1. Introduction to JDBC

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with relational databases such as MySQL, Oracle, PostgreSQL, etc.

Features:

- Platform-independent database access
- Supports basic SQL operations like CRUD
- Allows for **dynamic SQL**, **prepared statements**, and **stored procedures**

JDBC Architecture:

- **DriverManager** – Manages JDBC drivers
- **Connection** – Establishes connection with the DB
- **Statement** – Used to execute SQL queries
- **ResultSet** – Holds data retrieved from DB
- **PreparedStatement / CallableStatement** – Used for parameterized queries and stored procedures

---

□ 2. Connecting to a Database

To connect to a database, you need:

- Database Driver (e.g., `mysql-connector-java`)
- Database URL
- Username and Password

Example: Connecting to MySQL

```
import java.sql.*;

public class DBConnect {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/testdb"; // DB URL
        String username = "root";
        String password = "password";

        try {
            Class.forName("com.mysql.cj.jdbc.Driver"); // Load driver
            Connection conn = DriverManager.getConnection(url, username,
password);
            System.out.println("Connected to DB!");
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Live Scenario:

A Java-based school management app connects to the MySQL database to validate admin login credentials.

---

3. Creating & Executing SQL Statements

JDBC allows executing SQL using `Statement`, `PreparedStatement`, or `CallableStatement`.

Create Table Example

```
Statement stmt = conn.createStatement();
String sql = "CREATE TABLE Students (id INT PRIMARY KEY, name VARCHAR(100))";
stmt.executeUpdate(sql);
System.out.println("Table created successfully.");
```

Insert, Update, Delete

```
String insertQuery = "INSERT INTO Students VALUES (1, 'Alice')";
stmt.executeUpdate(insertQuery);

String updateQuery = "UPDATE Students SET name='Bob' WHERE id=1";
stmt.executeUpdate(updateQuery);

String deleteQuery = "DELETE FROM Students WHERE id=1";
stmt.executeUpdate(deleteQuery);
```

Live Scenario:

A library system stores book data using insert and update SQL statements through JDBC.

□ 4. Handling Result Sets

`ResultSet` is used to store data retrieved by SELECT queries.

Example: Retrieving Data

```
String query = "SELECT * FROM Students";
ResultSet rs = stmt.executeQuery(query);

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

Live Scenario:

An employee portal displays employee records fetched using `ResultSet` and displayed in a table view on the frontend.

□ 5. Using PreparedStatement

`PreparedStatement` prevents SQL injection and allows you to reuse SQL queries with different parameters.

Syntax:

```
String query = "INSERT INTO Students (id, name) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setInt(1, 2);
pstmt.setString(2, "Charlie");
pstmt.executeUpdate();
```

Benefits:

- Prevents SQL injection
- Faster for repeated executions

Live Scenario:

User registration form inserts user data into the DB using `PreparedStatement`.

□ 6. Using CallableStatement (Stored Procedures)

`CallableStatement` is used to call stored procedures from the database.

Example: Calling a Stored Procedure

Suppose you have a stored procedure:

```
CREATE PROCEDURE getStudent(IN stu_id INT)
BEGIN
   SELECT * FROM Students WHERE id = stu_id;
END;
```

Java Code:

```
CallableStatement cstmt = conn.prepareCall("{call getStudent(?)}");
cstmt.setInt(1, 1);
ResultSet rs = cstmt.executeQuery();

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

Live Scenario:

An enterprise payroll app uses `CallableStatement` to run monthly salary calculations via stored procedures.

□ 7. Best Practices in JDBC

- Always **close** `Connection`, `Statement`, and `ResultSet` in `finally` block or use **try-with-resources**
- Use **PreparedStatement** instead of `Statement` to avoid SQL injection
- Use **Connection Pooling** (e.g., HikariCP, Apache DBCP) for real-time apps
- Handle **SQL exceptions** carefully using logging and rollback

---

□ 8. Sample JDBC Program (End-to-End)

```
import java.sql.*;

public class StudentApp {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/testdb";
        String user = "root";
        String pass = "password";

        try (Connection conn = DriverManager.getConnection(url, user, pass))
{
            String insertSQL = "INSERT INTO Students (id, name) VALUES (?,
?)";
            PreparedStatement pstmt = conn.prepareStatement(insertSQL);
            pstmt.setInt(1, 101);
            pstmt.setString(2, "Zara");
            pstmt.executeUpdate();

            String selectSQL = "SELECT * FROM Students";
            ResultSet rs = pstmt.executeQuery(selectSQL); // use stmt instead
in real scenario

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

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

□ 9. Real-World Project Use Cases

Use Case	JDBC Usage
Hospital Management System	Patient registration, appointment scheduling using `PreparedStatement`
Banking System	Transactions, balance updates using stored procedures via `CallableStatement`
Learning Management System (LMS)	Inserting student data, generating reports using `ResultSet`
E-commerce Checkout	Inventory and payment updates using JDBC