Module 7 – Java – RDBMS & Database Programming with JDBC

---

1. Introduction to JDBC

Theory

- **What is JDBC?** JDBC stands for **Java Database Connectivity**. It is a standard Java API (part of the Java SE platform) that allows Java applications to interact with various relational databases. Think of it as a universal translator: your Java code speaks "Java," and the database speaks "SQL." JDBC sits in the middle so you don't have to write different code for every different database.
- **Importance of JDBC:** Without JDBC, developers would have to learn the specific proprietary API of every database (Oracle, MySQL, SQL Server). JDBC provides a **standardized way** to connect to a database, execute SQL statements, and process results. This makes your application "Database Independent"—you can switch from MySQL to Oracle by changing just a few lines of configuration rather than rewriting your entire data layer.
- **JDBC Architecture:** The architecture is divided into two layers:
    - **JDBC API:** This provides the application-to-JDBC Manager connection. It uses the `java.sql` package.
    - **JDBC Driver API:** This supports the JDBC Manager-to-Driver connection.
    - **Driver Manager:** The backbone that matches Java applications to the correct database driver.
    - **Driver:** The actual software that communicates with the database server.
    - **Connection:** A session between the Java app and the DB.
    - **Statement:** The vehicle used to carry your SQL commands to the DB.
    - **ResultSet:** The object that holds the data returned from a `SELECT` query.

Lab Exercise

To connect to MySQL, you must first ensure you have the `mysql-connector-java` JAR file in your classpath.

```
Java

import java.sql.*;

public class FirstConnection {
    public static void main(String[] args) {
        try {
            // 1. Loading the driver (Registration)
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
            // 2. Establishing the connection
            String url = "jdbc:mysql://localhost:3306/your_database_name";
            Connection con = DriverManager.getConnection(url, "root", "your_password");

            if (con != null) {
                System.out.println("Success: Connection established to MySQL!");
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

---

2. JDBC Driver Types

Theory

There are four types of drivers used to connect Java to a database:

1. **Type 1: JDBC-ODBC Bridge:** It uses an ODBC driver to connect. It is slow and requires ODBC to be installed on the client machine. (Now obsolete).
2. **Type 2: Native-API Driver:** Converts JDBC calls into client-side calls for the specific database (like Oracle or MySQL libraries). Requires client-side library installation.
3. **Type 3: Network Protocol Driver:** Uses a middleware server that converts JDBC calls to the database protocol. It is flexible because it doesn't require client-side libraries.
4. **Type 4: Thin Driver (The Standard):** This is a pure Java driver that communicates directly with the database using network protocols. It is the most preferred because it is fast and requires no client-side installation.

Lab Exercise

- **Identification:** In modern MySQL development, we use the **Type 4 Thin Driver** (`com.mysql.cj.jdbc.Driver`).
- **Best Driver:** The Type 4 driver is considered the best for most environments because it is platform-independent, offers high performance, and doesn't require extra software (like ODBC) to be configured on the user's computer.

---

3. Steps for Creating JDBC Connections

Theory

To get data from a database into your Java app, you must follow these 7 "Golden Steps":

1. **Import Packages:** Include `java.sql.*` to access JDBC classes.
2. **Register Driver:** Tell Java which database driver you are using via `Class.forName()`.

3. **Open Connection:** Use `DriverManager.getConnection()` to create a bridge to the DB.
4. **Create Statement:** Build a "Statement" object to hold your SQL query.
5. **Execute Query:** Send the SQL to the database (e.g., `executeQuery()` for SELECT).
6. **Process ResultSet:** Use a loop to read the rows of data returned.
7. **Close Connection:** Always close the connection to free up database resources.

Lab Exercise

```
Java

// Logic for a confirmation program
Connection con = DriverManager.getConnection(url, user, password);
if(!con.isClosed()) {
    System.out.println("JDBC Connection is Active and Healthy!");
}
```

---

4. Types of JDBC Statements

Theory

- **Statement:** Used for general-purpose access. Best for static SQL queries (queries that don't change).
- **PreparedStatement:** Used when you need to provide parameters (like `WHERE id = ?`). It is **precompiled**, making it much faster for repeated execution and secure against **SQL Injection attacks**.
- **CallableStatement:** The only way to execute **Stored Procedures** (logic saved inside the database itself).

Lab Exercise

Using Statement:

```
Java

Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO users VALUES (1, 'John', 'Doe')");
```

Using PreparedStatement (Better approach):

```
Java

PreparedStatement pstmt = con.prepareStatement("INSERT INTO users (fname) VALUES (?)");
pstmt.setString(1, "Jane");
pstmt.executeUpdate();
```

---

5. JDBC CRUD Operations

Theory

CRUD stands for **Create, Read, Update, and Delete**. These are the four basic functions of persistent storage.

- **Insert:** Adding new data.
- **Update:** Changing existing data.
- **Select:** Viewing/Retrieving data.
- **Delete:** Removing data.

Lab Exercise

A typical program would use `executeUpdate()` for Insert, Update, and Delete (as they change the data) and `executeQuery()` for Select (as it returns data).

---

6. ResultSet Interface

Theory

The `ResultSet` is like a virtual table. When you run a `SELECT` query, the database sends back rows of data, and Java catches them in a `ResultSet` object.

- **Navigating:** It has a "cursor" that starts before the first row.
    - `next()`: Moves to the next row.
    - `previous()`: Moves to the previous row (requires scrollable ResultSet).
    - `first()`/`last()`: Jumps to the beginning or end.

---

7. Database & ResultSet Metadata

Theory

- **DatabaseMetaData:** Information **about the database** (e.g., What is the version? What are the table names? Does it support transactions?).
- **ResultSetMetaData:** Information **about the query results** (e.g., How many columns were returned? What is the name of column #2? Is it a String or an Integer?).

Lab Exercise

```
Java

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
System.out.println("This table has " + rsmd.getColumnCount() + " columns.");
```

---

8. Practical Example: Swing GUI & Stored Procedures

Theory

- **Swing Integration:** In a real application, users don't type in a console. You use `JTextFields` for names/emails and `JButtons` for "Save" or "Delete." When a button is clicked, the code inside the `ActionListener` picks up the text from the fields and runs a `PreparedStatement`.
- **CallableStatement (Stored Procedures):** These are like "functions" that live in the database.
    - **IN Parameter:** You send data to the procedure (e.g., `EmployeeID`).
    - **OUT Parameter:** The procedure sends data back to Java (e.g., `TotalSalary`).

Lab Exercise (CallableStatement)

Java

```
CallableStatement cstmt = con.prepareCall("{call GetName(?, ?)}");
cstmt.setInt(1, 101); // Setting IN parameter
cstmt.registerOutParameter(2, Types.VARCHAR); // Registering OUT parameter
cstmt.execute();
System.out.println("Result from DB: " + cstmt.getString(2));
```

---

9. Practical SQL Query Examples

Theory

In JDBC, SQL queries are the commands sent to the database to perform actions. While you can use a simple `Statement`, the industry standard is to use a `PreparedStatement`. This is because it uses placeholders (?), which allow the database to "pre-compile" the query. This results in faster execution and protects the system from SQL Injection, a common security vulnerability where hackers try to manipulate queries.

Lab Exercise: Implementation

Below is how you implement the four primary SQL operations in Java:

- **Inserting a Record:**
- **Java**

```
String sql = "INSERT INTO students (fname, lname, email) VALUES (?, ?, ?)";
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Rahul");
pstmt.setString(2, "Sharma");
```

```
pstmt.setString(3, "rahul@example.com");
pstmt.executeUpdate(); // Returns the number of rows affected
```

- **Updating a Record:**

```
String sql = "UPDATE students SET email = ? WHERE id = ?";
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "new_email@example.com");
pstmt.setInt(2, 101);
pstmt.executeUpdate();
```

- **Selecting Records:**

```
String sql = "SELECT * FROM students WHERE lname = ?";
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Sharma");
ResultSet rs = pstmt.executeQuery(); // Use executeQuery for SELECT
while(rs.next()) {
   System.out.println(rs.getString("fname"));
}
```

- **Deleting a Record:**

```
String sql = "DELETE FROM students WHERE id = ?";
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setInt(1, 101);
pstmt.executeUpdate();
```

---

10. Practical Example 1: Swing GUI for CRUD Operations

Theory

A Graphical User Interface (GUI) makes your database application user-friendly. Instead of typing commands in a terminal, users interact with text boxes and buttons.

- **Java Swing: This is a part of Java Foundation Classes (JFC) used to create window-based applications.**
- **Integration Logic: To connect Swing to JDBC, you follow an Event-Driven model.**
    1. **The user enters data into `JTextField` components.**
    2. **The user clicks a `JButton` (e.g., "Submit").**
    3. **The `ActionListener` for that button is triggered.**
    4. **Inside the listener, you extract the text from the fields using `.getText()` and pass those strings into a JDBC `PreparedStatement`.**

Lab Exercise: Step-by-Step Logic

1. **Design the Frame: Create a `JFrame` and add `JLabels` and `JTextFields` for `id`, `fname`, `lname`, and `email`.**
2. **Add Buttons: Create buttons for `Insert`, `Update`, `Delete`, and `Select`.**
3. **Database Connection: Create a global `Connection` object that opens when the application starts.**
4. **Button Implementation:**
    - **For the Insert Button, the code would look like: `pstmt.setString(1, txtFname.getText());`.**
    - **For the Select Button, the results from the `ResultSet` should be put back into the text fields: `txtEmail.setText(rs.getString("email"));`.**

---

11. Practical Example 2: Callable Statement (IN and OUT Parameters)

Theory

A `CallableStatement` is used specifically to call Stored Procedures. A stored procedure is like a function that is written in SQL and saved inside the database itself. This is highly efficient for complex logic because the processing happens on the database server, not on your computer.

- **IN Parameters: Values you send from Java to the database (e.g., searching for an Employee ID).**
- **OUT Parameters: Values the database sends back to Java after the procedure runs (e.g., the Employee's Salary).**

Lab Exercise: Full Implementation

Step 1: Create a Stored Procedure in MySQL

This procedure takes an ID and finds the full name.

```
SQL
```

```
CREATE PROCEDURE GetStudentFullName(IN sid INT, OUT full_name VARCHAR(100))
BEGIN
    SELECT CONCAT(fname, ' ', lname) INTO full_name
    FROM students WHERE id = sid;
END
```

Step 2: Write the Java Code

Java

```
// 1. Prepare the call using curly braces syntax
CallableStatement cstmt = con.prepareCall("{call GetStudentFullName(?, ?)}");

// 2. Set the IN parameter (Value sent to DB)
cstmt.setInt(1, 105);

// 3. Register the OUT parameter (Tell Java to expect a String back)
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);

// 4. Execute the procedure
cstmt.execute();

// 5. Retrieve the value from the OUT parameter
String studentName = cstmt.getString(2);
System.out.println("The Student's Full Name is: " + studentName);
```
