Java Database Connectivity

Comprehensive Guide to JDBC (Java Database Connectivity)

What is JDBC?

JDBC (Java Database Connectivity) is an API (Application Programming Interface) in Java that provides a standard way for Java applications to interact with databases. It allows you to connect Java programs to relational databases and perform CRUD (Create, Read, Update, Delete) operations. JDBC is essential for database-driven applications.

Setting Up JDBC for MySQL in Java

Step 1: Download the JDBC Driver (JAR file)

To connect Java with MySQL, you need to download the MySQL JDBC driver, which is a JAR file. This file allows Java applications to communicate with MySQL databases.

- **How to download the JDBC driver**:
  - Visit the official MySQL site: MySQL Connector/J.
  - Download the version that is compatible with your MySQL installation.

Step 2: Add the JDBC JAR to the Project

Once the JDBC driver JAR file is downloaded, you need to add it to your project classpath.

For Command-Line Development:

- Set the classpath directly when running the program:

```
javac -cp .:/path/to/mysql-connector-java-8.0.x.jar MyProgram.java
java -cp .:/path/to/mysql-connector-java-8.0.x.jar MyProgram
```

For IDE (e.g., Eclipse/IntelliJ IDEA):

- In IntelliJ: `File → Project Structure → Libraries → + → Add JAR`.
- In Eclipse: `Right-click project → Build Path → Configure Build Path → Add External JARs`.

Code to Establish a Connection with MySQL Database

Now, let's connect Java to the MySQL database. Here's the code to establish the connection.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JdbcConnection {
    public static void main(String[] args) {
        Connection con = null;

        try {
            // Step 1: Load the JDBC Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Database connection details
            String url = "jdbc:mysql://localhost:3306/mydatabase";
            String user = "root";  // Database username
            String password = "yourpassword";  // Database password

            // Step 3: Establish connection
            con = DriverManager.getConnection(url, user, password);
            System.out.println("Database connected successfully!");

        } catch (ClassNotFoundException e) {
            System.out.println("JDBC Driver not found. Make sure the JAR is added.");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Connection failed! Check your URL, username, or password.");
            e.printStackTrace();
        } finally {
            // Step 4: Close the connection
            try {
                if (con != null) con.close();
                System.out.println("Database connection closed.");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}
```

Detailed Note on Database Connection Components:

When you're working with JDBC (Java Database Connectivity) to interact with a relational database (such as MySQL), you need to use various objects to perform operations like connecting to the database, executing queries, and processing the results. Let's break down the objects you've declared:

```
Connection con = null;
Statement stmt = null;
PreparedStatement ps = null;
ResultSet rs = null;
```

1. Connection (`Connection con = null;`)

- **Purpose:** The `Connection` object represents a session between your Java application and the database. It is used to establish and maintain the connection to the database. Every time you want to execute a query or an update, you need a connection.
- **Why we use it:**
  - **Establishing Connection:** Before interacting with the database (whether it's querying, updating, or deleting data), we need to establish a connection to the database using this object. It’s essentially a "bridge" that allows your Java application to communicate with the database.
  - **Execute Queries/Updates:** After the connection is established, all operations (like `SELECT`, `UPDATE`, `INSERT`, `DELETE`) are done through the connection object.
  - **Transaction Management:** In case you need to manage transactions (commit/rollback), it is done via the `Connection` object.
- **Example usage:**

  ```
  con = DriverManager.getConnection(url, user, password);
  ```

2. Statement (`Statement stmt = null;`)

- **Purpose:** The `Statement` object is used to execute SQL queries against the database. It is mainly used for static SQL queries (queries that don’t change).
- **Why we use it:**
  - **Executing Queries:** It allows you to execute SQL statements like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. It's useful when the SQL query doesn't need any parameters or dynamic behavior.
  - **Simple Queries:** `Statement` is commonly used for simple, fixed queries that don't require user input or placeholders.
- **Example usage:**

```
stmt = con.createStatement();
stmt.executeQuery("SELECT * FROM users");
```

- **Important:**
  - `Statement` can also be used for batch updates where you want to execute multiple SQL commands at once, but it's often less secure than `PreparedStatement` for queries with parameters (because of SQL injection risks).

---

3. PreparedStatement (`PreparedStatement ps = null;`)

- **Purpose:** `PreparedStatement` is a subclass of `Statement` that is used for executing SQL queries with dynamic parameters. It allows you to write SQL statements with placeholders (`?`), which can be substituted with actual values at runtime.
- **Why we use it:**
  - **Security (SQL Injection Prevention):** Unlike `Statement`, `PreparedStatement` helps prevent SQL injection attacks by automatically escaping special characters in user input.
  - **Efficiency:** A `PreparedStatement` can be executed multiple times with different parameters, and the SQL query is precompiled by the database. This makes it more efficient for repetitive operations.
  - **Dynamic Queries:** Use `PreparedStatement` when your SQL query needs dynamic input (like user-provided values). It is ideal for `INSERT`, `UPDATE`, or `DELETE` queries where the values can change each time the query runs.
- **Example usage:**

```
String query = "INSERT INTO users (name, age) VALUES (?, ?)";
ps = con.prepareStatement(query);
ps.setString(1, "Alice");
ps.setInt(2, 25);
ps.executeUpdate();
```

- **Why we prefer it over `Statement`:**
  - It's safer (prevents SQL injection).
  - It's more efficient when executing the same query repeatedly with different parameters.
  - It allows us to handle complex queries with variable data.

---

4. ResultSet (`ResultSet rs = null;`)

- **Purpose:** The `ResultSet` object stores the result set (the rows returned by a `SELECT` query) from the database. It provides methods to retrieve data from the result of a query.
- **Why we use it:**
  - **Retrieve Data:** After executing a `SELECT` query, the result is stored in a `ResultSet` object. You can use this object to access the data returned by the query.
  - **Cursor-based Navigation:** The `ResultSet` allows you to iterate through the result set one row at a time. It maintains a cursor that points to the current row, and you can use methods like

`next()`, `getString()`, `getInt()`, etc., to access data from the current row.

- **Row Navigation:** It supports navigation through rows (e.g., `rs.next()` to move to the next row in the result set).

- **Example usage:**

```
rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println(name + ": " + age);
}
```

- **Important:**
  - `ResultSet` allows you to work with data fetched from the database, one row at a time.
  - You should always close the `ResultSet` after using it to release database resources.

Why Do We Use These Objects Together?

1. **Connection (`con`)**: The first thing we need to do is establish a connection to the database using `Connection`. Without it, there would be no communication between your Java code and the database.
2. **Statement (`stmt`)**: Once the connection is established, you can use a `Statement` or `PreparedStatement` to send SQL queries to the database.
   - `Statement` is fine for simple queries with fixed content, while `PreparedStatement` is more suitable for queries where you need to bind parameters (dynamic data).
3. **PreparedStatement (`ps`)**: If you need to execute SQL queries with parameters, it's better to use `PreparedStatement`. This helps ensure security (prevents SQL injection) and improves performance for repeated query executions.
4. **ResultSet (`rs`)**: When executing a `SELECT` query, you need to capture the results using a `ResultSet` so you can process and display the data in your application.

Summary of Their Roles:

- **Connection (`con`)**: Manages the connection to the database.
- **Statement (`stmt`)**: Executes static SQL queries without parameters.
- **PreparedStatement (`ps`)**: Executes dynamic SQL queries with parameters, ensuring security and efficiency.
- **ResultSet (`rs`)**: Holds the result of a query, allowing you to navigate and retrieve the data.

Detailed Explanation of Each Step in the Code

1. Import Required Classes:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

- `Connection`: Represents a connection to the database.
- `DriverManager`: Manages a list of database drivers.
- `SQLException`: Handles any database-related errors.

2. Load the JDBC Driver Class:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- This line dynamically loads the MySQL JDBC driver class into memory. This is required for Java to be able to communicate with the MySQL database.

3. Set Up Database Connection Details:

```
String url = "jdbc:mysql://localhost:3306/mydatabase";
String user = "root";
String password = "yourpassword";
```

- `url`: The connection URL specifies the database's location.
  - `jdbc:mysql://` is the protocol used for MySQL.
  - `localhost:3306` refers to the database server running locally on port `3306` (default MySQL port).
  - `mydatabase` is the name of the database you wish to connect to.
- `user` and `password`: These are your MySQL database login credentials (replace `root` and `yourpassword` with your actual credentials).

4. Establish Connection:

```
con = DriverManager.getConnection(url, user, password);
```

- This line actually establishes the connection between your Java application and the MySQL database.

5. Error Handling:

- **ClassNotFoundException**: This exception occurs if the JDBC driver class is not found. It typically happens if you have not included the JDBC JAR file in your project.
- **SQLException**: This exception handles any database-related errors, like incorrect URL, username, or password.

6. Close the Connection:

```
con.close();
```

- Closing the connection after the database operation is complete is crucial to avoid memory leaks and free up resources.

Performing CRUD Operations with JDBC

Once the connection is established, you can perform CRUD operations (Create, Read, Update, Delete). Let's explore each one:

1. INSERT Operation (Adding Data)

```
String sql = "INSERT INTO Users (Id, Name) VALUES (?, ?)";
PreparedStatement ps = con.prepareStatement(sql);
ps.setInt(1, 1);
ps.setString(2, "John Doe");
int rowsInserted = ps.executeUpdate();
```

- We are inserting a new user into the `Users` table with `Id = 1` and `Name = "John Doe"`.
- The `PreparedStatement` helps protect against SQL injection attacks by using placeholders (`?`) for the values.

2. SELECT Operation (Retrieving Data)

```
String sql = "SELECT * FROM Users";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    System.out.println("Id: " + rs.getInt("Id") + ", Name: " + rs.getString("Name"));
}
```

- The `Statement` object is used to execute SQL queries, and the `ResultSet` contains the results returned by the query.
- In this case, we retrieve all users from the `Users` table.

3. UPDATE Operation (Modifying Data)

```
String sql = "UPDATE Users SET Name = ? WHERE Id = ?";
PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1, "Jane Doe");
ps.setInt(2, 1);
ps.executeUpdate();
```

- This query updates the `Name` of the user where `Id = 1`.

4. DELETE Operation (Removing Data)

```
String sql = "DELETE FROM Users WHERE Id = ?";
PreparedStatement ps = con.prepareStatement(sql);
```

```
ps.setInt(1, 1);
ps.executeUpdate();
```

- This query deletes the user with `Id = 1` from the `Users` table.

Advantages of Using JDBC

1. **Database Independence:**
   - JDBC allows Java applications to be independent of the database. You can easily switch databases without modifying the Java code (just change the JDBC driver and URL).
2. **Security:**
   - Prepared statements help prevent SQL injection attacks, a major security concern in database interactions.
3. **Scalability:**
   - JDBC supports connection pooling, which improves performance by reusing database connections, especially for high-traffic applications.
4. **Flexibility:**
   - JDBC is flexible and allows you to interact with a variety of relational databases (MySQL, PostgreSQL, Oracle, etc.) with minimal changes.

Viva Questions and Key Points to Remember

1. **What is JDBC?**
   - JDBC is an API in Java that allows communication between Java applications and databases.
2. **What is the difference between `Statement` and `PreparedStatement`?**
   - `Statement` is used for simple queries, while `PreparedStatement` is precompiled and more efficient, especially when reusing queries and preventing SQL injection.
3. **Why is it necessary to load the JDBC driver using `Class.forName()`?**
   - This step is required to dynamically load the JDBC driver class into memory, enabling Java to use it to interact with the database.
4. **What are the common exceptions thrown by JDBC?**
   - `SQLException`: Thrown for database-related errors.
   - `ClassNotFoundException`: Thrown if the JDBC driver class is not found.
5. **Explain the use of `executeUpdate()` and `executeQuery()`.**
   - `executeUpdate()` is used for SQL statements that modify the database (like `INSERT`, `UPDATE`, `DELETE`), whereas `executeQuery()` is used for SELECT queries that return data.
6. **What is connection pooling in JDBC?**
   - Connection pooling allows for reusing database connections instead of creating a new connection each time, improving performance in high-traffic applications.

Conclusion

This detailed guide explains JDBC in Java, from setting up the environment to performing database operations, as well as important advantages and key points for a viva. With JDBC, Java programs can seamlessly interact with databases, allowing for efficient data management in enterprise-level applications.

Code for Connecting to the Database and Performing CRUD Operations

This code will establish a connection to the MySQL database and perform CRUD operations (Create, Read, Update, Delete).

```
import java.sql.*;

public class JdbcCrudOperations {
    public static void main(String[] args) {
        // Database connection variables
        String url = "jdbc:mysql://localhost:3306/mydatabase"; // Replace with your database name
        String user = "root"; // Replace with your username
        String password = "yourpassword"; // Replace with your password

        // Declare JDBC objects
        Connection con = null;
        Statement stmt = null;
        PreparedStatement ps = null;
        ResultSet rs = null;

        try {
            // Step 1: Load the MySQL JDBC Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Establish connection to the database
            con = DriverManager.getConnection(url, user, password);
            System.out.println("Database connected successfully!");

            // Step 3: Create Statement object for executing queries
            stmt = con.createStatement();

            // 1. CREATE Operation: Insert a new record
            String insertSql = "INSERT INTO Users (Id, Name) VALUES (?, ?)";
            ps = con.prepareStatement(insertSql);
            ps.setInt(1, 1);
            ps.setString(2, "John Doe");
            ps.executeUpdate();
            System.out.println("Data inserted successfully!");

            // 2. READ Operation: Retrieve all records
            String selectSql = "SELECT * FROM Users";
            rs = stmt.executeQuery(selectSql);
```

```
            while (rs.next()) {
                System.out.println("Id: " + rs.getInt("Id") + ", Name: " + rs.getString("Name"));
            }

            // 3. UPDATE Operation: Modify an existing record
            String updateSql = "UPDATE Users SET Name = ? WHERE Id = ?";
            ps = con.prepareStatement(updateSql);
            ps.setString(1, "Jane Doe");
            ps.setInt(2, 1);
            ps.executeUpdate();
            System.out.println("Data updated successfully!");

            // 4. DELETE Operation: Delete a record
            String deleteSql = "DELETE FROM Users WHERE Id = ?";
            ps = con.prepareStatement(deleteSql);
            ps.setInt(1, 1);
            ps.executeUpdate();
            System.out.println("Data deleted successfully!");

        } catch (ClassNotFoundException e) {
            System.out.println("JDBC Driver not found!");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("SQL Error!");
            e.printStackTrace();
        } finally {
            // Step 4: Close resources to avoid memory leaks
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
                if (ps != null) ps.close();
                if (con != null) con.close();
                System.out.println("Database connection closed.");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Explanation:

- **Create**: Insert a new record into the `Users` table.
- **Read**: Fetch all records from the `Users` table and display them.
- **Update**: Modify an existing record by changing the name of the user with `Id = 1`.
- **Delete**: Delete the record with `Id = 1`.

---

2. Code for Running CRUD Operations in Batch Mode

The second code demonstrates how to perform the same CRUD operations, but this time using **batch processing** for improved performance, especially when dealing with large datasets.

```
import java.sql.*;

public class JdbcBatchProcessing {
    public static void main(String[] args) {
        // Database connection variables
        String url = "jdbc:mysql://localhost:3306/mydatabase"; // Replace with your database name
        String user = "root"; // Replace with your username
        String password = "yourpassword"; // Replace with your password

        // Declare JDBC objects
        Connection con = null;
        PreparedStatement ps = null;

        try {
            // Step 1: Load the MySQL JDBC Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Establish connection to the database
            con = DriverManager.getConnection(url, user, password);
            System.out.println("Database connected successfully!");

            // Step 3: Turn off auto-commit mode for batch processing
            con.setAutoCommit(false);

            // 1. CREATE Operation: Insert multiple records using batch
            String insertSql = "INSERT INTO Users (Id, Name) VALUES (?, ?)";
            ps = con.prepareStatement(insertSql);
            ps.setInt(1, 1);
            ps.setString(2, "John Doe");
            ps.addBatch(); // Add to batch

            ps.setInt(1, 2);
            ps.setString(2, "Alice Smith");
            ps.addBatch(); // Add to batch

            ps.setInt(1, 3);
            ps.setString(2, "Bob Brown");
            ps.addBatch(); // Add to batch

            // Execute batch insert
            ps.executeBatch();
            System.out.println("Multiple records inserted successfully!");

            // 2. UPDATE Operation: Update multiple records in a batch
```

```
            String updateSql = "UPDATE Users SET Name = ? WHERE Id = ?";
            ps = con.prepareStatement(updateSql);
            ps.setString(1, "Updated John Doe");
            ps.setInt(2, 1);
            ps.addBatch(); // Add to batch

            ps.setString(1, "Updated Alice Smith");
            ps.setInt(2, 2);
            ps.addBatch(); // Add to batch

            // Execute batch update
            ps.executeBatch();
            System.out.println("Multiple records updated successfully!");

            // 3. DELETE Operation: Delete multiple records using batch
            String deleteSql = "DELETE FROM Users WHERE Id = ?";
            ps = con.prepareStatement(deleteSql);
            ps.setInt(1, 3);
            ps.addBatch(); // Add to batch

            ps.setInt(1, 2);
            ps.addBatch(); // Add to batch

            // Execute batch delete
            ps.executeBatch();
            System.out.println("Multiple records deleted successfully!");

            // Step 4: Commit all changes and enable auto-commit again
            con.commit();
            con.setAutoCommit(true);

        } catch (ClassNotFoundException e) {
            System.out.println("JDBC Driver not found!");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("SQL Error during batch processing!");
            e.printStackTrace();
            try {
                // Rollback if any error occurs during batch processing
                if (con != null) {
                    con.rollback();
                    System.out.println("Transaction rolled back.");
                }
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        } finally {
            // Step 5: Close resources to avoid memory leaks
            try {
```

```
            if (ps != null) ps.close();
            if (con != null) con.close();
            System.out.println("Database connection closed.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
  }
}
```

Explanation:

- **Batch Processing**: This code uses `addBatch()` to add multiple SQL commands to the batch. After all commands are added, `executeBatch()` is called to execute them all at once.
- **CREATE**: Insert multiple records into the `Users` table in one batch.
- **UPDATE**: Update multiple records in one batch.
- **DELETE**: Delete multiple records in one batch.
- **Transaction Handling**: The `setAutoCommit(false)` disables auto-commit to manage the transactions manually. If there's an error, the transaction is rolled back using `rollback()`, ensuring data integrity.

Advantages of Using Batch Processing:

- **Improved Performance**: Instead of executing multiple queries one by one, batch processing sends all queries together, reducing database round-trips and increasing performance.
- **Transaction Management**: Using batch processing, you can handle all operations in a single transaction, and if any issue occurs, you can rollback the entire transaction to ensure data consistency.

Conclusion

By following these two code examples, you will be able to:

1. Establish a database connection and perform CRUD operations.
2. Use batch processing for better performance, especially when dealing with large numbers of records.

Batch processing is a powerful technique in JDBC that helps optimize database interactions and ensures efficient data handling.

Explanation of the JDBC Code

The given Java program demonstrates how to use JDBC to connect to a MySQL database, execute an SQL query, retrieve metadata, and display records from a table. Below is a detailed breakdown of each section of the code:

Understanding Metadata in JDBC (ResultSetMetaData)

Why Use Metadata?

Metadata provides information about the structure of a database table without manually inspecting it. This helps when:

- You need to **dynamically handle queries** without prior knowledge of the table schema.
- You want to **list column names and data types** before processing the actual data.

Code for Retrieving Metadata

```
// Retrieve metadata about the result set
ResultSetMetaData rsmd = rs.getMetaData();

// Get the number of columns in the result set
int columnCount = rsmd.getColumnCount();

System.out.println("Table Metadata:");
System.out.println("----------------------------");

// Loop through all columns in the result set
for (int i = 1; i <= columnCount; i++) {
    System.out.println("Column " + i + ":");
    System.out.println("   Name: " + rsmd.getColumnName(i));
    System.out.println("   Type: " + rsmd.getColumnTypeName(i));
    System.out.println("   Display Size: " + rsmd.getColumnDisplaySize(i));
    System.out.println("   Nullable: " + (rsmd.isNullable(i) == ResultSetMetaData.columnNullable ?
"Yes" : "No"));
    System.out.println("----------------------------");
}
```

Explanation of Code

1. **Retrieve Metadata Object**

   ```
   ResultSetMetaData rsmd = rs.getMetaData();
   ```

   - Extracts **metadata** from the query result stored in `ResultSet`.

2. **Get Number of Columns**

   ```
   int columnCount = rsmd.getColumnCount();
   ```

   - Finds how many columns exist in the selected result.

3. **Loop Through Each Column**

   - **Column Name:**

```
rsmd.getColumnName(i)
```

- **Column Type:**

```
rsmd.getColumnTypeName(i)
```

- **Display Size (Max Width):**

```
rsmd.getColumnDisplaySize(i)
```

- **Nullable Check:**

```
rsmd.isNullable(i) == ResultSetMetaData.columnNullable ? "Yes" : "No"
```

Example Output

```
Table Metadata:
----------------------------
Column 1:
  Name: ID
  Type: INT
  Display Size: 10
  Nullable: No
----------------------------
Column 2:
  Name: NAME
  Type: VARCHAR
  Display Size: 255
  Nullable: Yes
----------------------------
```

This helps in **dynamically understanding table structure**, making JDBC applications more flexible! 🚀

IN, OUT, and INOUT Parameters in SQL Stored Procedures

Introduction

In SQL, a **Stored Procedure** is a precompiled collection of SQL statements that can be executed as a single unit. These procedures often accept parameters, which help make them dynamic and reusable. Parameters in stored procedures can be classified into three types:

1. **IN Parameter** – Used to pass values **into** the stored procedure.
2. **OUT Parameter** – Used to return values **from** the stored procedure.
3. **INOUT Parameter** – Used for both **input and output** in the stored procedure.

1. IN Parameter

Definition

- The **IN parameter** is used to pass values **into** a stored procedure.
- The procedure **can use** the parameter but **cannot modify** its value.
- This parameter must be provided by the caller when invoking the procedure.
- It is the default parameter type if none is specified.

Example

Creating a Procedure with an IN Parameter

```
DELIMITER //
CREATE PROCEDURE GetEmployeeDetails(IN emp_id INT)
BEGIN
    SELECT * FROM employees WHERE id = emp_id;
END //
DELIMITER ;
```

Calling the Procedure in Java (JDBC)

```
Connection conn = DriverManager.getConnection(url, user, password);
CallableStatement stmt = conn.prepareCall("{CALL GetEmployeeDetails(?)}");
stmt.setInt(1, 101);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    System.out.println("Employee Name: " + rs.getString("name"));
}
stmt.close();
conn.close();
```

2. OUT Parameter

Definition

- The **OUT parameter** is used to **return** a value from the stored procedure.
- Unlike **IN parameters**, OUT parameters **can be modified** inside the procedure.
- The caller must declare a variable to store the returned value.

Example

Creating a Procedure with an OUT Parameter

```
DELIMITER //
CREATE PROCEDURE GetEmployeeName(IN emp_id INT, OUT emp_name VARCHAR(255))
```

```
BEGIN
    SELECT name INTO emp_name FROM employees WHERE id = emp_id;
END //
DELIMITER ;
```

Calling the Procedure in Java (JDBC)

```
Connection conn = DriverManager.getConnection(url, user, password);
CallableStatement stmt = conn.prepareCall("{CALL GetEmployeeName(?, ?)}");
stmt.setInt(1, 101);
stmt.registerOutParameter(2, Types.VARCHAR);
stmt.execute();
String employeeName = stmt.getString(2);
System.out.println("Employee Name: " + employeeName);
stmt.close();
conn.close();
```

---

3. INOUT Parameter

Definition

- The **INOUT parameter** allows both **input and output** operations.
- The initial value is passed **into** the procedure, and the modified value is **returned** after execution.
- This is useful for updating or transforming values dynamically.

Example

Creating a Procedure with an INOUT Parameter

```
DELIMITER //
CREATE PROCEDURE UpdateSalary(INOUT salary DECIMAL(10,2))
BEGIN
    SET salary = salary + 5000;
END //
DELIMITER ;
```

Calling the Procedure in Java (JDBC)

```
Connection conn = DriverManager.getConnection(url, user, password);
CallableStatement stmt = conn.prepareCall("{CALL UpdateSalary(?)}");
stmt.setDouble(1, 40000);
stmt.registerOutParameter(1, Types.DECIMAL);
stmt.execute();
double updatedSalary = stmt.getDouble(1);
System.out.println("Updated Salary: " + updatedSalary);
```

```
stmt.close();
conn.close();
```

Comparison Table

Parameter Type	Direction	Can be Modified Inside Procedure?	Purpose
IN	Input Only	❌ No	Pass values to the procedure.
OUT	Output Only	✅ Yes	Return values from the procedure.
INOUT	Input & Output	✅ Yes	Pass a value, modify it, and return it.

When to Use Each Parameter?

Use Case	Parameter Type
Fetching records based on input	**IN**
Returning a single computed or queried value	**OUT**
Performing operations on input and returning the modified value	**INOUT**

Conclusion

Understanding **IN, OUT, and INOUT parameters** in SQL stored procedures helps in designing efficient and dynamic database operations.

- Use **IN parameters** when you only need to provide input.
- Use **OUT parameters** when you need to return a single value from a procedure.
- Use **INOUT parameters** when you need to both modify and return a value.

With proper use of these parameters, stored procedures can be optimized for **better performance, reusability, and scalability** in SQL-based applications.