Comprehensive Guide to Java Database Connectivity (JDBC)

1. Introduction to JDBC

JDBC (Java Database Connectivity) is a Java API that allows Java applications to interact with a wide range of relational databases (like MySQL, Oracle, PostgreSQL). It acts as a bridge, enabling you to send SQL statements to a database and retrieve the results.

Key Components

- **JDBC API (java.sql, javax.sql):** Provides the standard interfaces and classes for connecting to databases (e.g., Connection, Statement, ResultSet).
- **JDBC Driver Manager:** Manages a list of database drivers. It matches connection requests from the Java application with the proper database driver.
- **JDBC Driver:** A client-side adapter (implementation) that converts JDBC calls into the specific protocol required by the database vendor.

2. JDBC Architecture

JDBC follows a modular architecture that supports two main models:

1. **Two-Tier Architecture:** The Java application communicates directly with the database. This is common in client-server applications.
2. **Three-Tier Architecture:** The Java application communicates with a middle-tier (middleware/application server), which then talks to the database. This is standard for web applications.

3. Types of JDBC Drivers

There are four types of drivers available. **Type 4** is the industry standard for modern applications.

Driver Type	Name	Description	Pros/Cons
Type 1	**JDBC-ODBC Bridge**	Translates JDBC calls to ODBC calls. Relies on the OS's ODBC driver.	**Deprecated.** Platform-dependent and slow.
Type 2	**Native-API Driver**	Uses client-side libraries of the	Faster than Type 1 but still

		database (C/C++).	platform-dependent (requires native lib installation).
Type 3	**Network Protocol**	Communicates with a middleware server, which then talks to the DB.	Platform-independent but requires middleware setup.
Type 4	**Thin (Pure Java)**	Converts JDBC calls directly to the vendor's specific database protocol.	**Recommended.** Written in 100% Java. No client-side installation required. Fastest and portable.

4. Steps to Establish a JDBC Connection

Connecting to a database generally involves these 7 steps:

1. **Import Packages:** import java.sql.*;
2. **Load Driver:** Load the driver class (optional in JDBC 4.0+).
3. **Register Driver:** Register with DriverManager.
4. **Connect:** Create a Connection object.
5. **Create Statement:** Create an object to execute queries.
6. **Execute Query:** Run SQL (CRUD operations).
7. **Close:** Release resources.

Basic Code Template

```
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        // Database credentials
        String url = "jdbc:mysql://localhost:3306/testdb"; // 'jdbc:subprotocol:subname'
        String user = "root";
        String password = "password";

        // Try-with-resources (Best Practice: automatically closes connection)
        try (Connection con = DriverManager.getConnection(url, user, password)) {

            System.out.println("Connected to the database!");
```

```
            // ... Logic goes here

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

5. Statements: Statement vs. PreparedStatement vs. CallableStatement

A. Statement

Used for general-purpose access to the database. Useful when using static SQL statements at runtime.

- **Risk:** Vulnerable to **SQL Injection**.
- **Performance:** Slower for repeated queries (database compiles query every time).

B. PreparedStatement (Recommended)

Used when you plan to use the SQL statement many times. It accepts input parameters (?).

- **Security:** Prevents **SQL Injection**.
- **Performance:** Pre-compiled by the database, making it faster.

Example: Insert using PreparedStatement

```
String sql = "INSERT INTO Students(id, name, marks) VALUES(?, ?, ?)";
PreparedStatement ps = con.prepareStatement(sql);

ps.setInt(1, 101);          // 1st '?'
ps.setString(2, "Amit");    // 2nd '?'
ps.setInt(3, 95);           // 3rd '?'

int rowsAffected = ps.executeUpdate();
System.out.println(rowsAffected + " row(s) inserted.");
```

C. CallableStatement

Used to execute **Stored Procedures** stored in the database.

```
// Calling a procedure named 'getStudent'
CallableStatement cs = con.prepareCall("{call getStudent(?, ?)}");
```

```
cs.setInt(1, 101); // Input parameter
cs.registerOutParameter(2, Types.VARCHAR); // Output parameter
cs.execute();
System.out.println("Student Name: " + cs.getString(2));
```

6. Handling the ResultSet

The ResultSet interface holds data retrieved from a database after executing a SELECT query. It acts like an iterator.

Navigation Methods

- next(): Moves cursor forward one row.
- previous(): Moves cursor to the previous row.
- first() / last(): Moves to first/last row.
- absolute(int row): Moves to a specific row number.

ResultSet Types

When creating a statement, you can define how the ResultSet behaves:

1. **TYPE_FORWARD_ONLY (Default):** Cursor moves only forward.
2. **TYPE_SCROLL_INSENSITIVE:** Scrollable, but doesn't show changes made by others.
3. **TYPE_SCROLL_SENSITIVE:** Scrollable and shows live changes.

Concurrency Modes

1. **CONCUR_READ_ONLY (Default):** Cannot update data via ResultSet.
2. **CONCUR_UPDATABLE:** Allows updating database rows directly via Java code.

Example: Reading Data

```
String query = "SELECT id, name FROM Students";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);

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

7. CRUD Operations Summary

Operation	Method Used	SQL Example
Create (Insert)	executeUpdate()	INSERT INTO ...
Read (Select)	executeQuery()	SELECT * FROM ...
Update	executeUpdate()	UPDATE ... SET ...
Delete	executeUpdate()	DELETE FROM ...

> **Note:** executeUpdate() returns an int (number of rows affected), while executeQuery() returns a ResultSet.

8. Transaction Management

By default, JDBC is in **Auto-Commit** mode (every SQL statement is committed immediately). For complex operations (e.g., banking transfers), you must manage transactions manually to ensure **ACID** properties (Atomicity, Consistency, Isolation, Durability).

Steps for Manual Transaction

1. **Disable Auto-Commit:** con.setAutoCommit(false);
2. **Perform Operations:** Run multiple SQL updates.
3. **Commit:** If all succeed, save changes using con.commit();.
4. **Rollback:** If any fail, revert changes using con.rollback();.

Example: Bank Transfer

```
try {
   con.setAutoCommit(false); // Start transaction

   // Debit Account A
   PreparedStatement debit = con.prepareStatement("UPDATE Accounts SET balance =
balance - 500 WHERE id = 1");
   debit.executeUpdate();

   // Credit Account B
   PreparedStatement credit = con.prepareStatement("UPDATE Accounts SET balance =
balance + 500 WHERE id = 2");
   credit.executeUpdate();

   con.commit(); // Save changes
   System.out.println("Transfer Successful");
```

```
} catch (SQLException e) {
    con.rollback(); // Undo changes if error occurs
    System.out.println("Transaction Failed");
}
```

9. Multithreading in JDBC

In a multithreaded environment (like a web server handling multiple users), you **cannot** share a single Connection object across threads safely.

Best Practices for Multithreading

- **Isolation:** Each thread should obtain its own Connection.
- **Avoid Race Conditions:** Never share Statement or ResultSet objects between threads.
- **Connection Pooling:** Instead of creating a new connection for every thread (which is expensive), use a **Connection Pool** (like HikariCP or Apache DBCP). Threads borrow a connection, use it, and return it to the pool.

Example Logic (Thread-Safe):

```
public void run() {
    // Every thread gets a NEW connection instance
    try (Connection con = DriverManager.getConnection(url, user, pass)) {
        // Perform DB operations specific to this thread
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

10. Modern Best Practices (2024/2025)

1. **Use try-with-resources:** As shown in the examples, this ensures Connection, Statement, and ResultSet are closed automatically, preventing memory leaks.
2. **Use Connection Pooling:** Never manually manage connections in production; use a pool (e.g., HikariCP).
3. **Avoid SELECT *:** Always specify column names to reduce network load.
4. **Use PreparedStatement:** Always prefer this over standard Statement for security.