JDBC

📌 Subtopic 1: JDBC Introduction

What problem does JDBC solve?

Imagine this setup:

- Your **Java application** speaks Java.
- Your **Database** (Postgres, MySQL, Oracle) speaks SQL.
- They don't understand each other directly.

So JDBC is the **translator + bridge** between:

```
Java Program  ⇄  JDBC  ⇄  Database
```

What is JDBC?

JDBC = Java Database Connectivity

It is:

- A **standard API** (set of interfaces & classes)
- Provided by **Java**
- Used to **connect Java applications to databases**
- Lets you:
  - Open a connection
  - Send SQL queries
  - Read results
  - Insert / update / delete data

Think of JDBC as a **universal plug socket**.

Different databases plug in using their own adapters (drivers).

Why do we need JDBC?

Without JDBC:

- Java cannot talk to a database
- No way to run SQL from Java

With JDBC:

- Same Java code style
- Works with **any database**
- Only the **driver changes**

Example:

```
// Java code stays same
Connection con = DriverManager.getConnection(...);

// Database can be Postgres / MySQL / Oracle
```

JDBC is NOT a database

Important clarity (interview favorite):

❌ JDBC is NOT:

- A database
- A framework
- A tool like Hibernate

✅ JDBC IS:

- A **low-level API**
- Requires you to **write SQL manually**
- Gives **full control** but more code

Hibernate / JPA sit **on top of JDBC** and hide this complexity.

JDBC Architecture (High Level)

```
Java Application
      |
      |  (JDBC API)
      |
JDBC Driver (Postgres Driver)
      |
Database (PostgreSQL)
```

- **JDBC API** → Interfaces (Connection, Statement, ResultSet)
- **JDBC Driver** → Database-specific implementation
- **Database** → Actual data store

Real-life analogy 🧠

- Java app = You
- Database = Person speaking French
- JDBC = English
- JDBC Driver = Translator who knows English + French

You always speak English (JDBC).
Translator changes depending on country (Postgres, MySQL).

Key takeaway (remember this line)

JDBC is a **standard Java API** that allows Java applications to connect to and interact with databases using SQL.

📌 Subtopic 2: Postgres Setup

Now let's ground JDBC in something real: **PostgreSQL**.

What is PostgreSQL?

- Open-source **relational database**
- Very popular in **enterprise + finance systems**
- Strong ACID compliance
- Widely used with Java, Spring, Hibernate

What do we need before using JDBC with Postgres?

Before writing **any Java code**, we must have:

1. ✅ PostgreSQL installed and running
2. ✅ A database created
3. ✅ A user + password
4. ✅ A table with some data

JDBC cannot work if the database itself isn't ready.

Typical Postgres setup flow

```
Install Postgres
   ↓
Create Database
   ↓
Create User
   ↓
Create Table
   ↓
Insert Sample Data
```

Example (very basic)

Let's say we create:

Database

```
db name: companydb
```

User

```
username: postgres
password: admin
```

Table

```
CREATE TABLE employee (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    salary INT
);
```

Sample data

```
INSERT INTO employee VALUES (1, 'Sagar', 50000);
INSERT INTO employee VALUES (2, 'Rahul', 60000);
```

At this point:

- Database is ready
- JDBC can now connect to it

How JDBC “sees” Postgres

JDBC doesn’t care:

- How you created DB
- GUI (pgAdmin) or terminal

It only needs:

- **URL**
- **Username**
- **Password**

Example JDBC URL:

`jdbc:postgresql://localhost:5432/companydb`

Break it mentally:

- jdbc → JDBC protocol
- postgresql → driver type
- localhost → DB host
- 5432 → Postgres port
- companydb → DB name

Common beginner mistakes 🚨

- Postgres not running
- Wrong port (default is **5432**)
- Wrong DB name
- Wrong username/password
- Table doesn’t exist

When JDBC fails, **90% issues are here**, not Java code.

📌 Subtopic 3: JDBC Steps

Now let’s talk about **how JDBC actually works step by step**.
This is a **very important interview topic** and also how you should think while coding.

The 6 Core JDBC Steps (Golden Flow)

Whenever you use JDBC, the flow is almost always this:

```
1. Load Driver
2. Create Connection
3. Create Statement
4. Execute Query
5. Process Result
6. Close Resources
```

Don’t memorize blindly. Let’s understand **why each step exists**.

1️⃣ Load the JDBC Driver

Purpose:

- Tells Java **which database** it is talking to
- Registers the driver with JDBC

Older way:

```
Class.forName("org.postgresql.Driver");
```

Modern Java (JDBC 4+):

- Driver auto-loads from classpath
- This step is **optional**

Still asked in interviews because it shows understanding.

2️⃣ Create Connection

Purpose:

- Establishes a **live session** with the database

```
Connection con = DriverManager×getConnection(
    "jdbc:postgresql://localhost:5432/companydb",
    "postgres",
    "admin"
);
```

Think of Connection as:

- A **telephone call** to the DB
- Expensive resource
- Should be reused or closed properly

3️⃣ Create Statement

Purpose:

- Vehicle to send SQL to the database

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

Later we'll see:

- Statement
- PreparedStatement
- CallableStatement

4️⃣ Execute Query

Purpose:

- Actually run SQL

For SELECT:

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

For INSERT / UPDATE / DELETE:

```
int count = stmt.executeUpdate("UPDATE employee SET
salary=70000 WHERE id=1");
```

5️⃣ Process Result

Purpose:

- Read data row by row

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

ResultSet is like a **cursor** moving over rows.

6️⃣ Close Resources

Purpose:

- Prevent memory leaks
- Release DB connections

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

Order matters:
ResultSet → Statement → Connection

Mental model 🧠

```
Connection = Door to DB
Statement  = Pen to write SQL
ResultSet  = Notebook with results
```

📌 Subtopic 4: Connecting Java and DB

Now we'll put **everything together** and actually connect Java to Postgres.
This is the **first real "JDBC moment"**.

What does "connecting" really mean?

When we say:
"Java is connected to DB"
It means:

- Java opened a **network connection**
- Database authenticated the user
- A **session** is created
- SQL commands can now flow

All of this is represented by **one object**:
👉 Connection

Basic connection code (minimum working example)

```
import java.sql.Connection;
import java.sql.DriverManager;

public class JdbcDemo {
    public static void main(String[] args) {
        try {
            Connection con = DriverManager×getConnection(
                "jdbc:postgresql://localhost:5432/
companydb",
                "postgres",
                "admin"
            );

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

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

If this prints:

```
Connected to database!
```

🎉 Your JDBC setup is correct.

What happens internally?

1. DriverManager checks all loaded drivers
2. Finds Postgres driver
3. Driver validates:
    - URL
    - Username
    - Password
4. Database returns a connection handle
5. JDBC wraps it as Connection object

Why is Connection expensive?

Because it:

- Opens a socket
- Authenticates user
- Allocates DB resources

That's why in real systems:

- We **reuse** connections
- Use **connection pools** (HikariCP, etc.)

But JDBC basics first.

Common connection issues 🚨

- DB not running
- Wrong port
- Typo in DB name
- Authentication failure

Typical error:

```
Connection refused
FATAL: password authentication failed
```

These are **environment issues**, not Java bugs.

📌 Subtopic 5: Execute and Process

Now that we're connected, the real work begins:
sending SQL to the database and handling results.

Big picture flow

```
Connection
   ↓
Statement
   ↓
Execute SQL
   ↓
ResultSet / Update Count
```

Connection alone does nothing.
We need a **Statement** to fire SQL bullets 🎯

Step 1: Create a Statement

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

Purpose:

- Acts as a **carrier** for SQL
- Sends SQL text to the database

Step 2: Execute SQL

There are **two main execution methods**:

🔷 executeQuery() → SELECT

```
ResultSet rs = stmt.executeQuery(
```

```
    "SELECT id, name, salary FROM employee"
);
```

- Used only for **SELECT**
- Returns ResultSet

🔷 executeUpdate() → INSERT / UPDATE / DELETE

```
int rows = stmt.executeUpdate(
    "UPDATE employee SET salary = 70000 WHERE id = 1"
);
```

- Used for DML
- Returns number of rows affected

Step 3: Process ResultSet

ResultSet works like a **cursor**.
Initially:

- Cursor is **before first row**

You must call:

```
rs.next()
```

Example:

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

    System.out.println(id + " " + name + " " + salary);
}
```

Important ResultSet rules 🚨

- Column names must match DB
- Data type must be compatible
- Cursor moves **one-way** by default

Mental model 🧠

- Statement → Messenger
- executeQuery() → Asks question
- ResultSet → Answer sheet
- rs.next() → Move to next line

📌 Subtopic 6: Fetching All Records

Now let's slow this down and **deeply understand how records are fetched**, because this is where many beginners get confused.

What does "fetching all records" really mean?

When you run:

```
SELECT * FROM employee;
```

The database:

- Finds matching rows
- Sends them back **one by one**
- JDBC does NOT load everything at once

Instead:

- You iterate row-by-row using ResultSet

Step-by-step flow

```
Statement stmt = con×createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employee");

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

    System.out.println(id + " " + name + " " + salary);
}
```

What rs.next() actually does

- Moves cursor to next row
- Returns:
    - true → row exists
    - false → no more rows

Cursor positions:

```
Before first row
↓ rs.next()
Row 1
↓ rs.next()
Row 2
↓ rs.next()
After last row (false)
```

Different ways to read data

By **column name** (recommended):

```
rs.getString("name");
```

By **column index** (1-based):

```
rs.getString(2);
```

Why name is better:

- Safer
- Column order change doesn't break code

Common beginner mistakes 🚨

❌ Forgetting rs.next()
❌ Using wrong column name
❌ Using wrong datatype
❌ Assuming all rows are loaded automatically

Interview insight 💡

If interviewer asks:
"How does JDBC fetch records?"
Say:
JDBC uses a ResultSet cursor and fetches rows sequentially using rs.next().

📌 Subtopic 7: CRUD Operations (Create, Read, Update, Delete)

Now we connect JDBC to **real business work**.
Almost every backend application does **CRUD**.

What is CRUD?

Operation	SQL	JDBC Method
Create	INSERT	executeUpdate()
Read	SELECT	executeQuery()
Update	UPDATE	executeUpdate()
Delete	DELETE	executeUpdate()

Notice:

- Only **SELECT** returns ResultSet
- Others return **rows affected**

CREATE (Insert)

```
String sql = "INSERT INTO employee VALUES (3, 'Amit',
55000)";
int rows = stmt.executeUpdate(sql);
```

If:

`rows = 1`
→ Insert successful

READ (Select)

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

Handled via:
`while (rs.next()) { ... }`

UPDATE

```
String sql = "UPDATE employee SET salary = 65000 WHERE id = 2";
int rows = stmt.executeUpdate(sql);
```

DELETE

```
String sql = "DELETE FROM employee WHERE id = 3";
int rows = stmt.executeUpdate(sql);
```

Why executeUpdate() name?
Because it:
- Modifies DB state
- Returns count of affected rows
- Not actual data

Transaction note (important but light)
By default:
- JDBC runs in **auto-commit mode**
- Every statement is immediately committed

Later you'll see:

```
con.setAutoCommit(false);
con.commit();
```

📌 Subtopic 8: Problems with Statement
Now we come to a **very important turning point** in JDBC learning.
This explains **why PreparedStatement exists**.

What is Statement?
`Statement stmt = con×createStatement();`
- Sends **raw SQL strings** to DB
- SQL is built using **string concatenation**

Example:

```
String sql =
  "SELECT * FROM employee WHERE name = '" + name + "'";
```

Looks simple… but dangerous 😬

🚨 Problem 1: SQL Injection (Very serious)

If user input is:

```
' OR 1=1 --
```

Final SQL becomes:

```
SELECT * FROM employee WHERE name = '' OR 1=1 --
```

Meaning:

- Condition always true
- Entire table exposed

This is **how real systems get hacked**.

🚨 Problem 2: Performance

With Statement:

- DB parses SQL **every time**
- No query reuse
- Slower for repeated execution

🚨 Problem 3: Readability & Bugs

```
"INSERT INTO employee VALUES (" + id + ", '" + name + "', "
+ salary + ")"
```

- Hard to read
- Quotes easily break
- Error-prone

🚨 Problem 4: Type safety

- Everything becomes string
- DB has to infer types
- More runtime errors

Interview-ready line 💡

Statement is prone to SQL injection, has poor performance for repeated queries, and leads to messy, error-prone code.

Absolutely correct ✔️
That's the **main killer issue**.

SQL Injection is the biggest security risk when using Statement with string concatenation.

📌 Subtopic 9 : PreparedStatement

This is the **solution** JDBC gives to all the problems we just discussed.
Think of this as the **professional way** to write JDBC code.

What is PreparedStatement?

- A **precompiled SQL statement**
- Uses **placeholders (?)** instead of string concatenation
- SQL and data are sent **separately** to the DB

```
PreparedStatement ps =
    con.prepareStatement(
        "SELECT * FROM employee WHERE name = ?"
    );
```

How data is set

```
ps.setString(1, "Sagar");
```

Indexing starts from **1**, not 0.
Then execute:

```
ResultSet rs = ps.executeQuery();
```

Why is this secure?

Because:

- SQL structure is fixed
- User input is treated as **data**, not code
- Injection becomes impossible

Performance advantage 🚀

- SQL compiled **once**
- Execution plan reused
- Much faster in loops / batch operations

Clean CRUD example

```
PreparedStatement ps =
    con.prepareStatement(
        "INSERT INTO employee (id, name, salary) VALUES
(?, ?, ?)"
    );

ps.setInt(1, 4);
```

```
ps.setString(2, "Neha");
ps.setInt(3, 70000);

ps.executeUpdate();
```

Readable, safe, clean.

Statement vs PreparedStatement (Mental Table)

Feature	Statement	PreparedStatement
SQL Injection	❌ Yes	✅ No
Performance	❌ Slower	✅ Faster
Readability	❌ Messy	✅ Clean
Reusability	❌ No	✅ Yes

Final interview punchline 🥊

Always prefer PreparedStatement over Statement for security, performance, and maintainability.

🎯 Golden interview answer

JDBC is the low-level API.
Spring JDBC reduces boilerplate but keeps SQL.
Hibernate is an ORM that maps objects to tables.
Spring JPA is Spring's abstraction over JPA using Hibernate internally.