Definition and an example

JDBC

Java Database Connectivity

By Hamid Mosavi-Porasl

1 What is JDBC

JDBC stands for Java Database Connectivity. What JDBC does is "sending" SQL statements to DBMS.

1.1 Loading the driver

class.forName("jdbc.DriverXYZ")

We do not need to create an instance of a driver and register it with the *DriverManager* because calling *class.forName* will do that for us automatically.

1.2 Making the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS.

Connection con =
DriverManager.getConnection(url,"myLogin","myPassword");

If we are using the JDBC-ODBC Bridge driver, the JDBC URL will start with Jdbc:odbc:

String url = "jdbc:odbc:fred";
Connection con = DriverManager.getConnection(url,"Fernanda", "J8");

DriverManager.getConnection is an open connection we can use to create JDBC statements that pass our SQL statement to the DBMS.

Setting up Tables

COF_NAME	**SUP_ID**	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colomb_Decaf	101	8.99	0	0

SUP_ID	SUP_NAME	STREET	CITY	STATE
101	Acme, inc	99 Market stree	Groundsville	CA
49	Supperior	Party place	Mendocino	CA
150	High ground	100 Coffee	Meadows	CA

SQL code:

```
CREATE TABLE COFFEES
  (COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER)
```

Oracle uses a semicolon (;) to indicate the end of a statement, and **Sybase** uses the word *go*.

```
String createTableCoffeess = "CREATE TABLE COFFEES"+
  "(COF_NAME VARCHA(32), SUP_ID INTEGER,"+
              "PRICE FLOAT, SALES INTEGER, TOTAL INTEGER)";
```

1.3 Creating JDBC statements

A *statement* object Stmt.executeUpdate(*createTableCoffeess);* sends our SQL statement to the DBMS.

We supply create a statement object and then execute it.
For a *SELECT* statement the method to use is *executeQuery*.

For statement that creates or modifies tables the method to use is *ExecuteUpdate.*

It takes an instance of an active ***connection to create a Statement*** object. In the following example, we use our connection to create the Statement object *stmt*:

```
Statement stmt = con.crateStatement();

Stmt.executeUodate("CREATE TABLE COFFEES"+
  "(COF_NAME VARCHA(32), SUP_ID INTEGER,"+
              "PRICE FLOAT, SALES INTEGER, TOTAL INTEGER)");
```

or

createTableCoffeess is a DDL (data definition language)

The method used most often for executing SQL statements is *executeQuery*.

1.4 Entering Data into a Table

```
Statement stmt = con.createStatement();
Stmt.executeUpdate("INSERT INTO COFFEES"+
          "VALUES (' Espresso',150, 9.99, 0, 0)");
Stmt.executeUpdate("INSERT INTO COFFEES"+
          "VALUES (' Colombian_Decaf',101, 8.99, 0, 0)");
```

1.5 Getting Data from a table (SQL code)

*SELECT * FROM COFFEES*

```
COF_NAME          SUP_ID      PRICE       SALES
  TOTAL
-----------------  ----------  ----------  ----------
COLOMBIAN         101         7.99        0
FRENCH_ROAST      49          8.99        0
Espresso       150 9.99        0           0
……
```

SELECT COF_NAME, PRICE FROM COFFES

```
_COF_NAME         PRICE
-----------------  ---------
COLOMBIAN         7.99
FRENCH_ROAST      8.99
Espresso          9.99
……
```

SELECT COF_NAME, PRICE FROM COFFES WHERE PRICE < 9.00

The results would look similar to this

COF_NAME	PRIS
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

1.6 Retrieving Values from Result Sets

JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results.

```
ResultSet rs =
stmt.executedQuery("SELECT CONF_NAME, PRICE FROM COFFEES");
```

1.6.1 Using the method next

In Order to access the names and prices, we will to go to each row and retrieve the values according to their types. The method ***next*** moves what is called a cursor to the next row and makes that row (called current row) the one upon which we can operate.

1.6.2 Using method getxxx

We use the getxxx of the appropriate type to retrieve the value in each column. For example, the first column in each row of *rs* is *CONF_NAME,* which stores a value of SQL type *charvar*. The method for retrieving a value of SQL type *varchar* is *getString*. The second column in each row stores a value of SQL type *FLOAT,* and the method for retrieving a value of that type is *getFloat*.

```
String query =” SELECT COF_NAME, PRIS FROM COFFEE”;
ResultSet rs = stmt.executeQuery(query);
  While(rs.next()){
  String  s = rs.getString(“CONF_NAME”);
  Float n = rs.getFloat(“PRIS”);
  System.out.println(s+”    “+n);
  }
```

JDBC offers two ways to identify the column from which a getxxx method gets a value. Oneway is to give the column name, as was done in the example above. The second way is to give the column index (number of the column), with 1 signifying the first column, 2, the second, and so on. Using the column number

```
  String s = rs.getString(1);
  Float n = rs.getFloat(2);
```

1.7 Updating Tables

```
String updateString = “UPDATE COFFEES “+
   SET SALES = 75  WHERE CONF_NAME LIKE ´Colombian´”;

Stmt.executeUpdate(updateString);
```

The table COFFEES will now look like this:

CONF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0

Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

```
String query = “SELECT CONF_NAME, SALES FROM COFFEES”+
         “WHERE CONF_NAME LIKE ‘Colombian’”;

ResultSet rs = stmt.executeQuery(query);

While (rs.next){
String s = rs.getString(“CONF_NAME”);
                              int n = rs.getInt(“SALES”);
                  System.out.println(n+” pounds of “ +s+ “ Sold this week.”);
}
```

Output:
75 pounds of Colombian sold this week.

Update the TOTAL column by adding a weekly amount sold to the existing total, and then let’s print out the number of pounds sold to date:

```
String updateString =” UPDATE COFFEES ”+
             “SET TOTAL = TOTAL + 75 ” +
             WHERE COF_NAME LIKE ‘Colombian’”;

stmt.execueteUpdate(updateString);

String query =”SELECT COF_NAME, TOTAL FROM COF_NAMES ”+
  WHERE COF_NAME LIKE ‘Colombian’”;

ResultSet rs = stmt.executeQuery(query);

While(rs.next()){
  String s = rs.getString(1);
  int n = rs.getInt(2);
  System.out.println(n + “ pounds of  “+  s + “ sold to date.”);
  }
```

1.8 Using Prepared Statements

Sometimes it is more convenient or more efficient to use PreparedStatement object for sending SQL statements to the database. Preparedstatements slows us to develop an SQL query template that we can reuse to handle similar requests with different values between each execution. Essentially we create the query, which can be any sort of SQL statement, leaving any variable values undefined. We can then specify values for our undefined elements before executing the query, and repeat as necessary. Prepared statements are created from a Connection object, just like regular Statement objects. In the SQL, replace any variable values with a question mark.

1.8.1 Creating a PreparedStatement Object

```
String query=
   ”SELECT * FROM GAME_RECORDERS WHERE SCORE > ? AND TEAM=?”;
PreparedStatement statement = connection.preparedStatement(query);
```

1.8.2 Supplying values for PreparedStatements Object

Before we can execute the statement we must specify a value for all of our missing parameters. The PreparedStatement object supports a number of methods, each tied to setting a value of a specific type – int, long, string, and so forth. Each method takes two arguments, an index value indicating which missing parameter you are specifying, and the value itself. The first parameter has an index value of 1 (not 0) so to specify a query that selects all high scores <10,000 for the “Gold” team we use the following statements to set the values and execute the query:

```
Statement.setInt(1,10000);       //Score
Statement.setString(2,”Gold”);   //Team
ResultSet results = statement.execute();
```

Once you have defined a prepared statement you can reuse it simply by changing parameters, as needed. There is no need to create a new prepared statement instance as long as the basic query is unchanged. So, we can execute several queries without having to create a statement object. We can even share a single prepared statement among an application’s components or a servlet’s users. When using prepared statements, the RDBMS engine has to parse the SQL statement only once, rather than again and again with each new request. This results in more efficient database operations.

Not only is this more efficient in terms of database access, object creation, and memory allocation but the resulting code is cleaner and more easily understood.

Consider this example again, but this time the queries are not hard coded, but come from a Bean, userBean, which has been initialized from an input form.

```
Statement.setInt(1,useBean.getScore()); //Score
Statement.setString(2,userBean.getTeam(); //Team
ResultSet results = statement.execute();
```

The alternative is to build each SQL statement from strings, which can quickly get confusing, especially with complex queries. Consider the following example, this time without the benefit of a prepared statement:

```
Statement statement = connection.getStatement();
String query=”SELECT * FROM GAME_RECORDS WHERE
SCORE>”+ userBean.getScore()+ “AND TEAM =’”+user.getTeam()+
UserBean.getTeam()+” ‘”;
ResultSet results = Statement.executeQuery(query);
```

1.9 Using Joins

Sometimes you need to use two or more tables to get the data you want. For example, suppose the proprietor of the Coffee Break wants a list of the coffees he buys from Acme, inc.

1.10 Create the table SUPPLIERS

```
String createSUPPLIERS = “ create table SUPPLERS “+
        “(SUP_ID INTEGER, “+
                        “SUP_NAME VARCHAR(40),”+
                        “STREET VARCHAR(40),
                        “CITY VARCHAR(20),”+
                        “STATE VARCHAR(20),”+
                        “ZIP CHAR(5))”
stmt.executeUpdate(createSUPPLIERS);
```

Following statements inserts rows for three suppliers into SUPPLIERS

```
stmt.executeUpdate(“insert into SUPPLIERS values(101, “+
        “’Acme, Inc.’, ’99 Market Street’, ‘Grundsville’,”+
        “’CA’”);

stmt.executeUpdate(“insert into SUPPLIERS values(49, “+
        “’Superior Coffee’, ’1 Party Place’, ‘Mendocino’,”+
        “’CA’”);

stmt.executeUpdate(“insert into SUPPLIERS values(150, “+
        “’The High Ground’, ’100 Coffee Lane’, ‘Meadows’,”+
        “’CA’”);

ResultSet rs = stmt.executeQuery(“select * from SUPPLIERS”);
```

The result set will look similar to this:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Grundsville	CA	123
49	Superior Coffee1	Party Place	Mendocin	CA	123
150	The High Ground 100	Coffee Lane	Meadows	CA	123

```
String query =”
SELECT COFFEES.COF_NAME “+
“FROM COFFEES, SUPPLIERS “+
“WHERE SUPPLIERS.SUP_NAME LIKE ‘Acme, Inc. ”+
“and SUPPLIERS.SUP_ID = COFFEES.SUP_ID”;

ResultSet rs = stmt.executeQuery(query);

System.out.println(“Coffees bought from Acme, Inc.:”);
While (rs.next()){
  String coffeeName = getString(“COF_NAME”);
  System.out.println(“            “+ coffeeName);
```

This will produce the following output:

```
Coffee bought from Acme, Inc.:
  Colombian
  Colombian_Decaf
```

1.11 Create Statements for Creating a Stored Procedure

```
create procedure SHOW_SUPPLIERS
as
select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME
from SUPPLIERS.SUP_ID = COFFEES.SUP_ID
order by SUP_NAME
```

The following code puts the SQL statement into a string and assigns it to the variable createProcedure, which we will use later:

```
        String createProcedure =” create procedure SHOW_SUPPLIERS “+
“as  “+
“select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME “+
“from SUPPLIERS.SUP_ID = COFFEES.SUP_ID “+
“order by SUP_NAME”;

Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();

The ResultSet rs will be similar to the following:

SUP_NAME	**COF_NAME**
Acme, Inc.	Colombian
Acme, Inc.	Colombian_Decaf
Superior Coffee	French_Roast
Superior Coffee	French_Roast_Decaf
The Height Ground	Espresso