Contents

Preface

The Java Database Connectivity (JDBC) is an important Java API that defines how a client accesses a database. As such it is a critical component in building large-scale enterprise Java solutions. At a high level, interacting with a database involves the following steps:

1. Establish a connection to the database
2. Execute SQL queries to retrieve create or modify tables in the database
3. Close the connection to the database

Java provides a set of API to carry out these activities with databases. `JDBC` can be used to establish a connection to a database, execute SQL query and close the connection with the database. The benefit of `JDBC` is that it is not writing a program for a specific database. `JDBC` creates a loose coupling between your Java program and the type of database used. For instance, databases may differ in how they establish a connection, `JDBC` hides all that heterogeneity of these databases and offers a single set to API you can use to interact with all types of databases. Note that `JDBC` supports only relational databases such as `MySQL`, `Oracle`, `MSSQL` and `DB2`. It does not support new-generation databases such as `NoSQL` databases.

JDBC

There are several vital components which comprise the `JDBC` and how these components work is described below. A simplified architecture of `JDBC` is represented below

The database drivers and the driver manager play a key role in realizing the objective of the `JDBC`. `JDBC` drivers are specifically designed to interact with their respective DBMS. The driver manager works as a directory of `JDBC` drivers - it maintains a list of available data sources and their drivers. The driver manager chooses an appropriate driver to communicate with the respective DBMS. It can manage multiple concurrent drivers connected to their respective data sources.

Connection

The connection interface represents a connection from application to the database. It is a channel through which the application and the database communicate. Below are listed some of the most important methods

in the Connection interface.

Method	Description
Statement createStatement()	Creates a Statement object that can be used to send SQL statements to the database.
PreparedStatement prepareStatement(String sql)	Creates a PreparedStatement object that can contain SQL statements. The SQL statement can have IN parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.
CallableStatement prepareCall(String sql)	Creates a CallableStatement object for calling stored procedures in the database. The SQL statement can have IN or OUT parameters; they may contain ? symbol(s), which are used as placeholders for passing actual values later.
DatabaseMetaData getMetaData()	Gets the DataBaseMetaData object. This metadata contains database schema information, table information, and so on, which is especially useful when you don't know the underlying database.
Clob createClob()	Returns a Clob object (Clob is the name of the interface). Character Large Object (CLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
Blob createBlob()	Returns a Blob object (Blob is the name of the interface). Binary Large Object (BLOB) is a built-in type in SQL; it can be used to store a column value in a row of a database table.
void setSchema(String schema)	When passed the schema name, sets this Connection object to the database schema to access.
String getSchema()	Returns the schema name of the database associated with this Connection object; returns null if no schema is associated with it.

Driver

The first step to communicate with the database is to setup a connection between your application and the database server. Establishing a connection requires understanding the database URL, below is presented the base layout and format of a database connection URL string. The general format is as follows - `jdbc:<subprotocol>:<subname>`

Each of those components means something specific, and carries information for the database connection and driver manager, so it can establish the connection between the database and the java application

- `jdbc` - is the same for all DBMS, this is true only for relational type databases. That part of the connection string is different based on the type of the database provider - nosql, relational and so on
- `<subprotocol>` - differs for each DBMS, it specifies the vendor and database type, this is based on the type of the database, it helps the driver manager to identify what is the database vendor the connection string is describing
- `<subname>` - depends on the database, but its general format is `<server>:<port>/database`. The server depends on the location in which the database is deployed. Each DBMS uses specific `<port>` number (3306 is for example the default one for MySQL). Finally the database name to which to connect to is provided

```
jdbc:postgresql://localhost/test
jdbc:oracle://127.0.0.1:44000/test
jdbc:microsoft:sqlserver://himalaya:1433
```

```
class JdbcExample {
    public static void main(String[] args) {
        // URL points to JDBC protocol: mysql subprotocol;
        // localhost is the address of the server where we installed our
        // DBMS (i.e. on local machine) and 3306 is the port on which
        // we need to contact our DBMS
        String url = "jdbc:mysql://localhost:3306/";
        // we are connecting to the addressBook database we created earlier
        String database = "addressBook";
        // we login as "root" user with password "mysql123"
        String userName = "root";
        String password = "mysql123";
        try (Connection connection = DriverManager.getConnection(url +
           database, userName, password)) {
            System.out.println("Database connection: Successful");
        } catch (Exception e) {
            System.out.println("Database connection: Failed");
            e.printStackTrace();
        }
    }
}
```

The URL indicates that `JDBC` is the protocol and `MySQL` is the sub-protocol; `localhost` is the address of the sever where the database service is running, in this case it is on the local machine, but it can be any valid IP or FQDN. The connection object is obtained by invoking the `DriverManager.getConnection` method, the method expects the URL of the database along with a database name username and password. The connection has to be closed, before exiting the program.

In this example a try-with-resources is statement is used; hence the close method for the connection is called automatically. If anything goes wrong an exception will be thrown. In that case the program prints the exception's stack trace

If the program above is to be run, it will print out an exception, this is because the connection will fail due to missing drivers, by default the `JVM` does not ship with any drivers, for any database vendors, in this case a driver for connecting to the target type of database is required - `MySQL`. To do this the driver has to be downloaded from the official vendor distributor, usually those are jar files, which have to be added to the `classpath` when compiling / running the application. This can be done by appending the full path of the location of the jar with drivers to the `-cp` argument - `javac -cp .:path/to/mysql-connector-java-8.x.x.jar JdbcExample.java` and when running it `java -cp .:path/to/mysql-connector-java-8.x.x.jar JdbcExample`.

`Usually when a connection to the database fails and throws an exception, it is rarely possible to do something about it, these errors often happen to be unrecoverable, even though those are checked exception, the only thing that can be really done is log the exception and gracefully notify the user of the issue. Exceptions often will occur if the host is not reachable, the credentials are wrong or the database is protected in unexpected ways, all of which can not recover during the run-time of the application`

Manager

The `DriverManager` class helps establish the connection between the program and the `JDBC` drivers. This class also keeps track of different data sources and `JDBC` drivers. Hence, there is no need to explicitly load drivers: `DriverManager` searches for suitable driver and if found automatically loads it when the `getConnection`

method is called. The driver manager also manages multiple concurrent drivers connected to their respective data sources. `Connection connection = DriverManager.getConnection(url + database, userName, password)`

```
String url = "jdbc:mysql://localhost:3306/";
Driver driver = DriverManager.getDriver(url);
System.out.println(driver.getClass().getName());
```

The code segment above obtains the driver for the specified connection string, in this case it prints out the following text - `com.mysql.jdbc.Driver` this is the fully qualified name of the `MySQL JDBC` driver and `DriverManager` was able to load it. From this Driver object, one can establish a connection by calling the `connect` method and passing in the database URL and the optional Properties file reference - `Connection connection = driver.connect(url);` The properties file can contain the username and password, and other additional details

Querying

Once a connection is established to the desired database, one can perform various operations on it. Common operations are known by the acronym CRUD (create, read, update delete).

Statement

The statement is an SQL statement that can be used to communicate an SQL statement to the database, and receive results from the database. The statement interface stays atop the `PreparedStatement` and `CallableStatement` interfaces.

- `Statement` - sends a basic SQL statement to the database without any parameters for typical uses, one needs to use this interface. It can be created using the `createStatement` method in the Connection interface.
- `PreparedStatement` - represents a pre-compiled statement that can be customized using IN parameters. Usually it is more efficient than a raw Statement object; hence it is used to improve performance, especially if an SQL statement is executed many times. Instance of this `prepareStatement` can be obtained by calling the `preparedStatement` method on the Connection interface
- `CallableStatement` - exectues stored procedures. The `CallableStatement` instances can handle IN as well as OUT and INOUT parameters. To get an instance of it simply call the `prepareCall` method on the Connection interface.

Once the instance of the Statement is created, statement is ready to be executed. The statement interface provides three execute methods: executeQuery(), executeUpdate() and execute() - it is usually based on what type of Statement it is, to invoke the relevant method

- `SELECT` - if the type of the statement is select, then the `executeQuery` method has to be invoked, since it has no side effects, and is meant only to retrieve data, it is not destructive in nature. It returns a `ResultSet`.
- `INSERT / UPDATE or DELETE` - if the type of the statement is destructive in nature, or mutates the state of the database, then one must use the `executeUpdate` method, it returns an integer reflecting the updated number of rows.
- `unknown` - in case the statement is not of known type, i.e. the instance was not produced by the caller of the execute methods, one may use the execute method, which may return multiple `ResultSet` or multiple update counts, or a combination of both.

ResultSet

Relational databases contain tables. Each table has a set of attributes. The tabular nature of a relational database is returned as `ResultSet`. A `ResultSet` is a table with column headings and associated values requested by the query. A `ResultSet` maintains a cursor pointing to the current row. Only one row can be read at a time, so the cursor position must be changed in order to fetch other rows. Initially the cursor is set to just before the first row. One needs to call the `next` method to advance the cursor to the actual first row first. The method returns a boolean value ; hence it can be used in a while loop to iterate over the entire `ResultSet`.

Method	Description
void beforeFirst()	Sets the cursor just before the first row in the resultset.
void afterLast()	Sets the cursor just after the last row of the resultset.
boolean absolute(int rowNumber)	Sets the cursor to the requested row number (absolute position in the table-not relative to the current position).
boolean relative(int rowNumber)	Sets the cursor to the requested row number relative to the current position. rowNumber can be a positive or negative value: a positive value moves forward, and a negative value moves backward relative to the current position.
boolean next()	Sets the cursor to the next row of the resultset.
boolean previous()	Sets the cursor to the previous row of the resultset.

Extraction

Now that all the necessary interface have been visited and shown, to execute a simple SQL statement query on the database - `Connection, Statement and ResultSet`. The general high level overview of the steps which need to be taken to extract information from the database. First obtain a `Connection` object, then create a `Statement` which is used to create obtain the `ResultSet`. The general pattern of actions remains the same in all database working/related applications

```
public class DbConnector {
    public static Connection connectToDb() throws SQLException {
        String url = "jdbc:mysql://localhost:3306/";
        String database = "addressBook";
        String userName = "root";
        String password = "mysql123";
        return DriverManager.getConnection(url + database, userName,
           password);
    }
}

class DbQuery {
    public static void main(String[] args) {
        // Get connection, execute query, get the result set
        // and print the entries from the result rest
        try (Connection connection = DbConnector.connectToDb();
                Statement statement = connection.createStatement();
```

```
                ResultSet resultSet = statement.executeQuery("SELECT *
                   FROM contact"))
            {
            System.out.println("ID \tfName \tlName \temail \t\tphoneNo");
            while (resultSet.next()) {
                System.out.println(resultSet.getInt("id") + "\t"
                    + resultSet.getString("firstName") + "\t"
                    + resultSet.getString("lastName") + "\t"
                    + resultSet.getString("email") + "\t"
                    + resultSet.getString("phoneNo"));
            }
        }
        catch (SQLException sqle) {
            sqle.printStackTrace();
            System.exit(-1);
        }
    }
}
```

In the example above, the column names are being used to extract the information from the query, however indices instead can be used too. Remember that the indices are defined based on the query, not based on the absolute number of columns in the table being queried. In this case the select statement, selects all columns, therefore the table and query both have the same order and number of columns, however that is not always the case. One can use the following snippet to extract the column information from the Statement instead

```
while (resultSet.next()) {
    System.out.println(resultSet.getInt(1)
        + "\t" + resultSet.getString(2)
        + "\t" + resultSet.getString(3)
        + "\t" + resultSet.getString(4)
        + "\t" + resultSet.getString(5));
}
```

`Take a good note at the indices, used to extract the information from the result set object, the column index in the ResultSet object starts from 1, not 0.`

While referring to columns by column index, if one refers to a column by an index that is more than the total number of columns, an exception will be thrown. For instance, if another call is added to extract `resultSet.getString(6)` - this will produce `SQLException - Column Index out of range 6 > 5.`. This is different from accessing rows outside of the range with absolute(int) or relative(int) which simply wrap around the result set instead.

In this case both the data type and number of columns and names are known, however what if none of this information is present. In that case one must use the `getMetaData` method, on the `ResultSet`, object that returns a `ResultMetaData` object; on that object one can then use the `getColumnCount` method to get the column count. When the data type of a column entry is not known, the `getObject` method can be used on the `ResultSet`.

```
int numOfColumns = resultSet.getMetaData().getColumnCount();
while (resultSet.next()) {
    // remember that the column index starts from 1 not 0
    for(int i = 1; i <= numOfColumns; i++) {
```

```
        // since we do not know the data type of the column, we use
           getObject()
        System.out.print(resultSet.getObject(i) + "\t");
    }
}
```

The example above shows how to fetch and print out all columns and the information inside a given ResultSet instance without having to worry about the data type of the target object or column. This is often not the case since the columns are known at the point of writing the statement and query but sometimes it is a good idea to know that the meta data for the current Result set is available

```
try (Connection connection = DbConnector.connectToDb();
        Statement statement = connection.createStatement();
        ResultSet resultset = statement.executeQuery("SELECT firstName,
           email FROM contact WHERE firstName=\"Michael\""))
{
    System.out.println("fName \temail");
    while (resultset.next()){
        System.out.println(resultset.getString("firstName") + "\t"
           resultset.getString("email"));
    }
} catch (SQLException e) {
    e.printStackTrace();
    System.exit(-1);
}
```

In the example above, a similar get query is used, to extract all matching entries, where the first name equals Michael, but note that only two of the total columns are extracted, the first name and the email. In this case had we decided to use indices, the two allowed indices would be 1 and 2 since those are the only columns being extracted from this query.

Updating

Updating the database is done in one of two ways. One can use regular SQL queries to update the database directly, or one can fetch the result-set object using an SQL query and then change it and the database through the interface of the `ResultSet` itself. Both methods are possible and supported, it usually depends on the use case or application

In order to modify the `resultset` and the database the `ResultSet` class provides a set of update methods for each data type. There are also other supporting methods, such as `updateRow` and `deleteRow` to make the task simpler.

```
try (Connection connection = DbConnector.connectToDb();
        Statement statement =
           connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
           ResultSet.CONCUR_UPDATABLE);
        ResultSet resultSet = statement.executeQuery("SELECT * FROM
           contact WHERE firstName=\"Michael\""))
{
    // first fetch the data and display it before the update operation
    System.out.println("Before the update");
    System.out.println("id \tfName \tlName \temail \t\tphoneNo");
    while (resultSet.next()) {
```

```
        System.out.println(resultSet.getInt("id") + "\t"
            + resultSet.getString("firstName") + "\t"
            + resultSet.getString("lastName") + "\t"
            + resultSet.getString("email") + "\t"
            + resultSet.getString("phoneNo"));
    }
    // now update the resultset and display the modified data
    resultSet.absolute(1);
    resultSet.updateString("phoneNo", "+919976543210");
    // reflect those changes back to the database
    // by calling updateRow() method
    resultSet.updateRow();

    System.out.println("After the update");
    System.out.println("id \tfName \tlName \temail \t\tphoneNo");
    resultSet.beforeFirst();
    while (resultSet.next()) {
        System.out.println(resultSet.getInt("id") + "\t"
            + resultSet.getString("firstName") + "\t"
            + resultSet.getString("lastName") + "\t"
            + resultSet.getString("email") + "\t"
            + resultSet.getString("phoneNo"));
    }
} catch (SQLException e) {
    e.printStackTrace();
    System.exit(-1);
}
```

The example above is pretty straightforward, the result set is obtained, printed out, then the cursor is set to the first entry, the `phoneNo` for that entry is updated to a new value, and the result set is set back to the first entry and printed out in full. Note that an `updateRow` method is being called, this is what essentially `commits` the changes to the row, otherwise all changes done to a given row stay in-memory, not actually written out to the database.

`Two major things - first to make a ResultSet update-able one must use the correct form of the create statement, passing in TYPE_SCROLL_SENSITIVE and CONCUR_UPDATABLE, second major point, call updateRow, after making changes to a given row in the result set in order to actually persist the changes in the database`

Inserting

Inserting a row is very much similar to updating a row in a `ResultSet`, the difference is that when a new row is to be inserted the `ResultSet` cursor has to be moved in an appropriate position, which is the special `insert row`, then one can simply call the update methods that were already used to set the values for the given columns for that new row.

```
try (Connection connection = DbConnector.connectToDb();
        Statement statement = connection.createStatement(
           ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
        ResultSet resultSet = statement.executeQuery("SELECT * FROM
           contact"))
{
```

```
    System.out.println("Before the insert");
    System.out.println("id \tfName \tlName \temail \t\tphoneNo");
    while (resultSet.next()){
        System.out.println(resultSet.getInt("id") + "\t"
            + resultSet.getString("firstName") + "\t"
            + resultSet.getString("lastName") + "\t"
            + resultSet.getString("email") + "\t"
            + resultSet.getString("phoneNo"));
    }

    // make sure to first move the cursor of the result set to a valid
       insert row
    resultSet.moveToInsertRow();
    // update all the columns for the new row to be inserted
    resultSet.updateString("firstName", "John");
    resultSet.updateString("lastName", "K.");
    resultSet.updateString("email", "john@abc.com");
    resultSet.updateString("phoneNo", "+19753186420");
    // commit the new row to the database
    resultSet.insertRow();

    System.out.println("After the insert");
    System.out.println("id \tfName \tlName \temail \t\tphoneNo");
    resultSet.beforeFirst();
    while (resultSet.next()){
        System.out.println(resultSet.getInt("id") + "\t"
            + resultSet.getString("firstName") + "\t"
            + resultSet.getString("lastName") + "\t"
            + resultSet.getString("email") + "\t"
            + resultSet.getString("phoneNo"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

Note that similarly to the `updateRow` method, there is a call to `insertRow` method, which makes sure that the changes are actually committed to the database. One very important thing here is that the correct type has to be provided for each value / each column. Also a column can not be left blank (without any value) otherwise an `SQLException` will be thrown.

Deleting

Deleting follows the pattern of insert and update, as in that a special method has to be called to actually perform the operation on the database itself, to commit the changes that is. For deleting that is the method called `deleteRow`. Since unlike insert or update, there is nothing more to be done, no columns to be updated or inserted that is the only thing to do in order to commit the changes to the database.

```
try (Connection connection = DbConnector.connectToDb();
        Statement statement = connection.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
        ResultSet resultSet1 = statement.executeQuery("SELECT * FROM
           contact WHERE firstName=\"John\""))
```

```
{
    if(resultSet1.next()){
        // delete the first row
        resultSet1.deleteRow();
    }
    resultSet1.close();
    // now fetch again from the database
    try (ResultSet resultSet2 = statement.executeQuery("SELECT * FROM
       contact")) {
        System.out.println("After the deletion");
        System.out.println("id \tfName \tlName \temail \t\tphoneNo");
        while (resultSet2.next()){
            System.out.println(resultSet2.getInt("id") + "\t"
                + resultSet2.getString("firstName") + "\t"
                + resultSet2.getString("lastName") + "\t"
                + resultSet2.getString("email") + "\t"
                + resultSet2.getString("phoneNo"));
        }
    }
} catch (SQLException e) {
    e.printStackTrace();
    System.exit(-1);
}
```

Creation

It is also possible to run update statements from the statement itself, above it was demonstrated how a result set can be used to modify the database, here it is demonstrated how an `executeUpdate` can be used to alter the state of the database instead, in this case creating a table.

```
try (Connection connection = DbConnector.connectToDb(); Statement
   statement = connection.createStatement()) {
    // use CREATE TABLE SQL statement to create table familyGroup
    statement.executeUpdate("CREATE TABLE familyGroup (id int not null
       auto_increment, nickName varchar(30) not null, primary key(id));");
    System.out.println("Table created successfully");
}
catch (SQLException sqle) {
    sqle.printStackTrace();
    System.exit(-1);
}
```

The program would print that the table was created successfully in case no other abnormal conditions, such as inaccessible database or the likes. Note that in this case the `executeUpdate` is used, since the statement executes and update action on the database. If the statement itself was incorrect, has some sort of SQL syntax errors, the `MySQLSyntaxException` will be risen. Passing a correct statement is the responsibility of the application developer

Caveats

There are certain caveats that are discussed below which have to be mentioned to properly keep in mind some details regarding the behavior of the `Statement` and `ResultSet` objects when working with them. These

points encompass the sections already discussed above

- The boolean absolute(int) method of `ResultSet` moves the cursor to the passed row number that `ResultSet` object, if the row number is positive it moves to that position from the beginning of the `ResultSet` object; if the row number is negative, if moves to that position from the end of the `ResultSet` object. Assume that there are 10 entries in the `ResultSet` Calling absolute(3) moves the cursor to the third row. Calling absolute(-2) moves the cursor to the eighth row. If out of range values are given, the cursor moves to either the start or the beginning of the `ResultSet`
- In a `ResultSet` object, calling absolute(1) is equivalent to calling first() and same for absolute(-1) which is equivalent to calling last()
- One can use a column name or column index with `ResultSet` methods. The index corresponds to the index of the `ResultSet` object not the column number in the database table.
- A Statement object closes the current `ResultSet`, if the statement object is closed itself, re-executed or made to retrieve the next set of results. That means that it is not required to call close() method on the `ResultSet` since any `ResultSet` instance obtained from a Statement is bound to that Statement object instance in the first place.
- Think of a case in which one has two columns in a `ResultSet` object with the same name. How would one retrieve the associate values using the column name ? If the column name is used to retrieve the value, it always points to the first column that matches the name. Hence a column index has to be used instead
- One can use the column name of a `ResultSet` object without worrying about the casing of the column name using - `getXXX()` methods such as - `getString`, `getObject` and so on, methods accept case-insensitive column names to retrieve the associated value
- Note that `PreparedStatement` interface inherits from Statement. However `PreparedStatement` overrides all flavors of execute methods. For instance the behavior of `executeUpdate()` may be different from its base method.
- Any update may be canceled using the method `cancelRowUpdates()`, However this method must be called before calling `updateRow()`. In all other cases, it has no impact on the row.
- While connecting to the database, a correct username and password have to be specified, otherwise an `SQLException` will be thrown
- The calls to `insertRow` and `updateRow` or `deleteRow` act on the current row selected by the cursor for the current `ResultSet`, in other words, the last time the cursor was moved either with `moveToInsertRow` or absolute or relative, and any other methods which move the cursor.

Summary

Identify the components required to connect to a database

- `JDBC` hides the heterogeneity of all the `DBMSs` and offers a single set of `APIs` to interact with all types of databases. The complexity of heterogeneous interactions is delegated to the `JDBC` driver manager and `JDBC` drivers.
- The `getConnection()` method in the `DriverManager` class takes three arguments: a URL string, a username string, and a password string.
- The syntax of the URL (which needs to be specified to get the Connection object) is `jdbc:<subprotocol>:<subr`

- If the JDBC API is not able to locate the JDBC driver, it throws a SQLException. If jars for the drivers are available, they need to be included in the classpath to enable the JDBC API to locate the driver.

Describe the interfaces that make up the core of the JDBC

- The java.sql.Connection interface provides a channel through which the application and the database communicate.
- JDBC supports two classes for querying and updating: Statement and Resultset.
- A Statement is a SQL statement that can be used to communicate a SQL statement to the connected database and receive results from the database. There are three types of Statements:
- Statement: Sends a SQL statement to the database without any parameters
- PreparedStatement: Represents a pre-compiled SQL statement that can be customized using IN parameters
- CallableStatement: Executes stored procedures; can handle IN as well as OUT and INOUT parameters
- A resultset is a table with column heading and associated values requested by the query.

Submit queries and read results from the database

- A ResultSet object maintains a cursor pointing to the current row. Initially, the cursor is set to just before the first row; calling the next() method advances the cursor position by one row.
- The column index in the ResultSet object starts from 1 (not from 0).
- You need to call updateRow() after modifying the row contents in a resultset; otherwise, changes made to the ResultSet object are lost.
- You can use a try-with-resources statement to close resources (Connection, ResultSet, and Statement) automatically. It is not an error if a ResultSet is closed multiple times, even before the try-with-resources statement, has closed it at the end of the resource try-block, if the ResultSet is no longer required then it can be as closed as soon as that.