JDBC

---

1. Introduction to JDBC

1.1 What is JDBC?

Java Database Connectivity (JDBC) is an API provided by Java to connect and interact with databases. It enables Java applications to execute SQL statements.

1.2 Why use JDBC?

- Platform-independent database access
- Allows executing SQL queries directly from Java
- Integrates well with enterprise applications

1.3 Features of JDBC

- Supports DDL, DML, DQL operations
- Supports stored procedures
- Exception handling via SQLException
- Database-independent interface

1.4 JDBC Architecture

- Consists of two layers:
    - Application layer (Java code)
    - JDBC Driver layer
- Uses a driver manager to establish a connection to the database

1.5 JDBC Drivers

- Type 1: JDBC-ODBC Bridge
- Type 2: Native-API driver
- Type 3: Network Protocol driver
- Type 4: Thin driver (pure Java, widely used)

---

2. JDBC Setup

2.1 Steps to Setup JDBC in Java

1. Import JDBC package
2. Load and register driver
3. Establish connection
4. Create statement
5. Execute SQL
6. Process results
7. Close resources

2.2 Loading JDBC Driver

Use `Class.forName("com.mysql.cj.jdbc.Driver")` to load the driver class

2.3 Registering the Driver

- Manual: `Class.forName()`
- Automatic: Driver auto-registration in JDBC 4.0+

2.4 Creating a Connection

Use `DriverManager.getConnection(url, user, password)` to create a connection

2.5 Connection URL Format

For MySQL: `jdbc:mysql://hostname:port/dbname`

---

3. JDBC Core Interfaces and Classes

3.1 DriverManager

Manages a list of database drivers and establishes connection with databases

3.2 Connection

Represents a connection to a specific database instance. Provides transaction control

3.3 Statement

Used to execute static SQL queries

3.4 PreparedStatement

Used for executing parameterized queries

3.5 CallableStatement (Optional)

Used to execute stored procedures

3.6 ResultSet

Used to process data returned by SELECT queries

3.7 SQLException

Handles database access errors and provides methods to retrieve error information

---

4. Executing SQL Statements

4.1 Executing DDL Statements

Use `executeUpdate()` for commands like CREATE, ALTER, DROP

4.2 Executing DML Statements

Use `executeUpdate()` for INSERT, UPDATE, DELETE

4.3 Executing SELECT (DQL) Statements

Use `executeQuery()` to execute SELECT and return a ResultSet

4.4 Method Differences

- `execute()`: Returns boolean, used for any SQL
- `executeQuery()`: Returns ResultSet (for SELECT)
- `executeUpdate()`: Returns int (rows affected), for DML/DDL

---

5. ResultSet Navigation and Metadata

5.1 ResultSet Types

- TYPE_FORWARD_ONLY
- TYPE_SCROLL_INSENSITIVE
- TYPE_SCROLL_SENSITIVE

5.2 Navigating ResultSet

Use methods like `next()`, `previous()`, `first()`, `last()`, `absolute(index)`

5.3 ResultSetMetaData

Provides metadata about the ResultSet (e.g., column names, types)

5.4 Fetching Column Info

Use methods like `getColumnName()`, `getColumnTypeName()`

---

6. PreparedStatement vs Statement

6.1 Performance Benefits

- Query is precompiled and reused in `PreparedStatement`

6.2 SQL Injection Prevention

- Parameterized queries prevent injection vulnerabilities

6.3 Positional Parameters (?)

- Use `?` placeholders and set them via `setString()`, `setInt()` etc.

6.4 Reusing Statements

- Prepared statements can be reused with different parameter values

---

7. CallableStatement and Stored Procedures (Optional)

7.1 Calling Stored Procedures

Use `CallableStatement` to execute database procedures

7.2 Input/Output Parameters

Set input using `setXXX()` and retrieve output using `getXXX()`

7.3 Registering OUT Parameters

Use `registerOutParameter()` to define output parameters

---

8. JDBC Transactions

8.1 Auto-commit Mode

- By default, each SQL statement is committed automatically
- Disable with `connection.setAutoCommit(false)`

8.2 Commit and Rollback

- `commit()` to persist changes
- `rollback()` to undo changes

8.3 Savepoint

Intermediate points in transactions to rollback partially

8.4 ACID Properties

- Atomicity, Consistency, Isolation, Durability are supported in JDBC with proper transaction handling

---

9. JDBC Exception Handling

9.1 Handling SQLException

Use try-catch blocks to handle and analyze database errors

9.2 Methods in SQLException

- `getMessage()`
- `getSQLState()`
- `getErrorCode()`

9.3 Exception Chaining

Handle multiple exceptions via `getNextException()`

---

10. JDBC Best Practices (Optional)

10.1 Closing Resources

Always close `Connection`, `Statement`, and `ResultSet` using try-with-resources

10.2 Connection Pooling (Intro Only)

Use libraries like HikariCP or Apache DBCP for efficient resource management

10.3 Reusable Utility Class

Create a helper class to open and close connections

10.4 Thread Safety Basics

Avoid sharing `Connection`, `Statement`, or `ResultSet` across threads

JDBC is an API (Application Programming Interface) provided by Java that allows **Java programs to interact with databases** (like PostgreSQL, MySQL, Oracle, etc.).

It lets you:

- Connect to a database
- Send SQL queries/statements
- Retrieve and manipulate data

JDBC Architecture

JDBC has two main layers:

1. **JDBC API** (Java-side): Interfaces like `Connection`, `Statement`, `PreparedStatement`, `ResultSet`, etc.

Steps to Connect Java App to Database using JDBC:

Step 1: Import JDBC Packages

```
import java.sql.*;
```

Step 2: Load and Register JDBC Driver

You must load the appropriate driver for your database.

```
// For PostgreSQL
Class.forName("org.postgresql.Driver");

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

`Class.forName()` loads the driver class and registers it with the DriverManager.

Step 3: Establish the Connection

Use `DriverManager.getConnection()` with proper credentials:

```
Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/my_database",  // URL
    "postgres",                                      // Username
    "password"                                       // Password
);
```

URL Format:

- **PostgreSQL:** `jdbc:postgresql://host:port/dbname`
- **MySQL:** `jdbc:mysql://host:port/dbname`

Step 4: Create a Statement or PreparedStatement

Option 1: Using Statement (for simple queries)

```
Statement stmt = conn.createStatement();
```

Option 2: Using `PreparedStatement` (for parameterized queries)

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?");
pstmt.setInt(1, 101);
```

Step 5: Execute the Query

- **For SELECT:**

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

For INSERT/UPDATE/DELETE:

```
int rowsAffected = stmt.executeUpdate("UPDATE users SET name='John' WHERE id=1");
```

Step 6: Process the Result

```
while (rs.next()) {
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println("Name: " + name + ", Age: " + age);
}
```

Step 7: Close the Resources

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

Always close resources to avoid memory leaks.

Summary (Relationship from Java to DB)

1. **Load Driver** → `Class.forName(...)`
2. **Establish Connection** → `DriverManager.getConnection(...)`
3. **Create Statement** → `conn.createStatement()` or `conn.prepareStatement(...)`
4. **Execute Query** → `stmt.executeQuery()` or `stmt.executeUpdate()`
5. **Process Results** → Loop through `ResultSet`
6. **Close Resources**

Connecting to PostgreSQL and Selecting Data

Saved memory full

```java
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        try {
            // Step 1: Load driver
            Class.forName("org.postgresql.Driver");

            // Step 2: Connect to DB
            Connection conn = DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/my_database",
                "postgres",
                "mypassword"
            );

            // Step 3: Create Statement
            Statement stmt = conn.createStatement();

            // Step 4: Execute query
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");

            // Step 5: Process results
            while (rs.next()) {
                System.out.println("User ID: " + rs.getInt("id"));
                System.out.println("Name: " + rs.getString("name"));
            }

            // Step 6: Close resources
            rs.close();
            stmt.close();
            conn.close();

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