JDBC NOTES

codeforsuccess.in

Chapter 1

Components

The JDBC API allows programmers and Java applications to interact with databases. It supports executing different SQL statements and handling results coming from different data sources.

In this section we will try to summarize and list the most important JDBC components that are part of every Java application, all of them will be explained in more detail in the next chapters.

- First of all, Java applications need to create and establish a connection ao a specific database. This is done using a Driver Manager, for example, one instance of the interface java.sql.DriverManager, or directly via a JDBC data source. For this purpose, the interface javax.sql.DataSource can be used. As already mentioned, we will explain these components in more in detail in the next chapters.
- Once we are connected against a database, we can use our java.sql.Connection for executing CRUD (create, read, update, delete) SQL statements or operations. These statements are explained afterwards in this tutorial.
- In order to execute these these operations, programmers can use java.sql.Statement and java.sql.PreparedStatement based classes. The last ones are more efficient when executing the same statement several times and provide other benefits that we will list in this tutorial. The interface JDBC connection provides mechanisms to create statement instances:

```
PreparedStatement countriesStatement = connection.prepareStatement("UPDATE COUNTRIES SET  ↩
    NAME = ? WHERE ID = ?");
countriesStatement.setString(1, "Spain");
countriesStatement.setInt(2, 123456789);
```

- Operations like Insert, update or delete return back the number of modified rows and nothing else:

```
// countriesStatement belongs to the class Statement, returning number of updated rows
int n = countriesStatement.executeUpdate();
```

- Selection operations (queries) return results as rows inside a java.sql.ResultSet. Rows are retrieved by name or number; results metadata is also available:

```
// countriesStatement belongs to the class Statement
ResultSet rs = countriesStatement.executeQuery("SELECT NAME, POPULATION FROM COUNTRIES");
//rs contains the results in rows plus some metadata
...
```

- Normally, JDBC uses connection pools for managing connections. There are different implementations for connection pools like C3P0 or DBCP. These are groups of JDBC connections that are used or borrowed from the applications when needed and released when the task is finished. There is a lot of documentation about how to use and configure connection pools within JDBC, a good tutorial can be found in the following link http://docs.oracle.com/cd/E13222_01/wls/docs81/ConsoleHelp/-jdbc_connection_pools.html.

- Other features are available while working with JDBC: Stored Procedures, Callable Statements, Batch Processing... all these will be described in this tutorial.

Chapter 2

Connections

In order to connect to a database we need to use a `java.sql.Connection` object. We can do this using the `getConnection()` method of the `java.sql.DriverManager` class. This methods receives the database host and credentials as parameters.

This snippet shows how to create a connection for a local MySQL database.

```
//MySQL driver is loaded
Class.forName( "com.mysql.jdbc.Driver" );
//Connection object is created using the db host and credentials
Connection connect = DriverManager.getConnection("jdbc:mysql://localhost/countries?"
            + "user=root&amp;password=root" );
```

A connection objects allows programmers to do the following actions:

- Creation of JDBC Statements: Using a connection object is possible to create `Statement`, `PreparedStatement` or `CallableStatement` instances that offer methods to execute different SQL statements. Here is an example of the creation of a `PreparedStatement`:

```
//the connection conn is used to create a prepared statement with the given sql operation
PreparedStatement updateStmt = conn.prepareStatement( sql );
```

This statement can execute the sql update passed as parameter.

- Offers the possibility to commit or rollback a given transaction. JDBC connection supports two different ways of working: `autocommit=true` and `autocommit=false`. The first one commits all transactions directly to the database, the second one needs an special command in order to commit or rollback the transactions. We will see this is more detail in the related chapter in this tutorial. The following piece of code shows how to change the auto commit mode of a JDBC connection:

```
//it changes the mode to auto commit=false
connect.setAutoCommit( false );
```

- Possibility to get meta information about the database that is been used.
- Other options like batch processing, stored procedures, etc.

We will explain all these features in detail, for the moment it is good to know what a JDBC Connection is and what can be done using JDBC Connections.

Chapter 3

Data types

JDBC converts the Java data types into proper JDBC types before using them in the database. There is a default mapping between Java and JDBC data types that provides consistency between database implementations and drivers.

The following table contains these mappings:

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

Null values are treated differently in SQL and in Java. When handling with SQL null values in Java it is good to follow some best practices like avoiding the usage of primitive types, since they cannot be null but converted to their default values like 0 for int, false for booleans, etc.

Instead of that, the usage of wrapper classes for the primitive types is recommended. The class `ResultSet` contains a method called `wasNull()` that is very useful in these scenarios. Here is an example of its usage:

```
Statement stmt = conn.createStatement( );
String sql = "SELECT NAME, POPULATION FROM COUNTRIES";
ResultSet rs = stmt.executeQuery(sql);

int id = rs.getInt(1);
if( rs.wasNull( ) ) {
```

```
    id = 0;
}
```

Chapter 4

Drivers

The JDBC Driver Manager, `java.sql.DriverManager`, is one of the most important elements of the JDBC API. It is the basic service for handling a list of JDBC Drivers. It contains mechanisms and objects that allow Java applications to connect to a desired JDBC driver. It is in charge of managing the different types of JDBC database drivers. Summarizing the main task of the Driver Manager is to be aware of the list of available drivers and to handle the connection between the specific selected driver and the database.

The most frequently used method of this class is `DriverManager.getConnetion()`. This method establishes a connection to a database.

Here is an example of its use:

```
// Create the connection with the default credentials
java.sql.Connection conn = DriverManager.getConnection("jdbc:hsqldb:mem:mydb", "SA", "" );
```

We can register drivers using the method `DriverManager.registerDriver().`:

```
new org.hsqldb.jdbc.JDBCDriver();
DriverManager.registerDriver( new org.hsqldb.jdbc.JDBCDriver() );
```

We can also load a driver by calling the `Class.forName()` method:

```
// Loading the HSQLDB JDBC driver
Class.forName( "org.hsqldb.jdbc.JDBCDriver" );

...

// connection to JDBC using mysql driver
Class.forName( "com.mysql.jdbc.Driver" );
```

The main difference is that the method `registerDriver()` needs that the driver is available at compile time, loading the driver class does not require that the driver is available at compile time. After JDBC 4, there is no real need of calling these methods and applications do not need to register drivers individually neither to load the driver classes. It is also not recommended to register drivers manually using the method `registerDriver()`.

Other interesting methods of the `DriverManager` class are `getDriver(String url)`, that tries to locate the driver by a given string and `getDrivers()` that returns an enumeration of all the drivers that has been previously registered in the Driver Manager:

```
Enumeration drivers = DriverManager.getDrivers();
while( drivers.hasMoreElements() )
{
        Driver driver = drivers.nextElement();
        System.out.println( driver.getClass() );
}
```

Chapter 5

Databases

JDBC supports a large list of databases. It abstracts its differences and ways of working by using different Drivers. The `DriverManager` class is in charge of loading the proper database, after this is loaded, the code that access the database for querying and modifying data will remain (more or less) unchanged.

Here is a list of supported databases in JDBC (officially registered within Oracle): http://www.oracle.com/technetwork/java/index-136695.html.

In this chapter we are going to show how to use to different databases: MySQL and HSQLDB. The first one is very well known by programmers and wide used, the second one, HSQLDB, is a database very helpful for testing purposes that offers in memory capabilities. We will see how to use both and we will discover that except of the loading of the proper JDBC driver, the rest of the application remains unchanged:

MySQL example:

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

        // connection to JDBC using mysql driver
        Class.forName( "com.mysql.jdbc.Driver" );
        Connection connect = DriverManager.getConnection("jdbc:mysql://localhost/countries? ↩
            "
                + "user=root&amp;password=root" );


        selectAll( connect );

        // close resources, in case of exception resources are not properly cleared
...

}

/**
 * select statement and print out results in a JDBC result set
 *
 * @param conn
 * @throws SQLException
 */
private static void selectAll( java.sql.Connection conn ) throws SQLException
{
        Statement statement = conn.createStatement();

        ResultSet resultSet = statement.executeQuery( "select * from COUNTRIES" );

        while( resultSet.next() )
        {
```

```
                String name = resultSet.getString( "NAME" );
                String population = resultSet.getString( "POPULATION" );

                System.out.println( "NAME: " + name );
                System.out.println( "POPULATION: " + population );
        }

}
```

In memory (HSQLDB) example:

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

        // Loading the HSQLDB JDBC driver
        Class.forName( "org.hsqldb.jdbc.JDBCDriver" );

        // Create the connection with the default credentials
        java.sql.Connection conn = DriverManager.getConnection( "jdbc:hsqldb:mem:mydb", "SA ↩
            ", "" );

        // Create a table in memory
        String countriesTableSQL = "create memory table COUNTRIES (NAME varchar(256) not  ↩
            null primary key, POPULATION varchar(256) not null);";

        // execute the statement using JDBC normal Statements
        Statement st = conn.createStatement();
        st.execute( countriesTableSQL );

        // nothing is in the database because it is just in memory, non persistent
        selectAll( conn );

        // after some insertions, the select shows something different, in the next  ↩
            execution these
        // entries will not be there
        insertRows( conn );
        selectAll( conn );

}

...

/**
 * select statement and print out results in a JDBC result set
 *
 * @param conn
 * @throws SQLException
 */
private static void selectAll( java.sql.Connection conn ) throws SQLException
{
        Statement statement = conn.createStatement();

        ResultSet resultSet = statement.executeQuery( "select * from COUNTRIES" );

        while( resultSet.next() )
        {
                String name = resultSet.getString( "NAME" );
                String population = resultSet.getString( "POPULATION" );

                System.out.println( "NAME: " + name );
                System.out.println( "POPULATION: " + population );
        }
```

```
}
```

As we can see in last programs, the code of the `selectAll` methods is completely the same, only the JDBC Driver loading and connection creation changes; you can imagine how powerful this is when working in different environments. The HSQLDB version of the code contains also the piece of code in charge of creating the in memory database and inserting some rows, but this is just for showing and clarity purposes and can be done differently.

Chapter 6

Result sets

The class `java.sql.ResultSet` represents a result set of database table. It is created, normally; by executing an SQL query (select statement using Statement or PreparedStatement). It contains rows of data, where the data is stored. These data can be accessed by index (starting by 1) or by attribute name:

```
// creating the result set
ResultSet resultSet = statement.executeQuery( "select * from COUNTRIES" );

// iterating through the results rows

while( resultSet.next() )
{
        // accessing column values by index or name
        String name = resultSet.getString( "NAME" );
        int population = resultSet.getInt( "POPULATION" );

        System.out.println( "NAME: " + name );
        System.out.println( "POPULATION: " + population );


        // accessing column values by index or name
        String name = resultSet.getString( 1 );
        int population = resultSet.getInt( 2 );

        System.out.println( "NAME: " + name );
        System.out.println( "POPULATION: " + population );


}
```

As shown before, ResultSets contain getter methods for retrieving column values for different Java types. It also contains a cursor pointing to the current row of data. Initially, the cursor is pointing before the first row. The next method moves the cursor to the next row: `java.sql.ResultSet.next()`.

It is possible to create ResultSets with default properties like a cursor that moves forward only and that is not updatable. If programmers would like to use other kind of properties he can specify so in the creation of the Statement that is going to produce the result sets by changing the arguments passed:

```
/**
* indicating result sets properties that will be created from this statement: type,
* concunrrency and holdability
*/
Statement statement = conn.createStatement( ResultSet. TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE, ResultSet.CLOSE_CURSORS_AT_COMMIT );
```

Using this kind of result sets it is possible to move the cursor in both directions and to update or insert new data into the database using the result set with this purpose.

Chapter 7

Stored procedures

In this chapter we are going to explain what stored procedures are and how we can use them within JDBC. For the examples we are going to use MySQL based stored procedures.

Stored procedures are sets of SQL statements as part of a logical unit of executiion and performing a defined task. They are very useful while encapsulating a group of operations to be executed on a database.

First of all we are going to create a procedure in our MySQL database, following script will help us with this task:

```
delimiter //

CREATE PROCEDURE spanish (OUT population_out INT)
BEGIN
SELECT COUNT(*) INTO population_out FROM countries;
END//


delimiter ;

CALL simpleproc(@a);
```

Basically the script above creates a procedure called Spanish with one output attribute of the type int and without input parameters. The procedure returns the count of all countries in the database.

Once we have created the procedure we can work with it from our Java applications.In order to call Stored Procedures we need to use special statements of the interface java.sql.CallableStatement, these statements allow programmers to execute stored procedures indicating the output attributes and input parameters to be used. In our simple example, only output attributes are configured. Here is an example:

```
CallableStatement callableStatement = null;

// the procedure should be created in the database
String spanishProcedure = "{call spanish(?)}";

// callable statement is used
callableStatement = connect.prepareCall( spanishProcedure );

// out parameters, also in parameters are possible, not in this case
callableStatement.registerOutParameter( 1, java.sql.Types.VARCHAR );

// execute using the callable statement method executeUpdate
callableStatement.executeUpdate();

// attributes are retrieved by index
String total = callableStatement.getString( 1 );

System.out.println( "amount of spanish countries " + total );
```

We can appreciate how to indicate where to store the output of the procedure and how to execute it using the method `java.sql.PreparedStatement.executeUpdate()`. Stored procedures are supported in most of the databases but their syntax and behavior may differ, that is why there may be differences in the Java applications handling stored procedures depending on the databases where the procedures are stored.

Chapter 8

Statements

As already mentioned in this tutorial, JDBC uses the interface `java.sql.Statement` to execute different SQL queries and operations like insert, update or delete. This is the basic interface that contains all the basic methods like `java.sql.Statement.executeQuery(String)` or `java.sql.Statement.executeUpdate(String)`.

Implementations of this interface are recommended when programmers do not need to execute same query multiple times or when queries and statements do not need to be parameterized. In general, we can say that this interface is suitable when executing DDL statements (Create, Alter, Drop). These statements are not executed multiple times normally and do not need to support different parameters.

In case programmers need better efficiency when repeating SQL queries or parameterization they should use `java.sql.PreparedStatement`. This interface inherits the basic statement interface mentioned before and offers parameterization. Because of this functionalitiy, this interface is safer against SQL injection attacks. Here is a piece of code showing an example of this interface:

```
System.out.println( "Updating rows for " + name + "..." );

String sql = "UPDATE COUNTRIES SET POPULATION=? WHERE NAME=?";

PreparedStatement updateStmt = conn.prepareStatement( sql );

// Bind values into the parameters.
updateStmt.setInt( 1, 10000000 ); // population
updateStmt.setString( 2, name ); // name

// update prepared statement using executeUpdate
int numberRows = updateStmt.executeUpdate();

System.out.println( numberRows + " rows updated..." );
```

Another benefit of using prepared statements is the possibility to handle non standard objects by using the `setObject()` method. Here is an example:

```
PreparedStatement updateStmt2 = conn.prepareStatement( sql );

// Bind values into the parameters using setObject, can be used for any kind and type of
// parameter.
updateStmt2.setObject( 1, 10000000 ); // population
updateStmt2.setObject( 2, name ); // name

// update prepared statement using executeUpdate
numberRows = updateStmt2.executeUpdate();

System.out.println( numberRows + " rows updated..." );
updateStmt2.close();
```

As mentioned in the chapter related to stored procedures, another interface is available for this purpose, it is called `java.sql.CallableStatement` and extends the PreparedStatement one.

Chapter 9

Batch commands

JDBC offers the possibility to execute a list of SQL statements as a batch, that is, all in a row. Depending on what type of Statements the programmers are using the code may differ but the general idea is the same. In the next snippet is shown how to use batch processing with `java.sql.Statement`:

```
Statement statement = null;

statement = connect.createStatement();

// adding batchs to the statement
statement.addBatch( "update COUNTRIES set POPULATION=9000000 where NAME='USA'" );
statement.addBatch( "update COUNTRIES set POPULATION=9000000 where NAME='GERMANY'" );
statement.addBatch( "update COUNTRIES set POPULATION=9000000 where NAME='ARGENTINA'" );

// usage of the executeBatch method
int[] recordsUpdated = statement.executeBatch();

int total = 0;
for( int recordUpdated : recordsUpdated )
{
        total += recordUpdated;
}

System.out.println( "total records updated by batch " + total );
```

And using `java.sql.PreparedStatement`:

```
String sql = "update COUNTRIES set POPULATION=? where NAME=?";

PreparedStatement preparedStatement = null;

preparedStatement = connect.prepareStatement( sql );

preparedStatement.setObject( 1, 1000000 );
preparedStatement.setObject( 2, "SPAIN" );

// adding batches
preparedStatement.addBatch();

preparedStatement.setObject( 1, 1000000 );
preparedStatement.setObject( 2, "USA" );

// adding batches
preparedStatement.addBatch();
```

```
// executing all batchs
int[] updatedRecords = preparedStatement.executeBatch();
int total = 0;
for( int recordUpdated : updatedRecords )
{
        total += recordUpdated;
}

System.out.println( "total records updated by batch " + total );
```

We can see that the differences are basically the way the SQL query parameters are used and how the queries are built, but the idea of executing several statements on one row is the same. In the first case by using the method `java.sql.Statement.executeBatch()`, using `java.sql.PreparedStatement.addBatch()` and `java.sql.Statement.executeBatch()` in the second one.

Chapter 10

Transactions

JDBC supports transactions and contains methods and functionalities to implement transaction based applications. We are going to list the most important ones in this chapter.

- `java.sql.Connection.setAutoCommit(boolean)`: This method receives a Boolean as parameter, in case of true (which is the default behavior), all SQL statements will be persisted automatically in the database. In case of false, changes will not be persisted automatically, this will be done by using the method `java.sql.Connection.commit()`.
- `java.sql.Connection.commit()`. This method can be only used if the auto commit is set to false or disabled; that is, it only works on non automatic commit mode. When executing this method all changes since last commit / rollback will be persisted in the database.
- `java.sql.Connection.rollback()`. This method can be used only when auto commit is disabled. It undoes or reverts all changes done in the current transaction.

And here is an example of usage where we can see how to disable the auto commit mode by using the method `setAutoCommit(false)`. All changes are committed when calling `commit()` and current transaction changes are rolled back by using the method `rollback()`:

```
Class.forName( "com.mysql.jdbc.Driver" );
Connection connect = null;
try
{
    // connection to JDBC using mysql driver
    connect = DriverManager.getConnection( "jdbc:mysql://localhost/countries?"
                + "user=root&amp;password=root" );
    connect.setAutoCommit( false );

    System.out.println( "Inserting row for Japan..." );
    String sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('JAPAN', '45000000')";

    PreparedStatement insertStmt = connect.prepareStatement( sql );

    // insert statement using executeUpdate
    insertStmt.executeUpdate( sql );
    connect.rollback();

    System.out.println( "Updating row for Japan..." );
    // update statement using executeUpdate -> will cause an error, update will not be
    // executed becaues the row does not exist
    sql = "UPDATE COUNTRIES SET POPULATION='1000000' WHERE NAME='JAPAN'";
    PreparedStatement updateStmt = connect.prepareStatement( sql );

    updateStmt.executeUpdate( sql );
```

```
    connect.commit();

}
catch( SQLException ex )
{
    ex.printStackTrace();
    //undoes all changes in current transaction
    connect.rollback();
}
finally
{
    connect.close();
}
```

Chapter 11

CRUD commands

CRUD comes from Create, Read, Update and Delete. JDBC supports all these operations and commands, in this chapter we are going to show difference snippets of Java code performing all of them:

Create Statement. It is possible to create databases using JDBC, here is an example of creation of a in memory database:

```
// Create a table in memory
String countriesTableSQL = "create memory table COUNTRIES (NAME varchar(256) not null ↩
    primary key, POPULATION varchar(256) not null);";

// execute the statement using JDBC normal Statements
Statement st = conn.createStatement();
st.execute( countriesTableSQL );
```

Insert Statement. Inserts are supported in JDBC. Programmers can use normal SQL syntax and pass them to the different statement classes that JDBC offers like `Statement`, `PreparedStatement` or `CallableStatement`. Here are a couple of examples:

```
Statement insertStmt = conn.createStatement();

String sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('SPAIN', '45Mill')";
insertStmt.executeUpdate( sql );

sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('USA', '200Mill')";
insertStmt.executeUpdate( sql );

sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('GERMANY', '90Mill')";
insertStmt.executeUpdate( sql );
```

These statements return the number of inserted rows. The same is applicable to update statements, here is an example of how to update a set of rows in a database:

```
System.out.println( "Updating rows for " + name + "..." );

Statement updateStmt = conn.createStatement();

// update statement using executeUpdate
String sql = "UPDATE COUNTRIES SET POPULATION='10000000' WHERE NAME='" + name + "'";
int numberRows = updateStmt.executeUpdate( sql );

System.out.println( numberRows + " rows updated..." );
```

The output would be:

```
Updating rows for SPAIN...
4 rows updated...
```

Select Statement. It is possible to execute any (almost) kind of SQL query using JDBC statements. Here is a very simple example that reads all the rows of a given table and prints them out in the standard console:

```
Statement statement = conn.createStatement();

ResultSet resultSet = statement.executeQuery( "select * from COUNTRIES" );

while( resultSet.next() )
{
        String name = resultSet.getString( "NAME" );
        String population = resultSet.getString( "POPULATION" );
        System.out.println( "NAME: " + name );
        System.out.println( "POPULATION: " + population );
}
```

The output of this would be (depending on the database state):

```
NAME: GERMANY
POPULATION: 90Mill
NAME: SPAIN
POPULATION: 45Mill
NAME: USA
POPULATION: 200Mill
```

Delete statement. Finally, JDBC supports deletion of rows and dropping of tables and other SQL elements. Here is a snippet showing the deletion of all rows with an specific criteria (in this case, the name has to be "JAPAN"):

```
System.out.println( "Deleting rows for JAPAN..." );
String sql = "DELETE FROM COUNTRIES WHERE NAME='JAPAN'";
PreparedStatement deleteStmt = connect.prepareStatement( sql );

// delete statement using executeUpdate
int numberRows = deleteStmt.executeUpdate( sql );

System.out.println( numberRows + " rows deleted..." );
```

Delete statements return the number of affected rows, in this case the output would be (depending on the database state):

```
Deleting rows for JAPAN...
0 rows deleted...
```

These examples are all very simple ones; they have been written for learning purposes but you can imagine that you can execute more complicated SQL queries just by changing the argument passed to the `executeQuery()` or `executeUpdate()` methods.