Module 7 – Java – RDBMS & Database Programming with JDBC

1. Introduction to JDBC

What is JDBC (Java Database Connectivity)?

JDBC, which stands for Java Database Connectivity, is a standard Java API (Application Programming Interface) that enables Java applications to interact with relational databases. It provides a set of classes and interfaces that allow developers to perform database operations—such as connecting to a database, executing SQL queries, and processing results—in a way that is independent of the specific database system being used.

Importance of JDBC in Java Programming

The primary importance of JDBC is that it provides a uniform interface for accessing a wide variety of databases. Before JDBC, developers had to write database-specific code to connect their Java applications to different database systems. With JDBC, you can write a single program that can connect to Oracle, MySQL, PostgreSQL, or any other database, simply by changing the JDBC driver and connection URL. This makes Java applications more portable and easier to maintain.

JDBC Architecture

The JDBC architecture consists of two main layers: the JDBC API and the JDBC Driver Manager. Its key components are:

- **Driver Manager**: This is the central component of the JDBC API. It manages a list of database drivers and matches connection requests from the Java application with the appropriate driver. When an application requests a connection, the DriverManager scans the registered drivers to find one that can handle the given database URL.
- **Driver**: A JDBC driver is a software component that enables a Java application to interact with a specific database. Each database system (like MySQL, Oracle, etc.) requires its own driver. The driver is responsible for translating the standard JDBC calls into the database-specific protocol.
- **Connection**: A Connection object represents a session with a specific database. All communication with the database happens through this object. You use it to create statements, manage transactions, and get metadata about the database.
- **Statement**: A Statement object is used to execute a static SQL query and return the results it produces. There are different types of statements, such

as Statement, PreparedStatement, and CallableStatement, each serving a different purpose.

- **ResultSet**: A ResultSet object holds the data retrieved from a database after you execute a SELECT query. It provides methods to iterate through the rows of data and retrieve the values from each column.

2. JDBC Driver Types

There are four main types of JDBC drivers:

- **Type 1: JDBC-ODBC Bridge Driver**: This driver translates JDBC calls into ODBC (Open Database Connectivity) calls, which are then handled by an ODBC driver to connect to the database. This driver was included in early versions of Java but is now considered obsolete due to its dependency on ODBC and lower performance.
- **Type 2: Native-API Driver**: This driver converts JDBC calls into native API calls for the database, using a client-side library. It offers better performance than the Type 1 driver but requires the native database library to be installed on the client machine, making it platform-dependent.
- **Type 3: Network Protocol Driver**: This is a middleware-based driver that sends JDBC calls to a server, which then translates them into the database-specific protocol. It is a flexible solution as it does not require any vendor-specific library on the client side, but it adds an extra network hop.
- **Type 4: Thin Driver (Pure Java Driver)**: This driver converts JDBC calls directly into the network protocol used by the database. It is written entirely in Java and is platform-independent, making it the most commonly used driver type. The MySQL Connector/J is a Type 4 driver.

For most modern Java applications connecting to databases like MySQL, the **Type 4 Thin Driver** is the best choice. It is platform-independent, does not require any native libraries on the client machine, and connects directly to the database server, offering excellent performance and portability.

3. Steps for Creating JDBC Connections

Establishing a JDBC connection involves a sequence of well-defined steps :

1. **Import the JDBC packages**: Begin by importing the necessary Java SQL packages, primarily java.sql.*.
2. **Register the JDBC driver**: Load the database-specific JDBC driver into memory. This is typically done using the Class.forName() method (e.g., Class.forName("com.mysql.cj.jdbc.Driver");). In modern JDBC (4.0 and later), this step is often automatic as long as the driver's JAR file is on the classpath.

3. **Open a connection**: Use the DriverManager.getConnection() method to establish a connection to the database. This method requires a database URL, username, and password.
4. **Create a statement**: Once a connection is established, create a Statement, PreparedStatement, or CallableStatement object to execute SQL commands.
5. **Execute SQL queries**: Use the statement object to run your SQL queries. For SELECT queries, use executeQuery(), and for INSERT, UPDATE, or DELETE operations, use executeUpdate().
6. **Process the result set**: If your query returns data (i.e., a SELECT statement), the results are stored in a ResultSet object. You can iterate through this object to retrieve and process the data.
7. **Close the connection**: It is crucial to close the Connection, Statement, and ResultSet objects in a finally block to release database resources and prevent memory leaks.

4. Types of JDBC Statements

JDBC provides three main types of statement objects:

- **Statement**: This is used for executing simple, static SQL queries that do not have any parameters. It is suitable for DDL (Data Definition Language) statements like CREATE TABLE. However, it is vulnerable to SQL injection attacks if used with user input.
- **PreparedStatement**: This is used for executing precompiled SQL queries that can contain input parameters (represented by ?). PreparedStatement is generally more efficient than Statement for queries that are executed multiple times, as the SQL is compiled only once. More importantly, it automatically handles the escaping of special characters, which prevents SQL injection attacks, making it the preferred choice for executing DDL, DML, and DQL commands with parameters.
- **CallableStatement**: This is used specifically to execute stored procedures and functions in a database. It allows you to pass IN (input), OUT (output), and INOUT (input/output) parameters to the stored procedure.

Differences: The primary differences lie in their use case, performance, and security. Statement is for static SQL, PreparedStatement is for parameterized queries offering better performance and security, and CallableStatement is exclusively for calling stored procedures.

5. JDBC CRUD Operations

CRUD stands for the four fundamental operations of persistent storage: Create, Read, Update, and Delete.

- **Insert (Create)**: This operation adds a new record (or row) to a database table. In JDBC, this is done using the INSERT SQL statement, typically executed with executeUpdate().
- **Select (Read)**: This operation retrieves records from a database table. The SELECT SQL statement is used, and it's executed with executeQuery(), which returns the data in a ResultSet object.
- **Update**: This operation modifies existing records in a table. The UPDATE SQL statement is used to change data in one or more rows, executed with executeUpdate().
- **Delete**: This operation removes records from a database table. The DELETE SQL statement is used to remove one or more rows, also executed with executeUpdate().

6. ResultSet Interface

What is ResultSet in JDBC?
A ResultSet is a Java object that contains the results of executing an SQL query. It acts as a cursor, pointing to one row of data within the result set. Initially, the cursor is positioned before the first row.

Navigating and Working with ResultSet
You can navigate through the ResultSet using several methods:

- **next()**: This is the most common method used. It moves the cursor to the next row and returns true if there is a next row, or false if all rows have been processed. It is typically used in a while loop to iterate through the results.
- **previous()**: Moves the cursor to the previous row.
- **first()**: Moves the cursor to the first row in the ResultSet.
- **last()** Moves the cursor to the last row.

To retrieve data from the current row, you use getter methods like getString(), getInt(), getDouble(), etc. You can specify the column by its index (starting from 1) or by its name (e.g., rs.getString("column_name")).

7. Database Metadata

What is DatabaseMetaData?
DatabaseMetaData is an interface that provides comprehensive information about the

database as a whole. You can obtain a DatabaseMetaData object from an active Connection object.

Importance and Methods

This metadata is important for writing database-agnostic code, where the application needs to adapt its behavior based on the capabilities of the underlying database. Key methods include:

- **getDatabaseProductName()**: Returns the name of the database (e.g., "MySQL").
- **getDatabaseProductVersion()**: Returns the version of the database.
- **getDriverName()**: Returns the name of the JDBC driver.
- **getTables()**: Returns a ResultSet containing a list of tables in the database.
- **supportsBatchUpdates()**: Checks if the database supports batch updates.

8. ResultSet Metadata

What is ResultSetMetaData?

ResultSetMetaData is an interface that provides information about the columns of a ResultSet object. You can get a ResultSetMetaData object from a ResultSet.

Importance and Methods

This metadata is useful for analyzing the structure of a query result, especially when the columns are not known at compile time. It allows you to dynamically process the results. Key methods include:

- **getColumnCount()**: Returns the number of columns in the ResultSet.
- **getColumnName(int column)**: Returns the name of the specified column.
- **getColumnTypeName(int column)**: Returns the SQL type name of the column (e.g., "VARCHAR", "INT").
- **getColumnDisplaySize(int column)**: Returns the column's normal maximum width in characters.

10. Practical Example 1: Swing GUI for CRUD Operations

Introduction to Java Swing

Java Swing is a GUI (Graphical User Interface) widget toolkit for Java. It is part of Oracle's Java Foundation Classes (JFC) — an API for providing a graphical user interface for Java programs. Swing provides a rich set of components like buttons, text fields, tables, and labels, which can be used to build desktop applications.

Integrating Swing with JDBC

To integrate Swing with JDBC for CRUD operations, you create a GUI where user actions trigger database operations. The typical flow is:

1. **Design the GUI**: Create a window (JFrame) with input fields (JTextField) for data entry and buttons (JButton) for "Insert," "Update," "Select," and "Delete."
2. **Add Event Listeners**: Attach ActionListeners to the buttons.
3. **Implement JDBC Logic**: Inside the actionPerformed method of each listener, write the JDBC code to perform the corresponding CRUD operation. For example, when the "Insert" button is clicked, the application should:
    - Get the text from the input fields.
    - Establish a JDBC connection.
    - Use a PreparedStatement to execute an INSERT query with the data from the text fields.
    - Display a success or failure message to the user (e.g., using a JOptionPane).

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

What is a CallableStatement?

A CallableStatement is a specialized JDBC interface used to execute stored procedures in a database. Stored procedures are precompiled SQL code that can be stored and reused, often improving performance and centralizing business logic at the database level.

Calling Stored Procedures with IN and OUT Parameters

To call a stored procedure, you follow these steps:

1. **Create a Stored Procedure in MySQL**: First, define the procedure in your database. A procedure can accept IN parameters (values passed to it) and return values through OUT parameters.
2. **Prepare the Call in Java**: Use the Connection.prepareCall() method with a string that calls the procedure, using ? as placeholders for parameters (e.g., {call my_procedure(?, ?)}).
3. **Register OUT Parameters**: Before executing the call, you must register any OUT parameters using the registerOutParameter() method, specifying the parameter index and its SQL type (e.g., cs.registerOutParameter(2, java.sql.Types.VARCHAR);).
4. **Set IN Parameters**: Set the values for any IN parameters using the appropriate setter methods (e.g., cs.setInt(1, 123);).
5. **Execute the Statement**: Execute the CallableStatement using execute().

6. **Retrieve OUT Parameters**: After execution, retrieve the values of the OUT parameters using the getter methods (e.g., cs.getString(2);).