journaldev.jdbc.datasource;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.sql.DataSource;

public class DataSourceTest {

    public static void main(String[] args) {
        testDataSource("mysql");
        System.out.println("**********");
        testDataSource("oracle");

    }

    private static void testDataSource(String dbType) {
        DataSource ds = null;
        if("mysql".equals(dbType)){
            ds = MyDataSourceFactory.getMySQLDataSource();
        }else if("oracle".equals(dbType)){
            ds = MyDataSourceFactory.getOracleDataSource();
        }else{
            System.out.println("invalid db type");
            return;
        }

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            con = ds.getConnection();
            stmt = con.createStatement();
            rs = stmt.executeQuery("select empid, name from Employee");
            while(rs.next()){
                System.out.println("Employee ID="+rs.getInt("empid")+",
Name="+rs.getString("name"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
                try {
                    if(rs != null) rs.close();
                    if(stmt != null) stmt.close();
                    if(con != null) con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
        }
    }

}
```

Notice that the client class is totally independent of any Database specific classes. This helps us in hiding the underlying implementation details from client program and achieve loose coupling and abstraction benefits.

When we run above test program, we will get below output.

```
Employee ID=1, Name=Pankaj
Employee ID=2, Name=David
**********
Employee ID=10, Name=Pankaj
Employee ID=5, Name=Kumar
Employee ID=1, Name=Pankaj
```

C. Apache Commons DBCP Example

If you look at above DataSource factory class, there are two major issues with it.

1. The factory class methods to create the MySQL and Oracle DataSource are tightly coupled with respective driver API. If we want to remove support for Oracle database in future or want to add some other database support, it will require code change.
2. Most of the code to get the MySQL and Oracle DataSource is similar, the only different is the implementation class that we are using.

Apache Commons DBCP API helps us in getting rid of these issues by providing DataSource implementation that works as an abstraction layer between our program and different JDBC drivers.

Apache DBCP library depends on Commons Pool library, so make sure they both are in the build path as shown in the image.

Here is the DataSource factory class using BasicDataSource that is the simple implementation of DataSource.

```java
package com.journaldev.jdbc.datasource;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

import javax.sql.DataSource;

import org.apache.commons.dbcp.BasicDataSource;

public class DBCPDataSourceFactory {

    public static DataSource getDataSource(String dbType){
        Properties props = new Properties();
        FileInputStream fis = null;
        BasicDataSource ds = new BasicDataSource();
        try {
            fis = new FileInputStream("db.properties");
            props.load(fis);
        }catch(IOException e){
            e.printStackTrace();
            return null;
        }
        if("mysql".equals(dbType)){

ds.setDriverClassName(props.getProperty("MYSQL_DB_DRIVER_CLASS"));
            ds.setUrl(props.getProperty("MYSQL_DB_URL"));
            ds.setUsername(props.getProperty("MYSQL_DB_USERNAME"));
            ds.setPassword(props.getProperty("MYSQL_DB_PASSWORD"));
        }else if("oracle".equals(dbType)){

ds.setDriverClassName(props.getProperty("ORACLE_DB_DRIVER_CLASS"));
            ds.setUrl(props.getProperty("ORACLE_DB_URL"));
            ds.setUsername(props.getProperty("ORACLE_DB_USERNAME"));
            ds.setPassword(props.getProperty("ORACLE_DB_PASSWORD"));
        }else{
            return null;
        }

        return ds;
    }
}
```

As you can see that depending on user input, either MySQL or Oracle DataSource is created. If you are supporting only one database in the application then you don’t even need these logic. Just change the properties and you can switch from one database server to another. The key point through which Apache DBCP provide abstraction is *setDriverClassName()* method.

Here is the client program using above factory method to get different types of connection.

```
package com.journaldev.jdbc.datasource;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.sql.DataSource;

public class ApacheCommonsDBCPTest {

    public static void main(String[] args) {
        testDBCPDataSource("mysql");
        System.out.println("**********");
        testDBCPDataSource("oracle");
    }

    private static void testDBCPDataSource(String dbType) {
        DataSource ds = DBCPDataSourceFactory.getDataSource(dbType);

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            con = ds.getConnection();
            stmt = con.createStatement();
            rs = stmt.executeQuery("select empid, name from Employee");
            while(rs.next()){
                System.out.println("Employee ID="+rs.getInt("empid")+",
Name="+rs.getString("name"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally{
                try {
                    if(rs != null) rs.close();
                    if(stmt != null) stmt.close();
                    if(con != null) con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
        }
    }

}
```

When you run above program, the output will be same as earlier program.

If you look at the DataSource and above usage, it can be done with normal DriverManager too. The major benefit of DataSource is when it's used within a Context and with JNDI.

With simple configurations we can create a Database Connection Pool that is maintained by the Container itself. Most of the servlet containers such as Tomcat and JBoss provide its own DataSource implementation and all we need is to configure it through simple XML based configurations and then use JNDI context lookup to get the DataSource and work with it. This helps us by taking care of connection pooling and management from our application side to server side and thus giving more time to write business logic for the application.

5. JDBC Transaction Management and Savepoint

In the JDBC Tutorial we learned how we can use JDBC API for database connectivity and execute SQL queries. We also looked at the different kind of drivers and how we can write loosely couple JDBC programs that helps us in switching from one database server to another easily.

This tutorial is aimed to provide details about **JDBC Transaction Management** and using **JDBC Savepoint** for partial rollback.

By default when we create a database connection, it runs in **auto-commit** mode. It means that whenever we execute a query and it's completed, the commit is fired automatically. So every SQL query we fire is a transaction and if we are running some DML or DDL queries, the changes are getting saved into database after every SQL statement finishes.

Sometimes we want a group of SQL queries to be part of a transaction so that we can commit them when all the queries runs fine and if we get any exception, we have a choice of rollback all the queries executed as part of the transaction.

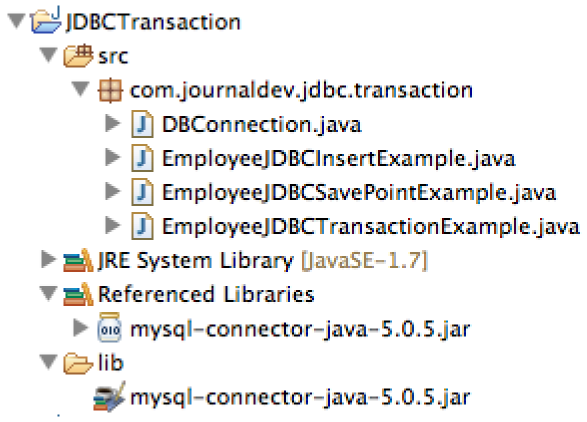
Let's understand with a simple example where we want to utilize JDBC transaction management support for data integrity. Let's say we have UserDB database and Employee information is saved into two tables. For my example, I am using MySQL database but it will run fine on other relational databases as well such as Oracle and PostgreSQL.

The tables store employee information with address details in tables, DDL scripts of these tables are like below.

```
CREATE TABLE `Employee` (
  `empId` int(11) unsigned NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`empId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `Address` (
  `empId` int(11) unsigned NOT NULL,
  `address` varchar(20) DEFAULT NULL,
  `city` varchar(5) DEFAULT NULL,
  `country` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`empId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Our final project looks like below image, we will look into each of the classes one by one.


JDBCTransaction
src
com.journaldev.jdbc.transaction
DBConnection.java
EmployeeJDBCInsertExample.java
EmployeeJDBCSavePointExample.java
EmployeeJDBCTransactionExample.java
JRE System Library [JavaSE-1.7]
Referenced Libraries
mysql-connector-java-5.0.5.jar
lib
mysql-connector-java-5.0.5.jar

As you can see that I have MySQL JDBC jar in the project build path, so that we can connect to the MySQL database.

```
package com.journaldev.jdbc.transaction;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnection {

    public final static String DB_DRIVER_CLASS =
"com.mysql.jdbc.Driver";
    public final static String DB_URL =
"jdbc:mysql://localhost:3306/UserDB";
    public final static String DB_USERNAME = "pankaj";
    public final static String DB_PASSWORD = "pankaj123";
    public static Connection getConnection() throws
ClassNotFoundException, SQLException {

        Connection con = null;

        // load the Driver Class
        Class.forName(DB_DRIVER_CLASS);

        // create the connection now
        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

        System.out.println("DB Connection created successfully");
        return con;
    }
}
```

DBConnection is the class where we are creating MySQL database connection to be used by other classes.

```
package com.journaldev.jdbc.transaction;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class EmployeeJDBCInsertExample {

    public static final String INSERT_EMPLOYEE_QUERY = "insert into
Employee (empId, name) values (?,?)";

    public static final String INSERT_ADDRESS_QUERY = "insert into
Address (empId, address, city, country) values (?,?,?,?)";

    public static void main(String[] args) {

        Connection con = null;
        try {
            con = DBConnection.getConnection();

            insertEmployeeData(con, 1, "Pankaj");

            insertAddressData(con, 1, "Albany Dr", "San Jose", "USA");
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        } finally {

            try {
                if (con != null)
                    con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public static void insertAddressData(Connection con, int id,
            String address, String city, String country) throws
SQLException {
        PreparedStatement stmt =
con.prepareStatement(INSERT_ADDRESS_QUERY);
        stmt.setInt(1, id);
        stmt.setString(2, address);
        stmt.setString(3, city);
        stmt.setString(4, country);

        stmt.executeUpdate();

        System.out.println("Address Data inserted successfully for ID="
+ id);
        stmt.close();
    }

    public static void insertEmployeeData(Connection con, int id,
String name)
            throws SQLException {
        PreparedStatement stmt =
con.prepareStatement(INSERT_EMPLOYEE_QUERY);
        stmt.setInt(1, id);
        stmt.setString(2, name);

        stmt.executeUpdate();

        System.out.println("Employee Data inserted successfully for
ID=" + id);
        stmt.close();
    }

}
```

This is a simple JDBC program where we are inserting user provided values in both Employee and Address tables created above. Now when we will run this program, we will get following output.

```
DB Connection created successfully
Employee Data inserted successfully for ID=1
com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too long for
column 'city' at row 1
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2939)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
    at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java
:1268)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
541)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
455)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
440)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCInsertExample.insertAddress
Data(EmployeeJDBCInsertExample.java:45)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCInsertExample.main(Employee
JDBCInsertExample.java:23)
```

As you can see that SQLException is raised when we are trying to insert data into Address table because the value is bigger than the size of the column.

If you will look at the content of the Employee and Address tables, you will notice that data is present in Employee table but not in Address table. This becomes a serious problem because only part of the data is inserted properly and if we run the program again, it will try to insert into Employee table again and throw below exception.

```
com.mysql.jdbc.exceptions.MySQLIntegrityConstraintViolationException:
Duplicate entry '1' for key 'PRIMARY'
    at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:931)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2941)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
    at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java
:1268)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
541)
```

```
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
455)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
440)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCInsertExample.insertEmploye
eData(EmployeeJDBCInsertExample.java:57)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCInsertExample.main(Employee
JDBCInsertExample.java:21)
```

So there is no way we can save the data into Address table now for the Employee. So this program leads to data integrity issues and that’s why we need transaction management to insert into both the tables successfully or rollback everything if any exception arises.

A. JDBC Transaction Management

JDBC API provide method setAutoCommit() through which we can disable the auto commit feature of the connection. We should disable auto commit only when it’s required because the transaction will not be committed unless we call the commit() method on connection. Database servers’ uses table locks to achieve transaction management and its resource intensive process. So we should commit the transaction as soon as we are done with it. Let’s write another program where we will use JDBC transaction management feature to make sure data integrity is not violated.

```
package com.journaldev.jdbc.transaction;

import java.sql.Connection;
import java.sql.SQLException;

public class EmployeeJDBCTransactionExample {

    public static void main(String[] args) {

        Connection con = null;
        try {
            con = DBConnection.getConnection();

            //set auto commit to false
            con.setAutoCommit(false);

            EmployeeJDBCInsertExample.insertEmployeeData(con, 1,
"Pankaj");
```

```
            EmployeeJDBCInsertExample.insertAddressData(con, 1, "Albany
Dr", "San Jose", "USA");

            //now commit transaction
            con.commit();

        } catch (SQLException e) {
            e.printStackTrace();
            try {
                con.rollback();
                System.out.println("JDBC Transaction rolled back
successfully");
            } catch (SQLException e1) {
                System.out.println("SQLException in
rollback"+e.getMessage());
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Please make sure you remove the earlier inserted data before running this program. When you will run this program, you will get following output.

```
DB Connection created successfully
Employee Data inserted successfully for ID=1
com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too long for
column 'city' at row 1
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2939)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
    at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java
:1268)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
541)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
455)
```

```
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
440)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCInsertExample.insertAddress
Data(EmployeeJDBCInsertExample.java:45)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCTransactionExample.main(Emp
loyeeJDBCTransactionExample.java:19)
JDBC Transaction rolled back successfully
```

The output is similar to previous program but if you will look into the database tables, you will notice that data is not inserted into Employee table. Now we can change the city value so that it can fit in the column and rerun the program to insert data into both the tables. Notice that connection is committed only when both the inserts executed fine and if any of them throws exception, we are rolling back complete transaction.

B. JDBC Savepoint Example

Sometimes a transaction can be group of multiple statements and we would like to rollback to a particular point in the transaction. JDBC Savepoint helps us in creating checkpoints in a transaction and we can rollback to that particular checkpoint. Any savepoint created for a transaction is automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

Let’s say we have a Logs table where we want to log the messages that employee information is saved successfully. But since it’s just for logging, if there are any exceptions while inserting into Logs table, we don’t want to rollback the entire transaction. Let’s see how we can achieve this with JDBC savepoint.

```
CREATE TABLE `Logs` (
  `id` int(3) unsigned NOT NULL AUTO_INCREMENT,
  `message` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
package com.journaldev.jdbc.transaction;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Savepoint;

public class EmployeeJDBCSavePointExample {

    public static final String INSERT_LOGS_QUERY = "insert into Logs
(message) values (?)";

    public static void main(String[] args) {

        Connection con = null;
        Savepoint savepoint = null;
        try {
            con = DBConnection.getConnection();

            // set auto commit to false
            con.setAutoCommit(false);

            EmployeeJDBCInsertExample.insertEmployeeData(con, 2,
"Pankaj");

            EmployeeJDBCInsertExample.insertAddressData(con, 2, "Albany
Dr",
                    "SFO", "USA");

            // if code reached here, means main work is done
successfully
            savepoint = con.setSavepoint("EmployeeSavePoint");

            insertLogData(con, 2);

            // now commit transaction
            con.commit();

        } catch (SQLException e) {
            e.printStackTrace();
            try {
                if (savepoint == null) {
                    // SQLException occurred in saving into Employee or
Address tables
                    con.rollback();
                    System.out
                            .println("JDBC Transaction rolled back
successfully");
                } else {
                    // exception occurred in inserting into Logs table
                    // we can ignore it by rollback to the savepoint
                    con.rollback(savepoint);
                    //lets commit now
                    con.commit();
                }
            } catch (SQLException e1) {
                System.out.println("SQLException in rollback" +
e.getMessage());
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    private static void insertLogData(Connection con, int i)
            throws SQLException {
        PreparedStatement stmt =
con.prepareStatement(INSERT_LOGS_QUERY);

        //message is very long, will throw SQLException
        stmt.setString(1, "Employee information saved successfully for
ID" + i);

        stmt.executeUpdate();
        System.out.println("Logs Data inserted successfully for ID=" +
i);

        stmt.close();
    }

}
```

The program is very simple to understand. As you can see that I am creating the savepoint after data is inserted successfully into Employee and Address tables. If SQLException arises and savepoint is null, it means that exception is raised while executing insert queries for either Employee or Address table and hence I am rolling back complete transaction.

If savepoint is not null, it means that SQLException is coming in inserting data into Logs table, so I am rolling back transaction only to the savepoint and committing it.

If you will run above program, you will see below output.

```
DB Connection created successfully
Employee Data inserted successfully for ID=2
Address Data inserted successfully for ID=2
com.mysql.jdbc.MysqlDataTruncation: Data truncation: Data too long for
column 'message' at row 1
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2939)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:3249)
    at
com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java
:1268)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
541)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
455)
    at
com.mysql.jdbc.PreparedStatement.executeUpdate(PreparedStatement.java:1
440)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCSavePointExample.insertLogD
ata(EmployeeJDBCSavePointExample.java:73)
    at
com.journaldev.jdbc.transaction.EmployeeJDBCSavePointExample.main(Emplo
yeeJDBCSavePointExample.java:30)
```

If you will check database tables, you will notice that the data is inserted successfully in Employee and Address tables. Note that we could have achieved this easily by committing the transaction when data is inserted successfully in Employee and Address tables and used another transaction for inserting into logs table. This is just an example to show the usage of JDBC savepoint in java programs.

You can download the source code of "JDBC Transaction Management Example Project" from here.

Download project from above link and play around with it, try to use multiple savepoints and JDBC transactions API to learn more about it.

Copyright Notice

References

1. http://www.journaldev.com/2471/jdbc-example-tutorial-drivers-connection-statement-resultset
2. http://www.journaldev.com/2489/jdbc-statement-vs-preparedstatement-sql-injection-example
3. http://www.journaldev.com/2494/jdbc-batch-processing-example-tutorial-with-insert-statements
4. http://www.journaldev.com/2509/jdbc-datasource-example-oracle-mysql-and-apache-dbcp-tutorial
5. http://www.journaldev.com/2483/jdbc-transaction-management-and-savepoint-example-tutorial