Module - 3

Introduction to JDBC

What is JDBC (Java Database Connectivity)?

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with relational databases. It provides a standard interface for connecting to databases, executing SQL queries, and retrieving results.

Key Features:

1. **Database Connection**: JDBC provides a set of interfaces and classes to establish a connection to a database.
2. **SQL Execution**: It allows Java programs to execute SQL commands (e.g., SELECT, INSERT, UPDATE, DELETE) on a database.
3. **Result Handling**: JDBC retrieves results from queries (usually in the form of result sets) and processes them.
4. **Database Independence**: By using JDBC, Java programs can connect to any database that has a JDBC driver, making Java applications database-agnostic.

Importance of JDBC in Java Programming

- Standardized Database Access: JDBC defines a standard way to interact with databases from Java programs.
- Flexibility: It allows interaction with multiple database systems.
- Performance: Supports features like prepared statements and batch processing for optimized performance.
- Transaction and Connection Management: Ensures data consistency and efficient resource use.

JDBCArchitecture: Driver Manager, Driver, Connection, Statement, and ResultSet

Driver Manager

- The Driver Manager is responsible for managing a list of database drivers. It acts as a mediator between the Java application and the database driver.
- Role: The Driver Manager's job is to load the appropriate database driver based on the connection URL specified by the Java application. It can manage multiple drivers and is used to establish a database connection.

Driver

- The Driver is a platform-specific implementation that handles the communication between the Java application and a specific type of database.
- Role: Each database (MySQL, Oracle, PostgreSQL, etc.) has its own JDBC driver that implements the `java.sql.Driver` interface. The driver translates Java calls into database-specific commands and sends them to the database.

Connection

- The Connection object represents an open connection to a specific database. It is the core object used to interact with the database.
- Role: The `Connection` object provides methods to create Statement objects, execute SQL queries, and manage database transactions.

Statement

- The Statement object is used to send SQL queries to the database. It allows the execution of SQL commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Role: The `Statement` object is used to create and execute SQL queries, and it can return a ResultSet (for queries that retrieve data).

ResultSet

- The ResultSet object is used to store the results of a query executed using a `Statement` or `PreparedStatement`. It provides methods to iterate over the result set and retrieve data.
- Role: When a query is executed, the ResultSet holds the data returned by the database. You can iterate through it and extract values from the rows.

JDBC Driver Types

Overview of JDBC Driver Types:

Type 1: JDBC-ODBC Bridge Driver

- **Architecture**: This driver uses ODBC (Open Database Connectivity) to connect to the database. It translates JDBC calls into ODBC calls, which are then passed to the database.
- **How it works**: When the Java application makes a JDBC request, the JDBC-ODBC bridge converts the request into an ODBC call. The ODBC driver then communicates with the database.

Type 2: Native-API Driver

- **Architecture**: This driver uses a database's native client API (for example, Oracle's OCI or Sybase's DB-Library) to connect to the database. It converts JDBC calls into database-specific calls and directly communicates with the database via the database's native protocol.
- **How it works**: JDBC calls are translated into the database's native protocol, and then the database's API is used to communicate with the database server.

Type 3: Network Protocol Driver

- **Architecture**: This driver communicates with the database via a middleware server. The driver translates JDBC calls into a database-independent network protocol, which is then sent to the middleware server. The middleware server in turn communicates with the database using the database's native protocol.
- **How it works**: The driver translates JDBC calls into an intermediate protocol (which is usually a standardized network protocol), and sends these requests to a middleware server. The middleware server then communicates with the database.

Type 4: Thin Driver (Pure Java Driver)

- **Architecture**: This driver is written entirely in Java and directly communicates with the database via its native protocol. It does not require any native libraries or middleware. It is considered the most efficient and flexible JDBC driver.
- **How it works**: JDBC calls are sent directly to the database using the database's native protocol without the need for additional layers such as ODBC or middleware.

Steps for Creating JDBC Connections

Step-by-Step Process to Establish a JDBC Connection:

- **Import the JDBC Packages**
  - The first step is to import the necessary Java classes and packages for JDBC functionality. These classes are part of the `java.sql` package.
  - import java.sql.*;

- **Register the JDBC Driver**
  - **Registering the driver** is necessary to make JDBC aware of the database driver you're using. This step ensures that the appropriate database driver is loaded into memory so that it can be used for creating connections.
  - Class.forName("com.mysql.jdbc.Driver");

- **Open a Connection to the Database**
  - Once the driver is registered, the next step is to establish a **connection** to the database using `DriverManager.getConnection()`.
  - String url = "jdbc:mysql://localhost:3306/mydatabase";
  - Connection con = DriverManager.getConnection(url, user, password);

- **Create a Statement**
  - After establishing the connection, you need to create a **Statement** object. A Statement is used to execute SQL queries against the database.
  - **Statement**: For basic queries that don't require parameters.
    - Statement stmt = conn.createStatement();
  - **PreparedStatement**: For SQL queries with parameters (more efficient and safer from SQL injection).
    - String query = "SELECT * FROM users WHERE username = ?"; PreparedStatement pstmt = conn.prepareStatement(query);
  - **CallableStatement**: For executing stored procedures.

- **Execute SQL Queries**
  - For **SELECT** queries, you use `executeQuery()` to retrieve the results.
    - ResultSet rs = stmt.executeQuery("SELECT * FROM users");
  - For **INSERT**, **UPDATE**, or **DELETE** queries, you use `executeUpdate()` to perform the operations.
    - int rowsAffected = stmt.executeUpdate("INSERT INTO users (username, password) VALUES ('john_doe', 'password123')");

❖ Process the Result Set

- ➢ If your query was a **SELECT** query, you will receive a **ResultSet** object, which contains the data returned by the database.
- ➢ You can use methods like `next()`, `getString()`, `getInt()`, etc., to navigate through and retrieve data from the ResultSet.
- ➢ while (rs.next())

```
 {

int id = rs.getInt("id");

String username = rs.getString("username");

String password = rs.getString("password");

System.out.println(id + ": " + username + " - " + password);

}
```

❖ Close the Connection

- ➢ Once the database operations are complete, it's important to **close** all the resources (Connection, Statement, ResultSet) to avoid memory leaks and free up database resources.

```
// Close the ResultSet

if (rs != null) {

rs.close();

}

// Close the Statement

if (stmt != null) {

stmt.close();

}

// Close the Connection

if (conn != null) {

conn.close();

}
```

Types of JDBC Statements

Overview of JDBC Statements:

1. **Statement:**
   - **1.1.** The `Statement` interface in JDBC is used to execute simple SQL queries against the database. When executing SQL queries that do not require parameters (i.e., static SQL queries), you typically use the `Statement` object.

2. **PreparedStatement:**
   - **2.1.** `PreparedStatement` is an interface in JDBC that provides a way to execute precompiled SQL statements. It is an enhancement over the `Statement` object, and it is used to execute SQL queries or updates that can include parameters. The primary advantage of using `PreparedStatement` over `Statement` is that it allows you to write more secure and efficient code, particularly when handling dynamic values or user inputs.
3. **CallableStatement:**
   - **3.1.** In JDBC, the `CallableStatement` interface is used to execute stored procedures and functions in a database. Stored procedures are precompiled SQL statements or a group of SQL statements that can be executed on the database server. Functions are similar, but they return a single value and can be used in SQL expressions.
   - **3.2.** While `Statement` and `PreparedStatement` are used for executing SQL queries and updates, `CallableStatement` is specifically designed for invoking stored procedures or functions in the database.

JDBC CRUD Operations (Insert, Update, Select, Delete)

- **Insert:**
  - In JDBC, inserting data into a database is typically done using the `PreparedStatement` interface. The `PreparedStatement` is preferred over the `Statement` because it is more secure (protecting against SQL injection) and often more efficient, especially when dealing with dynamic values.
  - String sql = "INSERT INTO Employees (id, name, salary) VALUES (?, ?, ?)";
- **Update:**
  - The `UPDATE` statement in SQL is used to modify existing data in a table. In JDBC, you can use `PreparedStatement` to securely update data in a table with dynamic values.
  - String sql = "UPDATE Employees SET salary = ? WHERE id = ?";
- **Select:**
  - The **`SELECT`** statement in SQL is used to **retrieve records** from one or more tables in a database. It allows you to query specific columns, filter data, sort it, and even combine data from multiple tables.
  - SELECT column1, column2 FROM table_name WHERE condition;
- **Delete:**
  - The **`DELETE`** statement in SQL is used to **remove records** from a table based on a specified condition.
  - DELETE FROM table_name WHERE condition;

ResultSet Interface

- **What is ResultSet in JDBC?**
  - In JDBC (Java Database Connectivity), a `ResultSet` is an interface that represents the result of a query executed against a database. When you run a `SELECT` query, the database returns the data as a result set. The `ResultSet` object allows you to retrieve and navigate through the rows and columns of this data.
  - A `ResultSet` is typically created by calling the **`executeQuery()`** method on a `Statement` or `PreparedStatement` object. The query is usually a **SELECT** statement, and the `ResultSet` will contain the rows and columns that match the query criteria.
  - ResultSet resultSet = statement.executeQuery("SELECT * FROM employees");
- **Navigatingthrough ResultSet (first, last, next, previous)**
  - **First:**
    - Moves the cursor to the first row in the result set. Returns `true` if the cursor was moved to the first row, otherwise `false`.
    - if (resultSet.first()) { // Retrieve data from the first row }
  - **Last:**
    - Moves the cursor to the **last row** in the result set. Returns `true` if the cursor was moved to the last row, otherwise `false`.
    - if (resultSet.last()) { // Retrieve data from the last row }
  - **Next:**
    - Moves the cursor forward by one row. It returns `true` if there is a next row, otherwise `false`.
    - if (resultSet.next()) { // Retrieve data from the current row }
  - **Previous:**
    - Moves the cursor **backward** by one row. Returns `true` if there is a previous row, otherwise `false`.
    - if (resultSet.previous()) { // Retrieve data from the previous row }

Database Metadata

- **What is DatabaseMetaData?**
  - In **JDBC (Java Database Connectivity)**, the `DatabaseMetaData` interface provides information about the **database** itself. It allows you to retrieve metadata (data about the database) and properties that describe the capabilities, structure, and features of the database you're connected to.
  - The `DatabaseMetaData` object is typically obtained from a `Connection` object and provides methods to get details about the database, tables, columns, supported SQL features, and more.
- **Importanceof Database Metadata in JDBC:**
  - **Getting Database and Driver Information**: You can use `DatabaseMetaData` to obtain details like the database product name, version, and supported SQL functions.
  - **Exploring Database Structure**: You can query `DatabaseMetaData` to get a list of tables, columns, and other objects in the database.
  - **Checking Database Capabilities**: You can check if the database supports specific features, such as stored procedures, batch updates, or different types of isolation levels.
- **Methods provided by DatabaseMetaData(getDatabaseProductName, getTables, etc.) :**
  - `getDatabaseProductName()`: Returns the name of the database product.
  - `getDatabaseProductVersion()`: Returns the version of the database product.
  - `getDriverName()`: Returns the name of the JDBC driver.
  - `getDriverVersion()`: Returns the version of the JDBC driver.
  - `supportsTransactions()`: Returns `true` if the database supports transactions.
  - `supportsBatchUpdates()`: Returns `true` if the database supports batch updates.
  - `supportsStoredProcedures()`: Returns `true` if the database supports stored procedures.
  - `getTables()`: Retrieves a list of tables in the database.
  - `getColumns()`: Retrieves the columns for a specific table.
  - `getPrimaryKeys()`: Retrieves the primary key information for a given table.
  - `getSQLKeywords()`: Returns a list of SQL keywords supported by the database.
  - `getStringFunctions()`: Returns the names of SQL string functions supported by the database.
  - `getNumericFunctions()`: Returns the names of SQL numeric functions supported by the database.
  - `getColumnPrivileges()`: Retrieves privileges for columns in a table.
  - `getTablePrivileges()`: Retrieves privileges for tables.

ResultSet Metadata

- **What is ResultSetMetaData?**
  - `ResultSetMetaData` is an interface in **JDBC (Java Database Connectivity)** that provides metadata about the columns in a `ResultSet`. In other words, it allows you to retrieve information about the structure of the result set returned by a database query, such as column names, types, and other properties of the columns (e.g., whether a column is nullable, the column's display size, etc.).
- **Importanceof ResultSet Metadata in analyzing the structure of query results:**
  - **Column Information**: Provides detailed information about the columns in a result set, such as column names, data types, and more.
  - **Dynamic Query Handling**: Useful for applications where the structure of the result set is not known in advance or is variable.
  - **Column Data Types**: Allows you to retrieve the SQL data type of each column (e.g., `VARCHAR`, `INT`, `DATE`).
- **Methods in ResultSetMetaData(getColumnCount, getColumnName, getColumnType):**
  - **getColumnCount:**
    - Description: Returns the number of columns in the result set.
    - Usage: This method is useful to know how many columns are present in the result set**.**
  - **getColumnName:**
    - **Description**: Returns the name of the column at the given index (1-based).
    - **Usage**: Useful to retrieve the column name dynamically.
  - **getColumnType:**
    - **Description**: Returns the SQL data type of the specified column. This is returned as an integer constant corresponding to SQL types (e.g., `Types.VARCHAR`, `Types.INTEGER`, `Types.DATE`).
    - **Usage**: Useful when you need to handle different data types dynamically.

Swing GUI for CRUD Operations

- ❖ **Introduction to Java Swing for GUI development**
  - ➢ **Java Swing** is a powerful set of libraries used to create Graphical User Interfaces (GUIs) in Java applications. It is part of the **Java Foundation Classes (JFC)**, which also includes the **Abstract Window Toolkit (AWT)**. Swing provides a more sophisticated and flexible set of components for building cross-platform graphical interfaces.
  - ➢ Swing is built on top of the AWT but offers more lightweight (pure Java-based) components, making it more portable and customizable. Unlike AWT, which uses native OS components for rendering, Swing renders components using Java code, allowing a uniform appearance across different platforms.
- ❖ **How to integrate Swing components with JDBC for CRUD operations:**
  - ➢ **Set up your database**: You'll need a database to interact with. For this example, let's assume you are using **MySQL** as the database.
  - ➢ **JDBC Setup**: You need to set up the connection to the database, which typically involves loading the driver and creating a `Connection` object to the database.
  - ➢ **Create the Swing GUI**: You'll design a basic Swing interface that will allow users to input data, update records, and view existing data.
  - ➢ **Perform CRUD Operations**:
    - **Create**: Add new records to the database.
    - **Read**: Display records from the database in the Swing interface (e.g., in a table).
    - **Update**: Modify existing records in the database.
    - **Delete**: Remove records from the database.

Callable Statement with IN and OUT Parameters

- **What is a CallableStatement?**
  - In JDBC (Java Database Connectivity), a **`CallableStatement`** is a special type of `Statement` used to execute stored procedures or functions in a relational database. Unlike the regular `Statement` or `PreparedStatement`, which are used for executing regular SQL queries, `CallableStatement` is specifically designed for interacting with **stored procedures** and **functions** that are already defined within the database.
  - Stored procedures are precompiled SQL statements stored in the database, which can be executed by the application when needed. They may accept input parameters (IN), return output parameters (OUT), or both. They can also return result sets or scalar values.
- **How to call stored procedures using CallableStatement in JDBC**
  - **Create a connection to the database**: Establish a connection to the database using `DriverManager.getConnection()`.
  - **Prepare the `CallableStatement`**: Prepare a call to the stored procedure using the `Connection.prepareCall()` method.
  - **Set input parameters (if any)**: Use the `setXXX()` methods to set the input parameters for the stored procedure.
  - **Register output parameters (if any)**: Use the `registerOutParameter()` method to register output parameters.
  - **Execute the statement**: Execute the `CallableStatement` using `execute()` or `executeQuery()`/`executeUpdate()` depending on whether the stored procedure returns a result set or not.
  - **Retrieve output parameters (if any)**: After executing the stored procedure, retrieve the output parameters using the `getXXX()` methods.
  - **Handle multiple result sets (if needed)**: Use `getMoreResults()` to retrieve additional result sets returned by the procedure.
- **Working with IN and OUT parameters in stored procedures**
  - **IN** parameters are used to send data into the procedure, while **OUT** parameters are used to retrieve data from the procedure after execution.
  - Here's a step-by-step guide for working with both **IN** and **OUT** parameters in a stored procedure.
  - **1. IN Parameters:**
    - These are used to pass input values into the stored procedure.
  - **2. OUT Parameters:**
    - These are used to return output values from the stored procedure.