[JavaSE实验] JDBC and MySQL

1 JDBC

JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. From a technical point of view, the API is as a set of classes in the java.sql package. To use JDBC with a particular database, we need a JDBC driver for that database.

2 About MySQL database

MySQL is a leading open source database management system. It is a multi user, multithreaded database management system. MySQL is especially popular on the web. It is one of the parts of the very popular LAMP platform. Linux, Apache, MySQL, PHP. Currently MySQL is owned by Oracle. MySQL database is available on most important OS platforms. It runs under BSD Unix, Linux, Windows or Mac. Wikipedia and YouTube use MySQL. These sites manage millions of queries each day. MySQL comes in two versions. MySQL server system and MySQL embedded system.

3 Before we start

For this tutorial, we need to have several libraries installed. We need to install mysql-server and mysql-client packages. The first package has the MySQL server and the second one contains, among others, the mysql monitor tool. We need to install the JDK, Java Development Kit, for compiling and running Java programs. Finally, we need the MySQL Connector/J driver. If you are using Netbeans IDE, than you have already the driver at hand. Inside the Projects tab, right click on the Libraries node and select Add Library option. From the list of options, select MySQL JDBC Driver.

4 Installation and Testing

If you don't already have MySQL installed, we must install it.

```
$sudo apt-get install mysql-server
```

This command installs the MySQL server and various other packages. While installing the package, we are prompted to enter a password for the MySQL root account.

Next, we are going to create a new database user and a new database. We use the mysql client.

```
$service mysql status
mysql start/running, process 1238
```

We check if the MySQL server is running. If not, we need to start the server. On Ubuntu Linux, this can be done with the service mysql start command.

```
$mysql -u root -p

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.1.66-0ubuntu0.10.04.1 (Ubuntu)

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
+--------------------+
2 rows in set (0.00 sec)
```

We use the mysql monitor client application to connect to the server. We connect to the database using the root account. We show all available databases with the SHOW DATABASES statement.

```
mysql> CREATE DATABASE testdb;
Query OK, 1 row affected (0.02 sec)
We create a new testdb database. We will use this database throughout the tutorial.

mysql> CREATE USER 'xiaodong'@'localhost' IDENTIFIED BY 'xiaodong';
Query OK, 0 rows affected (0.00 sec)

mysql> USE testdb;
Database changed
```

```
mysql> GRANT ALL ON testdb.* TO 'xiaodong'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> quit;
Bye
```

5 MySQL version

If the following program runs OK, then we have everything installed OK. We check the version of the MySQL server.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Version {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;
        ResultSet rs = null;

        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "testuser";
        String password = "test623";

        try {
            con = DriverManager.getConnection(url, user, password);
            st = con.createStatement();
            rs = st.executeQuery("SELECT VERSION()");

            if (rs.next()) {
                System.out.println(rs.getString(1));
            }
```

```
        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Version.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } finally {
            try {
                if (rs != null) {
                    rs.close();
                }
                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(Version.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We connect to the database and get some info about the MySQL server.

```
String url = "jdbc:mysql://localhost:3306/testdb";
```

This is the connection url for the MySQL database. Each driver has a different syntax for the url. In our case, we provide a host, a port and a database name.

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

We establish a connection to the database, using the connection url, user name and password.

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

The createStatement() method of the connection object creates a Statement object for sending SQL statements to the database.

```
rs = st.executeQuery("SELECT VERSION()");
```

The createStatement() method of the connection object executes the given SQL statement, which returns

a single ResultSet object. The ResultSet is a table of data returned by a specific SQL statement.

```
if (result.next()) {
    System.out.println(result.getString(1));
}
```

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next() method moves the cursor to the next row. If there are no rows left, the method returns false. The getString() method retrieves the value of a specified column. The first column has index 1.

```
} catch (SQLException ex) {
    Logger lgr = Logger.getLogger(Version.class.getName());
    lgr.log(Level.SEVERE, ex.getMessage(), ex);

}
```

In case of an exception, we log the error message. For this console example, the message is displayed in the terminal.

```
try {
    if (rs != null) {
        rs.close();
    }
    if (st != null) {
        st.close();
    }
    if (con != null) {
        con.close();
    }
...
```

Inside the finally block, we close the database resources. We also check if the objects are not equal to null. This is to prevent null pointer exceptions. Otherwise we might get a NullPointerException, which would terminate the application and leave the resources not cleaned up.

```
} catch (SQLException ex) {
    Logger lgr = Logger.getLogger(Version.class.getName());
    lgr.log(Level.WARNING, ex.getMessage(), ex);
}
```

We log an error message, when the resources could not be closed.

```
java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/Version
5.5.9
```

This is the output of the program on my system.

6 Creating and populating tables

Next we are going to create database tables and fill them with data. These tables will be used throughout this tutorial.

```
DROP TABLE IF EXISTS Books, Authors, Testing, Images;

CREATE TABLE IF NOT EXISTS Authors(Id INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(25)) ENGINE=InnoDB;
CREATE TABLE IF NOT EXISTS Books(Id INT PRIMARY KEY AUTO_INCREMENT,
    AuthorId INT, Title VARCHAR(100),
    FOREIGN KEY(AuthorId) REFERENCES Authors(Id) ON DELETE CASCADE)
    ENGINE=InnoDB;
CREATE TABLE IF NOT EXISTS Testing(Id INT) ENGINE=InnoDB;
CREATE TABLE IF NOT EXISTS Images(Id INT PRIMARY KEY AUTO_INCREMENT,
    Data MEDIUMBLOB);

INSERT INTO Authors(Id, Name) VALUES(1, 'Jack London');
INSERT INTO Authors(Id, Name) VALUES(2, 'Honore de Balzac');
INSERT INTO Authors(Id, Name) VALUES(3, 'Lion Feuchtwanger');
INSERT INTO Authors(Id, Name) VALUES(4, 'Emile Zola');
INSERT INTO Authors(Id, Name) VALUES(5, 'Truman Capote');

INSERT INTO Books(Id, AuthorId, Title) VALUES(1, 1, 'Call of the Wild');
INSERT INTO Books(Id, AuthorId, Title) VALUES(2, 1, 'Martin Eden');
INSERT INTO Books(Id, AuthorId, Title) VALUES(3, 2, 'Old Goriot');
INSERT INTO Books(Id, AuthorId, Title) VALUES(4, 2, 'Cousin Bette');
INSERT INTO Books(Id, AuthorId, Title) VALUES(5, 3, 'Jew Suess');
INSERT INTO Books(Id, AuthorId, Title) VALUES(6, 4, 'Nana');
INSERT INTO Books(Id, AuthorId, Title) VALUES(7, 4, 'The Belly of Paris');
INSERT INTO Books(Id, AuthorId, Title) VALUES(8, 5, 'In Cold blood');
INSERT INTO Books(Id, AuthorId, Title) VALUES(9, 5, 'Breakfast at Tiffany');
```

We have a books.sql file. It creates four database tables, Authors, Books, Testing and Images. Three tables are of InnoDB type. InnoDB databases support foreign key constraints and transactions. We place a foreign key constraint on the AuthorId column of the Books table. We fill the Authors and Books tables with initial data.

```
mysql> source books.sql
Query OK, 0 rows affected (0.07 sec)
Query OK, 0 rows affected (0.12 sec)
Query OK, 1 row affected (0.04 sec)
...
```

We use the source command to execute the books.sql script.

7 Prepared statements

Now we will concern ourselves with prepared statements. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements increase security and performance.

In Java a PreparedStatement is an object which represents a precompiled SQL statement.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Prepared {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;

        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "testuser";
        String password = "test623";

        try {

            String author = "Trygve Gulbranssen";
            con = DriverManager.getConnection(url, user, password);

            pst = con.prepareStatement("INSERT INTO Authors(Name) VALUES(?)");
            pst.setString(1, author);
```

```
            pst.executeUpdate();

        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Prepared.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } finally {

            try {
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(Prepared.class.getName());
                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }
        }
    }
}
```

We add a new author to the Authors table.

```
pst = con.prepareStatement("INSERT INTO Authors(Name) VALUES(?)");
```

Here we create a prepared statement. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements are faster and guard against SQL injection attacks. The ? is a placeholder, which is going to be filled later.

```
pst.setString(1, author);
```

A value is bound to the placeholder.

```
pst.executeUpdate();
```

The prepared statement is executed. We use the executeUpdate() method of the statement object when we don't expect any data to be returned. This is when we create databases or execute INSERT, UPDATE, DELETE statements.

```
$ java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/Prepared
```

```
mysql> select * from Authors;
+----+--------------------+
| Id | Name               |
+----+--------------------+
|  1 | Jack London        |
|  2 | Honore de Balzac   |
|  3 | Lion Feuchtwanger  |
|  4 | Emile Zola         |
|  5 | Truman Capote      |
|  6 | Trygve Gulbranssen |
+----+--------------------+
6 rows in set (0.00 sec)
```

We have a new author inserted into the table.

For the following two examples, we will use the Testing table. We will execute a normal statement and a prepared statement 1000 times. We check, if there is some difference in execution time.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;


public class NotPrepared {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;

        String cs = "jdbc:mysql://localhost:3306/testdb";
        String user = "testuser";
        String password = "test623";

        try {

            con = DriverManager.getConnection(cs, user, password);

            st = con.createStatement();

            for (int i=1; i<=1000; i++) {
```

```
            String query = "INSERT INTO Testing(Id) VALUES(" + 2*i + ")";
            st.executeUpdate(query);
        }

    } catch (SQLException ex) {
        Logger lgr = Logger.getLogger(NotPrepared.class.getName());
        lgr.log(Level.SEVERE, ex.getMessage(), ex);

    } finally {

        try {
            if (st != null) {
                st.close();
            }
            if (con != null) {
                con.close();
            }

        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(NotPrepared.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);
        }
    }
}
```

The first example uses the normal Statement object.

```
for (int i=1; i<=1000; i++) {
    String query = "INSERT INTO Testing(Id) VALUES(" + 2*i + ")";
    st.executeUpdate(query);
}
```

We build the query and execute it 1000 times.

```
$ /usr/bin/time java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/NotPrepa
1.09user 0.18system 0:46.37elapsed 2%CPU (0avgtext+0avgdata 92144maxresident)k
0inputs+96outputs (1major+6160minor)pagefaults 0swaps
```

We use the time command to measure the time, that the program ran. Note that we use a standard linux command, not the built-in bash time command. It took 46s to insert 1000 rows into the table using the Statement object.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Prepared2 {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;

        String cs = "jdbc:mysql://localhost:3306/testdb";
        String user = "testuser";
        String password = "test623";

        try {

            con = DriverManager.getConnection(cs, user, password);

            pst = con.prepareStatement("INSERT INTO Testing(Id) VALUES(?)");

            for (int i = 1; i <= 1000; i++) {
                pst.setInt(1, i * 2);
                pst.executeUpdate();
            }

        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Prepared2.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } finally {

            try {
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }
            } catch (SQLException ex) {
```

```
                Logger lgr = Logger.getLogger(Prepared2.class.getName());
                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }
        }
    }
}
```

Now we use the PreparedStatement to do the same task.

```
pst = con.prepareStatement("INSERT INTO Testing(Id) VALUES(?)");
```

We create the prepared statement using the prepareStatement() method.

```
for (int i = 1; i <= 1000; i++) {
    pst.setInt(1, i * 2);
    pst.executeUpdate();
}
```

We bind a value to the prepared statement, execute it in a loop thousand times.

```
$ /usr/bin/time java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/Prepared
1.08user 0.10system 0:32.99elapsed 3%CPU (0avgtext+0avgdata 90400maxresident)k
0inputs+96outputs (1major+6129minor)pagefaults 0swaps
```

Now it took 33s to insert 1000 rows. We have saved 13s using prepared statements.

8 Retrieving data

Next we will show, how to retrieve data from a database table. We get all data from the Authors table.

```
package zetcode;

import java.sql.PreparedStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Retrieve {
```

```
public static void main(String[] args) {

    Connection con = null;
    PreparedStatement pst = null;
    ResultSet rs = null;

    String url = "jdbc:mysql://localhost:3306/testdb";
    String user = "testuser";
    String password = "test623";

    try {

        con = DriverManager.getConnection(url, user, password);
        pst = con.prepareStatement("SELECT * FROM Authors");
        rs = pst.executeQuery();

        while (rs.next()) {
            System.out.print(rs.getInt(1));
            System.out.print(": ");
            System.out.println(rs.getString(2));
        }

    } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Retrieve.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

    } finally {

        try {
            if (rs != null) {
                rs.close();
            }
            if (pst != null) {
                pst.close();
            }
            if (con != null) {
                con.close();
            }

        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Retrieve.class.getName());
            lgr.log(Level.WARNING, ex.getMessage(), ex);
        }
    }
}
```

```
}
```

We get all authors from the Authors table and print them to the console.

```
pst = con.prepareStatement("SELECT * FROM Authors");
rs = pst.executeQuery();
```

We execute a query that selects all columns from the Authors table. We use the executeQuery() method. The method executes the given SQL statement, which returns a single ResultSet object. The ResultSet is the data table returned by the SQL query.

```
while (rs.next()) {
      System.out.print(rs.getInt(1));
      System.out.print(": ");
      System.out.println(rs.getString(2));
}
```

The next() method advances the cursor to the next record. It returns false, when there are no more rows in the result set. The getInt() and getString() methods retrieve the value of the designated column in the current row of this ResultSet object as an int/String in the Java programming language.

```
java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/Retrieve
1: Jack London
2: Honore de Balzac
3: Lion Feuchtwanger
4: Emile Zola
5: Truman Capote
6: Trygve Gulbranssen
```

We have Ids and Names of authors printed to the console.

9 Properties

It is a common practice to put the configuration data outside the program in a separate file. This way the programmers are more flexible. We can change the user, a password or a connection url without needing to recompile the program. It is especially useful in a dynamic environment, where is a need for a lot of testing, debugging, securing data etc.

In Java, the Properties is a class used often for this. The class is used for easy reading and saving of key/value properties.

```
db.url=jdbc:mysql://localhost:3306/testdb
```

```
db.user=testuser
db.passwd=test623
```

We have a database.properties file, in which we have three key/value pairs. These are dynamically loaded during execution of the program.

```
package zetcode;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Retrieve2 {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        Properties props = new Properties();
        FileInputStream in = null;

        try {
            in = new FileInputStream("database.properties");
            props.load(in);

        } catch (FileNotFoundException ex) {

            Logger lgr = Logger.getLogger(Retrieve2.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } catch (IOException ex) {

            Logger lgr = Logger.getLogger(Retrieve2.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);
```

```
    } finally {

        try {
             if (in != null) {
                 in.close();
             }
        } catch (IOException ex) {
            Logger lgr = Logger.getLogger(Retrieve2.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);
        }
    }

    String url = props.getProperty("db.url");
    String user = props.getProperty("db.user");
    String passwd = props.getProperty("db.passwd");

    try {

        con = DriverManager.getConnection(url, user, passwd);
        pst = con.prepareStatement("SELECT * FROM Authors");
        rs = pst.executeQuery();

        while (rs.next()) {
            System.out.print(rs.getInt(1));
            System.out.print(": ");
            System.out.println(rs.getString(2));
        }


    } catch (Exception ex) {
        Logger lgr = Logger.getLogger(Retrieve2.class.getName());
        lgr.log(Level.SEVERE, ex.getMessage(), ex);

    } finally {

        try {
            if (rs != null) {
                rs.close();
            }
            if (pst != null) {
                pst.close();
            }
            if (con != null) {
                con.close();
            }
```

```
            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(Retrieve2.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We connect to the testdb database and print the contents of the Authors table to the console. This time, we load the connection properties from a file. They are not hard coded in the proram.

```
Properties props = new Properties();
FileInputStream in = null;

try {
    in = new FileInputStream("database.properties");
    props.load(in);
...
```

The Properties class is created. The data is loaded from the file called database.properties, where we have our configuration data.

```
String url = props.getProperty("db.url");
String user = props.getProperty("db.user");
String passwd = props.getProperty("db.passwd");
```

The values are retrieved with the getProperty() method.

10 Multiple statements

It is possible to execute multiple SQL statements in one query. The allowMultiQueries must be set to enable multiple statements in MySQL.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class Multiple {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        String cs = "jdbc:mysql://localhost:3306/testdb?allowMultiQueries=true";
        String user = "testuser";
        String password = "test623";

        try {

            con = DriverManager.getConnection(cs, user, password);

            String query = "SELECT Id, Name FROM Authors WHERE Id=1;"
                    + "SELECT Id, Name FROM Authors WHERE Id=2;"
                    + "SELECT Id, Name FROM Authors WHERE Id=3";

            pst = con.prepareStatement(query);
            boolean isResult = pst.execute();

            do {
                rs = pst.getResultSet();

                while (rs.next()) {
                    System.out.print(rs.getInt(1));
                    System.out.print(": ");
                    System.out.println(rs.getString(2));
                }

                isResult = pst.getMoreResults();
            } while (isResult);


        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(Multiple.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } finally {

            try {
                if (rs != null) {
                    rs.close();
```

```
                }
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {

                Logger lgr = Logger.getLogger(Multiple.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

In the code example, we retrieve three rows from the Authors table. We use three SELECT statements to get three rows.

```
String cs = "jdbc:mysql://localhost:3306/testdb?allowMultiQueries=true";
```

We enable multiple statements queries in the database URL by setting the allowMultiQueries parameter to true.

```
String query = "SELECT Id, Name FROM Authors WHERE Id=1;"
        + "SELECT Id, Name FROM Authors WHERE Id=2;"
        + "SELECT Id, Name FROM Authors WHERE Id=3";
```

Here we have a query with multiple statements. The statements are separated by a semicolon.

```
boolean isResult = pst.execute();
```

We call the execute() method of the prepared statement object. The method returns a boolean value indicating if the first result is a ResultSet object. Subsequent results are called using the getMoreResults() method.

```
do {
    rs = pst.getResultSet();

    while (rs.next()) {
        System.out.print(rs.getInt(1));
        System.out.print(": ");
```

```
            System.out.println(rs.getString(2));
        }

        isResult = pst.getMoreResults();
    } while (isResult);
```

The processing of the results is done inside the do/while loop. The ResultSet is retrieved with the getResultSet() method call. To find out, if there are other results, we call the getMoreResults() method.

```
java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/Multiple
1: Jack London
2: Honore de Balzac
3: Lion Feuchtwanger
```

The output of the example. The first three rows were retrieved from the Authors table.

11 Column headers

Next we will show, how to print column headers with the data from the database table. We refer to column names as MetaData. MetaData is data about the core data in the database.

```
package zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Formatter;
import java.util.logging.Level;
import java.util.logging.Logger;


public class ColumnHeaders {


    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;
```

```
String cs = "jdbc:mysql://localhost:3306/testdb";
String user = "testuser";
String password = "test623";

try {

    con = DriverManager.getConnection(cs, user, password);
    String query = "SELECT Name, Title From Authors, " +
       "Books WHERE Authors.Id=Books.AuthorId";
    pst = con.prepareStatement(query);

    rs = pst.executeQuery();

    ResultSetMetaData meta = rs.getMetaData();

    String colname1 = meta.getColumnName(1);
    String colname2 = meta.getColumnName(2);

    Formatter fmt1 = new Formatter();
    fmt1.format("%-21s%s", colname1, colname2);
    System.out.println(fmt1);

    while (rs.next()) {
        Formatter fmt2 = new Formatter();
        fmt2.format("%-21s", rs.getString(1));
        System.out.print(fmt2);
        System.out.println(rs.getString(2));
    }

} catch (SQLException ex) {
    Logger lgr = Logger.getLogger(ColumnHeaders.class.getName());
    lgr.log(Level.SEVERE, ex.getMessage(), ex);

} finally {

    try {
        if (rs != null) {
            rs.close();
        }
        if (pst != null) {
            pst.close();
        }
        if (con != null) {
            con.close();
        }
```

```
            } catch (SQLException ex) {

                Logger lgr = Logger.getLogger(ColumnHeaders.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

In this program, we select authors from the Authors table and their books from the Books table. We print the names of the columns returned in the result set. We format the output.

```
String query = "SELECT Name, Title From Authors, " +
    "Books WHERE Authors.Id=Books.AuthorId";
```

This is the SQL statement which joins authors with their books.

```
ResultSetMetaData meta = rs.getMetaData();
```

To get the column names we need to get the ResultSetMetaData. It is an object that can be used to get information about the types and properties of the columns in a ResultSet object.

```
String colname1 = meta.getColumnName(1);
String colname2 = meta.getColumnName(2);
```

From the obtained metadata, we get the column names.

```
Formatter fmt1 = new Formatter();
fmt1.format("%-21s%s", colname1, colname2);
System.out.println(fmt1)
```

We print the column names to the console. We use the Formatter object to format the data.

```
while (rs.next()) {
    Formatter fmt2 = new Formatter();
    fmt2.format("%-21s", rs.getString(1));
    System.out.print(fmt2);
    System.out.println(rs.getString(2));
}
```

We print the data to the console. We again use the Formatter object to format the data. The first column is 21 characters wide and is aligned to the left.

```
$ java -cp .:lib/mysql-connector-java-5.1.13-bin.jar zetcode/ColumnHeaders
Name                Title
Jack London         Call of the Wild
Jack London         Martin Eden
Honore de Balzac    Old Goriot
Honore de Balzac    Cousin Bette
Lion Feuchtwanger   Jew Suess
Emile Zola          Nana
Emile Zola          The Belly of Paris
Truman Capote       In Cold blood
Truman Capote       Breakfast at Tiffany
```

Output of the program.

Date: 2013-11-29 Fri

Author: http://zetcode.com/databases/mysqljavatutorial/

Org version 7.8.11 with Emacs version 24

Validate XHTML 1.0