

Database access

One can access a database using the Java Database Connectivity (JDBC) library. The library is comparable to ODBC and allows all common database interactions. A driver is required for each database type, eg a driver is required to access MySQL. For MySQL the driver is MySQL Connector/J. It is “100% pure java” code that communicates directly with the particular database through sockets. Database vendors typically provide their own drivers. No special software or configuration is required on the client PC or the server, unlike ODBC where a DSN must be configured. These native drivers also provide high performance. There will thus be different drivers for other databases eg Oracle, postgresSQL etc. To be able to use one code base to access more than one database, use middleware (also referred to as a Type 3 driver, or JDBC-Net pure Java) to access the database.^{1, 2}

Steps for connecting to a database³:

- Import packages needed
- Register the JDBC driver
- Open a connection
- Execute a query,
- Access the data returned
- Close the connection when done

The following example illustrates database connection and access to the cdcols database on most XAMPP mysql installations.

Import packages needed

In this case import `java.sql.*` can be used, but it could be more specific. (as per recommended practice).

Register the JDBC driver

The JDBC driver is `com.mysql.jdbc.Driver`. The following call registers the driver:
`Class.forName("com.mysql.jdbc.Driver").newInstance();`
Remember to place all database interactions in try-catch blocks as they may fail.
For Oracle, the driver is `oracle.jdbc.driver.OracleDriver`.

Open a connection

To open a connection, a connection string is required. The string includes the protocol (`jdbc:mysql`), the server address (`localhost` in this case), the database name (`cdcol`) and the username and password. There was no password in this particular example.

¹ <http://www.careerride.com/JDBC-driver-types.aspx>

² <http://www.tutorialspoint.com/jdbc/jdbc-driver-types.htm>

³ <http://www.tutorialspoint.com/jdbc/jdbc-sample-code.htm>

```
conn = DriverManager.getConnection(  
    "jdbc:mysql://localhost/cdcol?user=root&password=");
```

For oracle, the connection string will be

```
jdbc:oracle:thin:username/password@serverAdress:port:dbName  
(thin is the driver)
```

Because a server address is specified (localhost in this case), the server need not be the local machine. It could be any other database server with a database account that permits remote access. Note that some accounts are created only for local access to increase security of data. Simply provide the IP address or server name of the remote database to access it.

Execute a query

This is a two step process: create a statement and execute it. This lends itself to the more robust parameterized queries (to be discussed later)

```
Statement s = conn.createStatement();  
ResultSet r = s.executeQuery("SELECT * FROM cds");
```

Access the data returned

```
while(r.next()) {  
    System.out.println (  
        r.getString("titel") + " " +  
        r.getString("interpret") + " " +  
        r.getString("jahr") );  
}
```

The records are returned as a **set**, and the individual fields (corresponding to the columns can be retrieved by field name.

Other methods besides getString() return specific types eg getInt() etc.

Also, instead of r.getString("title"), column numbers could be passed eg r.getString(1). This will return the first column. (it is not zero based as in arrays).

Close the connection when done

The usual steps are to close the resultSet, the statement and the connection

Eg

```
r.close();  
s.close();  
conn.close();
```

The following is a full example.

```
import java.sql.*;  
public class DBExample {  
  
    public static void main(String [] args) {  
  
        java.sql.Connection conn = null;
```

```

        System.out.println("This program demos DB connectivity");

        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = java.sql.DriverManager.getConnection(
                "jdbc:mysql://localhost/cdcol?user=root&password=");

        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(0);
        }

        System.out.println("Connection established");

        try {
            java.sql.Statement s = conn.createStatement();
            java.sql.ResultSet r = s.executeQuery("SELECT * FROM cds");
            while(r.next()) {
                System.out.println (
                    r.getString("titel") + " " +
                    r.getString("interpret") + " " +
                    r.getString("jahr") );
            }
            r.close();
            s.close();
            conn.close();

        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(0);
        }
    } //main
} //class

```

To run this, unzip the driver package and place the file mysql-connector-java-5.1.36-bin.jar in the same folder as your class file.

/* If using an older driver, or one with source code, unzip the driver eg mysql-connector-java-3.1.11-bin.jar file. Place the com folder (and org) folder in same directory as this file.
*/

Compile as:

Javac DBExample.java

Run as :

java -cp ../mysql-connector-java-5.1.36-bin.jar DBExample

on windows:

java -cp .\mysql-connector-java-5.1.36-bin.jar DBExample

The -cp parameter specifies the class path. The current directory as well as the (relative) path to the driver jar file are provided, separated by : on unix type systems or ; on windows. The final parameter is the program name.

Output for this particular example is

```

This program demos DB connectivity
Connection established
Beauty Ryuichi Sakamoto 1990

```

Error handling

The more appropriate, more specific errors to try to catch is SQLException.

It is useful to have a finally block to clean up whether in case one or more things fail, rather than just to simply exit at the first sign of failure. A more elaborate finally block can be helpful. Generally release resources in reverse order to that in which they were created. The following verifies that a resource is not null before attempting to close it finally. Note that there are tries and catches even within the finally.

```
finally{
    try{
        if(r!=null)
            r.close();
    }catch(SQLException se){
    }

    r=null;
    if(s!=null)
        s.close();
    }catch(SQLException se){
        //not much option here to do anything further
    }

    s=null;
    try{
        if(conn!=null)
            conn.close();
    }catch(SQLException se){
        se.printStackTrace();
    }
}
```

CRUD operations

Database operations make use of statement objects. As seen in the previous example, they may be regular **statements** objects, which may be fixed SQL queries which have no parameters.

After creating a statement, one of three methods may be called on the statement:

- `execute (String SQL)` → returns Boolean to indicate success or failure
- `executeUpdate (String SQL)` → returns an int, indicating number of rows affected.
- `executeQuery (String SQL):` → returns a ResultSet object of the set of records returned from the query.

The example above used the `executeQuery()` method, and obtained a result set. By default, recordsSets returned navigate forwards only, and are read only. See the documentation for navigating backward or making the result set updateable.

Two other ways to interact with the database are by

- **Prepared statements** which accept parameters for the query or
- **Callable statements** for calling stored procedures. Parameters may be accepted.

Prepared statements help avoid SQL injection, although parameters can be passed on the fly, the quotes supplied and other special characters are ignored.

The following snippet shows how to insert into a database

```
public void addRecord(){
    Scanner in=new Scanner(System.in);
```

```

        System.out.println("enter a Title");
        String title=in.nextLine();
        System.out.println("enter a Description");
        String description=in.nextLine();
        System.out.println("enter a Year");
        int year=in.nextInt();

        try{
            PreparedStatement p=conn.prepareStatement("Insert Into cds set titel=?,
interpret=?, jahr =?");
            p.setString(1, title);
            p.setString(2, description);
            p.setInt(3, year);
            p.execute(); //use execute if no results expected back
        }catch(Exception e){
            System.out.println("Error"+e.toString());
            return;
        }
    }
}

```

Place holders are provided in the SQL statement and the placehodlers are replaced using functions such as `setString`, `setInt`, `setDate`, `setFloat`, etc (functions exist for all the primitive types). The placeholders are numbered from 1 and up.

The following is an example of updating records

```

try{
    PreparedStatement p=conn.prepareStatement("update cds set titel=? where jahr
>?");
    p.setString(1, "Futuristic");
    p.setInt(2, newYr);
    int x= p.executeUpdate(); //use execute if no results expected back
    System.out.println("Number of rows affected is "+ x);
    p.close();
}catch(Exception e){
    System.out.println("Error"+e.toString());
    return;
}
}

```

Deleting a record is similar.

The following is a fuller example

```

/* unzip the mysql-connector-java-3.1.11-bin.jar file. Place the com folder (and
org) folder in same directory as this file
*/

```

```

import java.util.*;
import java.io.*;
import java.sql.*;
public class DBcrud {

    java.sql.Connection conn = null;

    public static void main(String [] args) {

        System.out.println("This program demos DB CRUD operations");
        DBcrud app= new DBcrud();
        app.initialize();
        app.menu();

    }
}

```

```

public void initialize(){
    try {
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        conn = java.sql.DriverManager.getConnection(
            "jdbc:mysql://localhost/cdcol?user=root&password=");
    }
    catch (Exception e) {
        System.out.println(e);
        System.exit(0);
    }
    System.out.println("Connection established");
}

public void listAll(){
    try {
        java.sql.Statement s = conn.createStatement();
        java.sql.ResultSet r = s.executeQuery("SELECT * FROM cds");
        while(r.next()) {
            System.out.println (
                r.getString("titel") + " " +
                r.getString("interpret") + " " +
                r.getString("jahr") );
        }
    }
    catch (Exception e) {
        System.out.println(e);
        System.exit(0);
    }
}

public void addRecord(){
    Scanner in=new Scanner(System.in);
    System.out.println("enter a Title");
    String title=in.nextLine();
    System.out.println("enter a Description");
    String description=in.nextLine();
    System.out.println("enter a Year");
    int year=in.nextInt();

    try{
        PreparedStatement p=conn.prepareStatement("Insert Into cds set titel=?,
interpret=?, jahr =?");
        p.setString(1, title);
        p.setString(2, description);
        p.setInt(3, year);
        p.execute(); //use execute if no results expected back
    }catch(Exception e){
        System.out.println("Error"+e.toString());
        return;
    }
    // do another raw insert
    try{
        PreparedStatement p2=conn.prepareStatement("Insert into cds set
titel='Ashesi Uni', interpret='Made in Berekuso', jahr=2002");
        p2.execute();
        p2.close();
    }catch(Exception e){
        System.out.println("Error"+e.toString());
        return;
    }
}

public void updateRecords(int newYr){
    try{

```

```

        PreparedStatement p=conn.prepareStatement("update cds set titel=? where jahr
>?");
        p.setString(1, "Futuristic");
        p.setInt(2, newYr);
        int x= p.executeUpdate(); //use execute if no results expected back
        System.out.println("Number of rows affected is "+ x);
        p.close();
    }catch(Exception e){
        System.out.println("Error"+e.toString());
        return;
    }
}

public void menu(){
while (true){
    int choice=-1;
    System.out.println("\n\nWhat do you want to test\n"+
        "1. Print All records\n"+
        "2. Add a record\n"+
        "3. Delete a record\n"+
        "4. Update a record\n\n"+
        "9. Exit Program\n");
    try{
        BufferedReader s= new BufferedReader(new
InputStreamReader(System.in));
        choice= Integer.parseInt(s.readLine());
        //s.close(); //move this to later.
    }catch (IOException ioe){ System.out.println(ioe.toString()); }

    switch(choice){
        case 1: listAll(); break;
        case 2: addRecord(); break;
        case 3: System.out.println("implementation pending"); break;
        case 4: updateRecords(2011); break;
        case 9:
            System.out.println("thank you for testing...");
            System.exit(0);
            break;

        default:
            System.out.println("Please select an item from menu");

    } //switch
} //while
} //menu
} //class

```

Additional notes: A RecordSet can be open to allow traversal forwards and backward. Data in the recordSet can also be updated and appended to, depending on how it is open. As an example, this snippet will open the connection so that it can be traversed both ways (scrollable), and not show concurrent changes by other users. The data will be updateable: Statement stmt = con.createStatement(

```

        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

```

See the documentation for methods to insert new blank record (rs.moveToInsertRow) and to actually insert the new data, also methods to update a row.⁴

⁴ "Interface ResultSet", <http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html>

Using the swing table model with a table

A useful method to obtain information about table column names (or field names in a result set) is to use the `getMetaData()` method. It retrieves the number, types and properties of a `ResultSet` object's columns.

To use a table model, the data should be retrieved from the database and placed in a data structure such that the three required methods can be easily implemented. Note that it is generally not prudent to retrieve all the records of a large table. A subset will usually suffice, and pagination may be employed to obtain additional records.

This example is illustrative. The key method is `fetchTableData()`.

```
package com.namanquah.mvc.models;

import java.sql.*;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.ArrayList;
import javax.swing.table.AbstractTableModel;

/**
 *
 * @author namanquah
 */
public class MyTableModel extends AbstractTableModel {
    Object[][] data;
    static MyTableModel theModel=null;
    String[] columnNames =null;

    private MyTableModel() {
        super();
        fetchTableData(); //my own function to initialize the arraylist
    }

    public static MyTableModel getInstance(){ //I am using the Singleton
        pattern here
        if (theModel==null)
            theModel= new MyTableModel();
        return theModel;
    }

    @Override
    public int getColumnCount() {
        return columnNames.length; //we know student has 4 fields
    }

    @Override
    public int getRowCount() {
        return data.length;
    }

    @Override
    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    //fxn getValueAt

    //optional functions
}
```



```

@Override
public String getColumnName(int col) {
    return columnNames[col]; //otherwise returns A, B, C etc
}
//override this to allow editing of some columns/rows

@Override
public boolean isCellEditable(int row, int col) {

    return false;
}

// uses this to determine the default renderer or
// editor for each cell.
public Class getColumnClass(int c) {
    return getValueAt(0, c).getClass();
}

void fetchTableData(){
try {
    Connection conn = null;
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    conn = java.sql.DriverManager.getConnection(
        "jdbc:mysql://localhost/cdcol?user=root&password=");

    Statement s = conn.createStatement();

    ResultSet rs = s.executeQuery("SELECT titel, interpret, jahr, id FROM
cds");
    ResultSetMetaData meta = rs.getMetaData();
    int numberOfColumns = meta.getColumnCount();
    columnNames= new String[numberOfColumns];
    for (int i = 1; i <= numberOfColumns; i++) { //note its 1 based.
        columnNames[i - 1] = meta.getColumnName(i); //columns Zero based.
    }

    ArrayList <Object> rows = new ArrayList<Object>();
    //get actual row data

    while(rs.next()) {
        ArrayList <String> tmp = new ArrayList<String>();
        tmp.add(rs.getString(1)); System.out.println("row"+ rs.getString(1));
        tmp.add(rs.getString(2));
        tmp.add(rs.getString("jahr"));
        tmp.add(rs.getString("id"));
        //System.out.println(tmp);
        rows.add(tmp);
    }

    //convert to 2d array data
    data= new Object[rows.size()][3];

    for(int i=0; i<data.length; i++){
        ArrayList<Object> tmp2= (ArrayList<Object>) rows.get(i);
        data[i]= tmp2.toArray();
    }

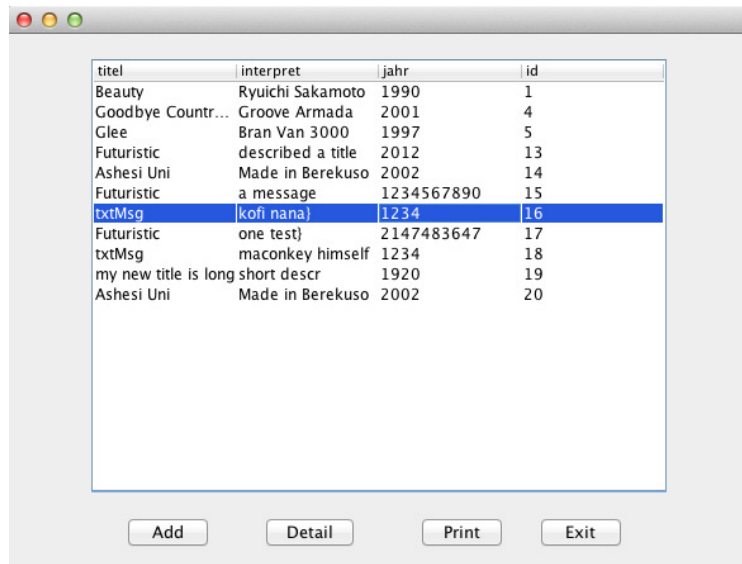
}
catch (Exception e) {
    e.printStackTrace();
    System.out.println(e);
    System.exit(0);
}

}

```

}

The output (based on a previous example) is as shown in Figure 1.



titel	interpret	jahr	id
Beauty	Ryuichi Sakamoto	1990	1
Goodbye Countr...	Groove Armada	2001	4
Glee	Bran Van 3000	1997	5
Futuristic	described a title	2012	13
Ashesi Uni	Made in Berekuso	2002	14
Futuristic	a message	1234567890	15
txtMsg	kofi nana	1234	16
Futuristic	one test	2147483647	17
txtMsg	maconkey himself	1234	18
my new title is long	short descr	1920	19
Ashesi Uni	Made in Berekuso	2002	20

Figure 1 Output of using table model filled with database entries

There are other strategies for populating the table model⁵.

⁵ "Datatable&JTable with Abstract Table Model"

<http://babek555.blogspot.com/2012/10/database-with-abstract-table-model.html>