Java

Module 2 – Java – RDBMS & Database Programming with JDBC

Assignment :2

1) What is JDBC (Java Database Connectivity)?

→JDBC (Java Database Connectivity) is a Java API (Application Programming Interface) that allows Java programs to connect to databases, send SQL queries, and retrieve or modify data.

Basic JDBC Steps:

1. **Load driver**
2. **Connect to DB**
3. **Create statement**
4. **Execute query**
5. **Process result**
6. **Close connection**

2) Importance of JDBC in Java Programming

→1) Database Connectivity
JDBC allows Java programs to connect with databases like MySQL, Oracle, etc.

2) Platform Independent
Write once, run anywhere — JDBC works across different databases with drivers.

3) Execute SQL Queries
Helps perform INSERT, UPDATE, DELETE, SELECT operations from Java code.

4) Secure and Reliable
Supports PreparedStatement to prevent SQL injection attacks.

5) Bridge Between Java and Database
Acts as a link to interact with data stored in a database.

6) Supports Enterprise Applications
Used in web apps, banking systems, e-commerce, etc., where database interaction is needed.

3) JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet

→1. DriverManager:Manages a list of database drivers.

Establishes connection between Java application and the database.

EX:

Connection con = DriverManager.getConnection(URL, username, password);

2) **Driver:** Interface that communicates with the database.

Each database has its own driver (e.g., MySQL Driver, Oracle Driver).

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

3) **Connection:**Represents a connection/session between Java and the database.
Used to send SQL statements.

Ex: Connection con = DriverManager.getConnection(...);

4) Statement: Used to execute SQL queries (like SELECT, INSERT, UPDATE).

There are 3 types: Statement, PreparedStatement, and CallableStatement.

Ex: Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM users");

5) **ResultSet**

- Holds the data returned by a SQL SELECT query.
- You can loop through it to read dat

Ex: while (rs.next()) {

System.out.println(rs.getString("name"));

}

*JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet

→ **Driver Manager**

- Controls the list of database drivers.
- Loads the driver class and establishes a connection.

 Driver

- Implements the JDBC API for a specific database type.
- Converts JDBC calls into database-specific calls.

 Connection

- Represents the session between the Java application and the database.
- Used to create Statement/PreparedStatement objects.

4**Statement**

- Sends SQL queries to the database.
- Types: Statement, PreparedStatement, CallableStatement.

5.**ResultSet**

- Stores the data returned by a SELECT query.
- Allows row-by-row and column-by-column access to data.

2. JDBC Driver Types

→

Type 1 — JDBC-ODBC Bridge:-

1. Java app calls JDBC API.
2. JDBC-ODBC bridge receives the calls and converts them to ODBC calls.
3. ODBC Driver Manager locates the native ODBC driver.
4. ODBC driver communicates with the database.

Type 2 — Native-API:-

1. Java app calls JDBC API.
2. Type-2 driver's Java part forwards calls to its native library
3. Native library speaks the DB's native protocol to the database.

Type 3 — Network Protocol (middleware) driver:-

1. Java app calls JDBC API.
2. Type-3 driver sends requests over the network in a vendor-neutral protocol to a middleware/proxy.
3. Middleware translates requests into the target DB protocol and forwards to the DB.

4. Middleware returns results to the Type-3 driver which relays them to the app.

Type 4 — Thin / Pure Java driver:-

1. Java app calls JDBC API.
2. Type-4 driver (a single .jar) converts calls directly to the database protocol and connects over the network to the DB.
3. DB responds; driver maps results back to JDBC objects.

3) Steps for Creating JDBC Connections.

→

*__Register the JDBC driver:__

Modern JDBC : driver JARs auto-register if they include the service file — no need for Class.forName().

Legacy / explicit: use Class.forName("com.mysql.cj.jdbc.Driver"); only if auto-loading fails.

Common driver class names:

MySQL: com.mysql.cj.jdbc.Driver.

*__Open a connection to the database:__

Use DriverManager.getConnection(url, user, password).

EX: String url = "jdbc:mysql://localhost:3306/mydatabase";

String user = "appuser";

String pass = "secret";

Connection conn = DriverManager.getConnection(url, user, pass);

***Create a statement:**

Prefer PreparedStatement for parameters and safety (prevents SQL injection).

Other types: Statement (simple), CallableStatement (stored procedures).

Ex:

String sql = "SELECT id, name FROM users WHERE status = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, "ACTIVE");

***Execute SQL queries:**

- executeQuery() → returns ResultSet (SELECT).
- executeUpdate() → returns affected rows (INSERT/UPDATE/DELETE).
- Ex:
- ResultSet rs = ps.executeQuery(); // SELECT
- int rows = ps.executeUpdate(); // INSERT/UPDATE/DELETE.

*Process the result set

- Iterate with while (rs.next()).
- Retrieve columns by name or index: rs.getInt("id"), rs.getString(2).

Ex:

```
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
}
```

***Close the connection**

- Always close ResultSet, Statement/PreparedStatement, and Connection — in reverse order of creation.
- Best way: use try-with-resources so resources close automatically.

Ex:

```
try (Connection conn = DriverManager.getConnection(url, user, pass);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql)) {
   // use rs here
}
```

4) Types of JDBC Statements

→1) Statement: Executes plain SQL strings.

Steps

1. Create Statement: Statement stmt = conn.createStatement();
2. Execute: ResultSet rs = st.executeQuery("SELECT * FROM users"); or int r = st.executeUpdate("DELETE FROM users WHERE id=5");
3. Process rs.
4. Close resources (prefer try-with-resources).

Ex: try (Statement st = conn.createStatement();

ResultSet rs = st.executeQuery("SELECT id,name FROM users")) {

while (rs.next()) { /* ... */ }

}

2) PreparedStatement:

Steps

1. Prepare: PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE status = ? AND age > ?");
2. Set params (index starts at 1): ps.setString(1, "ACTIVE"); ps.setInt(2, 18);
3. Execute: ResultSet rs = ps.executeQuery(); or ps.executeUpdate();
4. Process results.
5. Close resources.

Ex: String sql = "INSERT INTO users(name, email) VALUES(?, ?)";

try (PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setString(1, "Sagar");

ps.setString(2, "sagar@example.com");

ps.executeUpdate();

}

3) CallableStatement:

Steps

1. Prepare call: CallableStatement cs = conn.prepareCall("{call getUser(?, ?)}");
2. Set IN params: cs.setInt(1, userId);
3. Register OUT params: cs.registerOutParameter(2, java.sql.Types.VARCHAR);

4. Execute: cs.execute();
5. Read OUT: String name = cs.getString(2);
6. Close resources.

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

→

```
import java.sql.*;
import java.util.Scanner;

public class JDBC_CRUD {
    static final String URL = "jdbc:mysql://localhost:3306/studentdb";
    static final String USER = "root";  // change as per your DB
    static final String PASS = "password"; // change as per your DB

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try (Connection conn = DriverManager.getConnection(URL, USER, PASS)) {
            while (true) {
                System.out.println("\n--- Student Management Menu ---");
                System.out.println("1. Insert Student");
                System.out.println("2. Update Student");
                System.out.println("3. View All Students");
                System.out.println("4. Delete Student");
                System.out.println("5. Exit");
                System.out.print("Enter choice: ");
                int choice = sc.nextInt();

                switch (choice) {
```

```
                case 1: insertStudent(conn, sc); break;
                case 2: updateStudent(conn, sc); break;
                case 3: selectStudents(conn); break;
                case 4: deleteStudent(conn, sc); break;
                case 5: System.out.println("Exiting..."); return;
                default: System.out.println("Invalid choice!");
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

// Insert
private static void insertStudent(Connection conn, Scanner sc) throws SQLException {
    System.out.print("Enter ID: ");
    int id = sc.nextInt();
    System.out.print("Enter Name: ");
    sc.nextLine();
    String name = sc.nextLine();
    System.out.print("Enter Age: ");
    int age = sc.nextInt();
    System.out.print("Enter Course: ");
    sc.nextLine();
    String course = sc.nextLine();

    String sql = "INSERT INTO student (id, name, age, course) VALUES (?, ?, ?, ?)";
    try (PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(1, id);
```

```
            ps.setString(2, name);
            ps.setInt(3, age);
            ps.setString(4, course);
            int rows = ps.executeUpdate();
            System.out.println(rows > 0 ? "Student inserted successfully!" : "Insert failed!");
        }
    }

    // Update
    private static void updateStudent(Connection conn, Scanner sc) throws SQLException
{
        System.out.print("Enter ID to Update: ");
        int id = sc.nextInt();
        System.out.print("Enter New Name: ");
        sc.nextLine();
        String name = sc.nextLine();

        String sql = "UPDATE student SET name = ? WHERE id = ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, name);
            ps.setInt(2, id);
            int rows = ps.executeUpdate();
            System.out.println(rows > 0 ? "Student updated successfully!" : "Student not
found!");
        }
    }

    // Select
    private static void selectStudents(Connection conn) throws SQLException {
        String sql = "SELECT * FROM student";
```

```
        try (Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery(sql))
{
            System.out.println("\nID\tName\tAge\tCourse");
            while (rs.next()) {
                System.out.println(rs.getInt("id") + "\t" +
                                   rs.getString("name") + "\t" +
                                   rs.getInt("age") + "\t" +
                                   rs.getString("course"));
            }
        }
    }

    // Delete
    private static void deleteStudent(Connection conn, Scanner sc) throws SQLException {
        System.out.print("Enter ID to Delete: ");
        int id = sc.nextInt();

        String sql = "DELETE FROM student WHERE id = ?";
        try (PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, id);
            int rows = ps.executeUpdate();
            System.out.println(rows > 0 ? "Student deleted successfully!" : "Student not
found!");
        }
    }
}
```

6) ResultSet Interface

→

A ResultSet is an object in Java that stores the data returned from a **SELECT** SQL query executed using JDBC.

It acts like a table in memory, where each row represents a record from the database, and you can move through it using a **cursor** to read data column by column.

Key points:

- Created when you run executeQuery() on a Statement or PreparedStatement.
- Stores query results temporarily in memory.
- Provides **getter methods** (getInt(), getString(), etc.) to read data.
- Has a **cursor** that points to the current row.
- Can be **forward-only** or **scrollable** (move to first, last, previous, etc.).

  Ex:

```
ResultSet rs = stmt.executeQuery("SELECT id, name FROM users");
while (rs.next()) {
   System.out.println(rs.getInt("id") + " " + rs.getString("name"));
}
```

1. Navigating through ResultSet (first, last, next, previous)

→*Make ResultSet Scrollable

By default, ResultSet is forward-only (you can only use .next()), so you must create a scrollable ResultSet:

Ex:

```
Statement stmt = conn.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,  // Allows forward & backward navigation
   ResultSet.CONCUR_READ_ONLY          // Read-only mode
);
```

*Execute the Query

Ex:

```
ResultSet rs = stmt.executeQuery("SELECT id, name FROM users");
```

*Navigate Through the ResultSet

Ex:

```
Move to first row
if (rs.first()) {
   System.out.println("First Row: " + rs.getInt("id") + " " + rs.getString("name"));
}
```

- Move to last row:

```
Ex: if (rs.last()) {
  System.out.println("Last Row: " + rs.getInt("id") + " " + rs.getString("name"));
}
Move to next row:
while (rs.next()) {
  System.out.println("Next Row: " + rs.getInt("id") + " " + rs.getString("name"));
}
Move to previous row:
while (rs.previous()) {
  System.out.println("Previous Row: " + rs.getInt("id") + " " + rs.getString("name"));
}
```

4)Close Resources:

Ex:

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

3)Working with ResultSet to retrieve data from SQL queries

→Execute Query :

```
ResultSet rs = st.executeQuery("SELECT id, name FROM users");
```

Read Data:

```
while (rs.next()) {
  System.out.println(rs.getInt("id") + " " + rs.getString("name"));
}
```

Close:

rs.close();

7) Database Metadata

→

Database MetaData in JDBC is an interface that provides information about the database and the JDBC driver you are connected to.

Step:

Database name & version (e.g., MySQL 8.0)

Driver name & version

List of tables, columns, and keys

Database capabilities

Ex:

DatabaseMetaData meta = conn.getMetaData();

DatabaseMetaData is important because it allows your Java program to understand the database it's connected to without hardcoding details.

Discover Database Structure:

- List tables, columns, indexes, keys in the database.
- Helps when you don't know the schema beforehand.

8) Result Set Metadata ?

→

- An interface in JDBC.
- Describes the **structure** of columns in a ResultSet.
- Tells you column **names, types, counts, sizes, nullability**, etc.

Ex:

ResultSetMetaData md = rs.getMetaData();

2. Importance of ResultSet Metadata

- Lets you **analyze query results** without knowing table structure in advance.
- Useful for **dynamic applications** (reports, table generators, generic tools).
- Helps pick **correct getters** (getInt, getString, etc.) based on column type.

- Enables **validation** (e.g., checking column count or names).

10) Practical Example 1: Swing GUI for CRUD Operations

→1. Introduction to Java Swing for GUI Development

- **Swing**: A Java API for creating **Graphical User Interfaces (GUI)**.
- Part of javax.swing package.
- Works on **top of AWT**, but more flexible and lightweight.
- Common components: JFrame (window), JButton (button), JLabel (label), JTextField (input box), JTable (table).
- Event handling done via **listeners** (e.g., ActionListener).

2. Integrating Swing with JDBC for CRUD Operations

Steps:

1. **Create GUI** with Swing components for user input and output (e.g., forms, tables).
2. **Add event listeners** to buttons (Add, Update, Delete, View).
3. **Connect to database** using JDBC (DriverManager.getConnection).
4. **Perform CRUD**:
    - **Create** → Insert data from text fields to DB.
    - **Read** → Fetch data from DB and display in JTable.
    - **Update** → Modify selected record.
    - **Delete** → Remove selected record.
5. **Close resources** after operations.

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

→

CallableStatement is a **JDBC interface** used to **call stored procedures** in a database.

- Created using Connection.prepareCall().
- Supports **IN**, **OUT**, and **INOUT** parameters.

Steps:

1. **Create stored procedure** in DB. Example:

sql

```
CREATE PROCEDURE getEmployeeName(IN empId INT)

BEGIN
```

```
    SELECT name FROM employees WHERE id = empId;
END;
```

2. **Connect to DB** using JDBC.
3. **Prepare callable statement**:

```
CallableStatement cs = con.prepareCall("{call getEmployeeName(?)}");
```

4. **Set parameters** (if IN type).
5. **Execute** using executeQuery() or executeUpdate().

3. Working with IN and OUT Parameters

- **IN parameter** → Pass value to procedure using setXXX() methods.
- **OUT parameter** → Get returned value using registerOutParameter() and getXXX().

Example

```
CallableStatement cs = con.prepareCall("{call getSalary(?, ?)}");

cs.setInt(1, 101); // IN param

cs.registerOutParameter(2, Types.DOUBLE); // OUT param

cs.execute();

double salary = cs.getDouble(2).
```

SUBMITTED BY SAGAR GOSWAMI