JDBC

JDBC Architecture & Interfaces

Definition:
JDBC (Java Database Connectivity) is a **Java API** that allows Java applications to **connect and interact with relational databases** like MySQL, Oracle, PostgreSQL, etc.

- It provides **methods for querying and updating** databases using SQL.

Why JDBC is Needed / Purpose

- Enable Java applications to work with databases
- Standard interface, independent of database vendor
- Allows CRUD operations (Create, Read, Update, Delete)
- Supports **transaction management and metadata access**

JDBC Architecture Overview
JDBC follows a **tiered architecture**:

```
Java Application
      |
      V
JDBC API (Interfaces)
      |
      V
JDBC Driver Manager
      |
      V
Database (Oracle, MySQL, etc.)
```

Key Layers

1. **JDBC API (Java side)**
    - Provides **interfaces and classes** like Connection, Statement, ResultSet
    - Developer writes code using these interfaces
2. **JDBC Driver (Database side)**
    - Translates Java calls to database-specific calls
    - Types:
        - **Type 1** – JDBC-ODBC bridge
        - **Type 2** – Native API driver
        - **Type 3** – Network protocol driver
        - **Type 4** – Pure Java driver (most used)
3. **Database**
    - Executes SQL queries and returns results

JDBC Interfaces (Core Components)

Interface	Purpose
DriverManager	Manages database drivers and connections
Connection	Establishes a connection to DB
Statement	Executes SQL queries
PreparedStatement	Executes precompiled SQL queries (safe & fast)
CallableStatement	Executes stored procedures
ResultSet	Holds query results
DatabaseMetaData	Retrieves database information

Interface	Purpose
ResultSetMetaData	Retrieves info about result columns

Flow of JDBC Operations

```
   1. Load the driver
Class.forName("com.mysql.cj.jdbc.Driver");
   2. Establish connection
Connection con = DriverManager.getConnection(url, user, pass);
   3. Create statement
Statement stmt = con.createStatement();
   4. Execute query
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
   5. Process results
while(rs.next()) { System.out.println(rs.getString("name")); }
   6. Close resources
rs.close(); stmt.close(); con.close();
```

Types of Statements

Statement Type	Use Case	Notes
Statement	Simple SQL	Less secure
PreparedStatement	Parameterized SQL	Prevents SQL Injection
CallableStatement	Stored procedures	For DB functions

JDBC Driver Types

Type	Description	Example
Type 1	JDBC-ODBC bridge	Legacy
Type 2	Native API driver	Oracle OCI
Type 3	Network protocol	Middleware driver
Type 4	Pure Java driver	MySQL Connector/J

Most used: Type 4

Advantages of JDBC

- Database-independent API
- Supports multiple SQL databases
- Powerful query execution
- Supports transactions

Limitations of JDBC

- Verbose code (lots of boilerplate)
- Manual resource handling
- No ORM features (use JPA/Hibernate for objects)

Real-Life Analogy

- **DriverManager** → Receptionist assigning DB connection
- **Connection** → Door to the database
- **Statement** → Worker sending instructions
- **ResultSet** → Worker returning results

Common Interview Questions (Cognizant Level)

Q1. What is JDBC?

A. Java API to interact with relational databases.

Q2. Which interface is used to execute SQL queries?
A. Statement or PreparedStatement.
Q3. Difference between Statement & PreparedStatement?
- Statement → Executes dynamic SQL
- PreparedStatement → Precompiled, safer, faster

Q4. What is DriverManager?
A. Manages JDBC drivers and connections.
Q5. Most used JDBC driver type?
A. Type 4 (Pure Java).

One-Line Summary (Quick Revision)
JDBC provides Java applications with a standard API to connect, query, and manage relational databases using interfaces like Connection, Statement, and ResultSet.

JDBC Driver Types & Registration

Definition:
JDBC Drivers are **software components** that allow **Java applications to communicate with a database.**
Driver registration tells the JVM **which driver to use** for database connection.

Why Driver Types & Registration are Important
- Java is **database-independent**, drivers handle vendor-specific translation
- Correct driver ensures **successful connection**
- Registration allows **DriverManager** to manage connections

---

Types of JDBC Drivers

Type	Description	Pros	Cons	Example
Type 1 – JDBC-ODBC Bridge	Converts JDBC calls to ODBC	Easy to use	Dependent on ODBC, slow	Legacy
Type 2 – Native API Driver	Uses database client libraries	Fast	Platform dependent	Oracle OCI
Type 3 – Network Protocol Driver	Sends JDBC calls via middleware	Database independent	Needs middleware	NetDirect
Type 4 – Thin / Pure Java Driver	Converts JDBC calls directly to DB protocol	Fast, portable	Vendor-specific	MySQL Connector/J, Oracle Thin

Most widely used: **Type 4 – Pure Java**

Driver Registration

A. Automatic Registration (Java 6+)
Most modern drivers register themselves automatically if **JAR is in classpath.**
```
Connection con = DriverManager.getConnection(url, user, password);
```
No Class.forName() needed.

B. Manual Registration (Older Java versions)
```
Class.forName("com.mysql.cj.jdbc.Driver");
Connection con = DriverManager.getConnection(url, user, password);
```
Steps:

1. Load driver class using Class.forName()
2. Driver registers itself with DriverManager
3. DriverManager returns a Connection

Real-Life Analogy

- **Driver** → Translator between Java and DB language
- **DriverManager** → Dispatcher that selects which driver to use

---

Advantages of Type 4 Driver

- No native libraries required
- Portable (cross-platform)
- Fastest and commonly used in enterprise apps

Common Interview Questions (Cognizant Level)

Q1. How many JDBC driver types are there?

A. 4 types: Type 1, 2, 3, 4

Q2. Which JDBC driver is platform-independent?

A. Type 4 (Pure Java driver)

Q3. What is driver registration?

A. Process to let JVM know which JDBC driver to use.

Q4. Is Class.forName() mandatory in Java 8+?

A. No, automatic registration works if driver JAR is in classpath.

Q5. Difference between Type 1 and Type 4 driver?

- Type 1 → Uses ODBC, platform-dependent
- Type 4 → Pure Java, vendor-specific, platform-independent

One-Line Summary (Quick Revision)

JDBC driver types define how Java communicates with databases, and registration ensures the JVM knows which driver to use.

Setting Up JDBC Database Driver

Definition:

Setting up a **database driver** means **configuring your Java project** so it can communicate with a database using JDBC.

It involves **adding the driver library** and **registering the driver.**

Why Setting Up Driver is Needed

- JDBC requires a **driver to translate Java calls to SQL**
- Without proper setup, **connection fails**
- Ensures **database independence**

Steps to Set Up JDBC Driver

Step 1: Download JDBC Driver

- Visit database vendor website:
  - MySQL → Connector/J (mysql-connector-java-x.x.x.jar)
  - Oracle → ojdbc8.jar
  - PostgreSQL → postgresql-x.x.x.jar

Step 2: Add Driver to Project

- **IDE (Eclipse/IntelliJ/VS Code)**
  - Add JAR to project build path or dependencies

- **Maven Project**

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.1.0</version>
</dependency>
```

Step 3: Load/Register Driver

A. Automatic Registration (Java 6+)

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

Works if driver JAR is in classpath

B. Manual Registration (Optional)

```
Class.forName("com.mysql.cj.jdbc.Driver");
Connection con = DriverManager.getConnection(url, user, password);
```

Ensures compatibility with older Java versions

Step 4: Establish Connection

```
String url = "jdbc:mysql://localhost:3306/testdb";
String user = "root";
String password = "root";
Connection con = DriverManager.getConnection(url, user, password);
```

Step 5: Verify Setup

```
if (con != null) {
    System.out.println("Database connected successfully!");
} else {
    System.out.println("Connection failed!");
}
```

Real-Life Analogy

- **Driver JAR** → Translator between Java and database language
- **DriverManager** → Dispatcher choosing the right translator
- **Connection** → Door to the database

Tips for Smooth Setup

- Ensure **JAR version matches database version**
- Keep **database server running**
- Use **try-with-resources** to manage Connection safely

Common Interview Questions (Cognizant Level)

Q1. How do you set up a JDBC driver?

A. Download the driver, add JAR to project, load/register driver, and establish connection.

Q2. Do you need Class.forName() in Java 8+?

A. Not mandatory, but works for older versions.

Q3. Which file is added to project for MySQL connection?

A. mysql-connector-java-x.x.x.jar

Q4. How to add driver in Maven project?

A. Add <dependency> for driver in pom.xml.

Q5. What happens if driver is not in classpath?

A. SQLException: No suitable driver found

One-Line Summary (Quick Revision)

Setting up JDBC driver involves downloading the driver JAR, adding it to the project, registering it, and establishing a database connection.

Setting Up JDBC Utility Class

Definition:
A **JDBC Utility Class** is a **helper class** that manages **common database tasks** like **connection creation, closing resources, and driver setup.**

- Reduces repetitive code and improves maintainability.

Why Utility Class is Needed

- Avoids writing DriverManager.getConnection() repeatedly
- Centralizes **DB configuration** (URL, username, password)
- Ensures **proper resource closing**
- Simplifies **CRUD operations** in projects

Key Responsibilities of Utility Class

1. Load/Register JDBC driver
2. Create database connection
3. Close resources (Connection, Statement, ResultSet)
4. Optional: Manage **connection pool** (advanced)

Steps to Set Up a Utility Class

Step 1: Create Class

```
public class DBUtil {
    private static final String URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String USER = "root";
    private static final String PASSWORD = "root";

    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver"); // load driver
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

Step 2: Create Method to Get Connection

```
    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
```

Step 3: Create Methods to Close Resources

```
    public static void close(Connection con, Statement stmt, ResultSet rs) {
        try {
            if(rs != null) rs.close();
            if(stmt != null) stmt.close();
            if(con != null) con.close();
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Using Utility Class in Code

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

```
        Statement stmt = null;
        ResultSet rs = null;

        try {
            con = DBUtil.getConnection();
            stmt = con.createStatement();
            rs = stmt.executeQuery("SELECT * FROM users");
            while(rs.next()) {
                System.out.println(rs.getString("name"));
            }
        } catch(SQLException e) {
            e.printStackTrace();
        } finally {
            DBUtil.close(con, stmt, rs);
        }
    }
}
```

Real-Life Analogy

- **Utility class** → Kitchen helper
- **Connection method** → Provides cooking ingredients
- **Close method** → Cleans up after cooking

Advantages

- Reduces boilerplate code
- Centralized DB management
- Easier maintenance
- Avoids resource leaks

Best Practices

- Use **static methods** for easy access
- Use **try-with-resources** if possible
- Keep **DB credentials in properties file** for security
- Handle exceptions properly

Common Interview Questions (Cognizant Level)

Q1. What is a JDBC utility class?
A. A helper class to manage connections and resources.
Q2. Why use a utility class?
A. To avoid repetitive code and improve maintainability.
Q3. Which resources should be closed?
A. Connection, Statement, ResultSet.
Q4. Can utility class use connection pool?
A. Yes, for high-performance apps.
Q5. What is static block used for?
A. To load JDBC driver when class is loaded.

One-Line Summary (Quick Revision)

JDBC Utility Class centralizes database connection, resource management, and driver setup to simplify coding and avoid repetition.

SQL Injection

Simple Definition
SQL Injection is a **security attack** where a hacker inserts **malicious SQL code** into input fields to **access, modify, or delete database data.**

Why SQL Injection Happens

- Using **dynamic SQL queries**
- Accepting **user input without validation**
- Using **Statement instead of PreparedStatement**

How SQL Injection Works (Step-by-Step)
Vulnerable Code

```
String query = "SELECT * FROM users WHERE username='" + user +
               "' AND password='" + pass + "'";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Malicious Input

```
username: admin
password: ' OR '1'='1
```

Final Query Executed

```
SELECT * FROM users WHERE username='admin' AND password='' OR '1'='1'
```

- '1'='1' is always true → **login bypassed**

Real-Life Analogy

- **Normal lock** → Checks correct key
- **SQL Injection** → Breaking lock by tricking the mechanism instead of using key

---

Types of SQL Injection

1. **Classic SQL Injection** – Manipulating input fields
2. **Blind SQL Injection** – No error shown, attacker guesses response
3. **Union-based Injection** – Uses UNION SELECT to fetch data

---

How to Prevent SQL Injection
Best Solution: PreparedStatement

```
String query = "SELECT * FROM users WHERE username=? AND password=?";
PreparedStatement ps = con.prepareStatement(query);
ps.setString(1, user);
ps.setString(2, pass);
ResultSet rs = ps.executeQuery();
```

- Input treated as **data**, not SQL code

Additional Prevention Techniques

- Input validation
- Use **stored procedures**
- Least privilege DB user
- Use ORM (JPA/Hibernate)
- Avoid displaying SQL errors

Statement vs PreparedStatement (Interview Favorite)

Statement	PreparedStatement
Vulnerable to SQL injection	Safe from SQL injection
Slow	Faster
Dynamic SQL	Precompiled SQL

Advantages of Preventing SQL Injection

- Protects sensitive data
- Prevents data loss
- Improves application security

Common Interview Questions (Cognizant Level)

Q1. What is SQL Injection?
A. A security attack using malicious SQL code via user input.
Q2. Which JDBC interface prevents SQL Injection?
A. PreparedStatement.
Q3. Why is Statement unsafe?
A. It directly concatenates user input into SQL query.
Q4. Can SQL Injection happen in JDBC?
A. Yes, if Statement and dynamic queries are used.
Q5. One best way to avoid SQL Injection?
A. Use PreparedStatement.

One-Line Summary (Quick Revision)

SQL Injection is a security vulnerability where attackers manipulate SQL queries using user input, prevented by using PreparedStatement and input validation.

Callable Statements (JDBC)

Definition:

CallableStatement is a **JDBC interface** used to **execute stored procedures** in a database from Java.

- Allows passing **input/output parameters** and getting results efficiently.

Why CallableStatement is Needed / Purpose

- Stored procedures improve **performance** and **security**
- Reduces **SQL injection risk**
- Encapsulates **complex SQL logic** in DB
- Simplifies **repetitive queries**

Difference: Statement vs PreparedStatement vs CallableStatement

Feature	Statement	PreparedStatement	CallableStatement
SQL Type	Dynamic	Precompiled	Stored Procedure
Input Parameters	No	Yes	Yes (IN, OUT, INOUT)
Security	Vulnerable	Safe	Safe
Reusability	Low	Medium	High

Syntax / Example

Step 1: Create Stored Procedure (MySQL Example)

```
DELIMITER //
CREATE PROCEDURE getUser(IN userId INT)
BEGIN
    SELECT * FROM users WHERE id = userId;
END //
DELIMITER ;
```

Step 2: Java Code to Call Procedure

```
Connection con = DBUtil.getConnection();

// Prepare callable statement
CallableStatement cs = con.prepareCall("{call getUser(?)}");

// Set input parameter
cs.setInt(1, 101);

// Execute
ResultSet rs = cs.executeQuery();

// Process result
while(rs.next()) {
    System.out.println(rs.getString("name"));
}

// Close resources
DBUtil.close(con, cs, rs);
```

CallableStatement with OUT Parameter

```
CREATE PROCEDURE getCount(OUT totalUsers INT)
BEGIN
    SELECT COUNT(*) INTO totalUsers FROM users;
END;
CallableStatement cs = con.prepareCall("{call getCount(?)}");
cs.registerOutParameter(1, Types.INTEGER);
cs.execute();
int count = cs.getInt(1);
System.out.println("Total Users: " + count);
```

Real-Life Analogy

- **Stored Procedure** → Predefined recipe in cookbook
- **CallableStatement** → Following recipe to prepare dish efficiently
- Reduces chance of mistakes (SQL injection)

Advantages

- Faster than dynamic queries
- Reusable & centralized logic
- Supports IN, OUT, INOUT parameters
- Safer (prevents SQL injection)

Best Practices

- Always **close CallableStatement** in finally or try-with-resources
- Use **IN/OUT parameters carefully**
- Avoid dynamic SQL inside stored procedures if possible

Common Interview Questions (Cognizant Level)

Q1. What is CallableStatement?
A. JDBC interface to execute stored procedures.
Q2. Which JDBC statement type is safest for SQL injection?
A. PreparedStatement or CallableStatement.
Q3. Can CallableStatement return multiple ResultSets?
A. Yes, using cs.getMoreResults()
Q4. Difference between CallableStatement and PreparedStatement?

- PreparedStatement → Executes SQL queries
- CallableStatement → Executes stored procedures

Q5. What parameter types are supported?

A. IN, OUT, INOUT

One-Line Summary (Quick Revision)

CallableStatement is used in JDBC to call stored procedures, supporting input/output parameters, improving performance, reusability, and security.

JDBC ResultSet

Definition:

ResultSet is a **JDBC interface** that represents the **data retrieved from a database** after executing a **SELECT query**.

- Think of it as a **pointer to the database table rows**.

Why ResultSet is Needed

- Holds **query results** in memory
- Allows **iterating over rows**
- Enables **reading, updating, or scrolling** data
- Works with Statement, PreparedStatement, CallableStatement

Types of ResultSet

Type	Description	Scrollable?	Updatable?
TYPE_FORWARD_ONLY	Moves only forward	No	Optional
TYPE_SCROLL_INSENSITIVE	Can scroll, does **not** see DB changes	Yes	No
TYPE_SCROLL_SENSITIVE	Can scroll, sees DB changes	Yes	Optional

---

Concurrency Modes

Mode	Description
CONCUR_READ_ONLY	Read-only access
CONCUR_UPDATABLE	Can update DB directly from ResultSet

Cursor Movement Methods

Method	Purpose
next()	Move forward one row
previous()	Move backward one row
first()	Move to first row
last()	Move to last row
absolute(int row)	Move to specific row
relative(int row)	Move relative to current row
isBeforeFirst() / isAfterLast()	Check position

Retrieving Data from ResultSet

```
while(rs.next()) {
    int id = rs.getInt("id"); // By column name
    String name = rs.getString(2); // By column index
    System.out.println(id + " - " + name);
```

```
}
```
Methods to get data:

- getInt(), getString(), getDouble(), getDate(), etc.

Updating Data in ResultSet (Updatable Mode)

```
rs.moveToInsertRow();
rs.updateInt("id", 104);
rs.updateString("name", "Ali");
rs.insertRow();
```

Real-Life Analogy

- **ResultSet** → Spreadsheet with rows & columns
- next() → Move to next row
- getString() → Read column data

Advantages

- Simplifies reading query results
- Can scroll forward/backward (if scrollable)
- Supports read-only or updatable operations

Limitations

- Consumes memory for large result sets
- Requires proper closing to avoid leaks
- Not suitable for huge datasets without streaming

Common Interview Questions (Cognizant Level)

Q1. What is ResultSet?

A. Interface representing data from a SELECT query.

Q2. Difference between TYPE_FORWARD_ONLY and TYPE_SCROLL_INSENSITIVE?

- Forward-only → Only next()
- Scroll-insensitive → Scrollable, does not see DB changes

Q3. How to get data from ResultSet?

A. Using getInt(), getString(), etc., by column name or index.

Q4. Can ResultSet update database?

A. Yes, if CONCUR_UPDATABLE is used.

Q5. How to close ResultSet?

A. rs.close();

One-Line Summary (Quick Revision)

ResultSet holds the rows returned by a query, allowing navigation, reading, and optional updating of database data.

Navigating ResultSet Rows (JDBC)

Definition:

Navigating ResultSet Rows means **moving the cursor** through the rows of a ResultSet to **read or update data.**

- JDBC provides methods to move forward, backward, or jump to specific rows.

Why Navigating ResultSet is Needed

- To process query results in **custom order**
- Read previous, next, first, or last row

- Perform **updates or deletions** on specific rows
- Supports scrollable and updatable operations

ResultSet Types (Affect Navigation)

Type	Scrollable?	Notes
TYPE_FORWARD_ONLY	No	Can only use next()
TYPE_SCROLL_INSENSITIVE	Yes	Scrollable, **DB changes not visible**
TYPE_SCROLL_SENSITIVE	Yes	Scrollable, **DB changes visible**

Cursor Movement Methods

Method	Purpose
next()	Move to **next row** (most common)
previous()	Move to **previous row**
first()	Move to **first row**
last()	Move to **last row**
absolute(int row)	Move to **specific row number**
relative(int row)	Move **relative to current row**
isBeforeFirst()	Check if cursor is **before first row**
isAfterLast()	Check if cursor is **after last row**

Example: Forward and Backward Navigation

```
Statement stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

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

// Backward
rs.afterLast(); // Move after last row
while(rs.previous()) {
    System.out.println(rs.getInt("id") + " " + rs.getString("name"));
}

// Jump to specific row
rs.absolute(2);
System.out.println("Second row: " + rs.getString("name"));
```

Updating Rows While Navigating (Updatable ResultSet)

```
Statement stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

// Move to specific row
rs.absolute(3);
rs.updateString("name", "Ali Updated");
rs.updateRow();
```

Real-Life Analogy

- **ResultSet** → Spreadsheet
- next() → Move down one row
- previous() → Move up one row
- absolute(5) → Jump to row 5

Advantages

- Flexible access to data
- Supports forward, backward, and random access
- Works with scrollable and updatable ResultSets

Limitations

- Scrollable ResultSets use more memory
- Only works if Statement is **scrollable**
- Not suitable for **very large datasets** without streaming

Common Interview Questions (Cognizant Level)

Q1. How to move to the previous row in ResultSet?
A. Use rs.previous() (requires scrollable ResultSet)
Q2. How to jump to a specific row?
A. Use rs.absolute(rowNumber)
Q3. Can TYPE_FORWARD_ONLY ResultSet move backward?
A. No
Q4. Which ResultSet type is sensitive to DB changes?
A. TYPE_SCROLL_SENSITIVE
Q5. How to update a row while navigating?
A. Use CONCUR_UPDATABLE ResultSet + updateRow()

One-Line Summary (Quick Revision)
Navigating ResultSet rows lets you move the cursor forward, backward, or to a specific row to read or update database data efficiently.

JDBC Statement vs PreparedStatement

Definition
Statement:

- A JDBC interface used to **execute static SQL queries**.
- Accepts SQL as a **String**.

PreparedStatement:

- A JDBC interface for **precompiled SQL queries** with **parameters** (?).
- Improves **performance and security.**

Why They Are Needed

- **Statement:** Quick and simple execution of SQL queries.
- **PreparedStatement:**
  - Prevents **SQL injection**
  - Supports **parameterized queries**
  - Improves **performance** by precompiling queries

Syntax Examples
A. Statement

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

```
String sql = "SELECT * FROM users WHERE id = 101";
ResultSet rs = stmt.executeQuery(sql);
```

B. PreparedStatement

```
String sql = "SELECT * FROM users WHERE id = ?";
PreparedStatement ps = con.prepareStatement(sql);
ps.setInt(1, 101);
ResultSet rs = ps.executeQuery();
```

SQL Injection Example

Vulnerable (Statement):

```
String sql = "SELECT * FROM users WHERE username='" + user + "' AND password='" + pass
+ "'";
```

Safe (PreparedStatement):

```
String sql = "SELECT * FROM users WHERE username=? AND password=?";
PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1, user);
ps.setString(2, pass);
```

Differences: Statement vs PreparedStatement

Feature	Statement	PreparedStatement
Query Type	Static SQL	Parameterized SQL
SQL Injection	Vulnerable	Safe
Performance	Slower	Faster (precompiled)
Reusability	No	Yes (set parameters multiple times)
Parameters	Not supported	Supported (IN, OUT via CallableStatement)

Real-Life Analogy

- **Statement** → Writing letter from scratch every time
- **PreparedStatement** → Using a **template**, just filling blanks

---

Advantages of PreparedStatement

- Safer against SQL injection
- Faster for repeated queries
- Cleaner and more maintainable code

Example: Reusing PreparedStatement

```
String sql = "INSERT INTO users(name, age) VALUES (?, ?)";
PreparedStatement ps = con.prepareStatement(sql);

ps.setString(1, "Ali");
ps.setInt(2, 22);
ps.executeUpdate();

ps.setString(1, "Ahmed");
ps.setInt(2, 25);
ps.executeUpdate();
```

Best Practices

- Always **close Statement / PreparedStatement**
- Use **PreparedStatement** for **user input queries**
- Use batch operations for **multiple inserts/updates**

Common Interview Questions (Cognizant Level)

Q1. What is the difference between Statement and PreparedStatement?

A. Statement executes static SQL, PreparedStatement executes parameterized SQL and prevents SQL injection.

Q2. Which one is safer for user input?

A. PreparedStatement

Q3. Can PreparedStatement be reused?

A. Yes, by setting new parameters

Q4. Why is PreparedStatement faster than Statement?

A. Query is **precompiled**, reducing parsing time

Q5. How to prevent SQL Injection?

A. Use PreparedStatement or CallableStatement

One-Line Summary (Quick Revision)

Statement executes static SQL queries, while PreparedStatement executes parameterized queries safely and efficiently.
