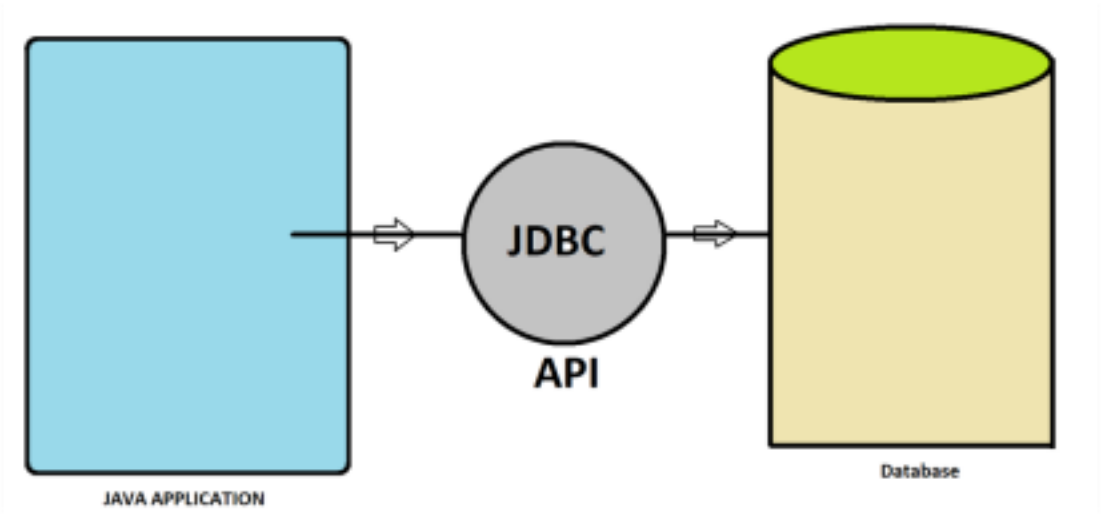
Java DatabaseConnectivity (JDBC)

DAY 1:

What is JDBC?

- **JDBC (Java Database Connectivity) is an API (Application Programming Interface) provided by Java to enable Java applications to interact with relational databases.**
- **NOTE: JDBC (Java Database Connectivity) is a part of J2SE/Java SE (Java Standard Edition).**



- **When we say "interact with relational databases", it means your Java program (using JDBC) can perform all the CRUD operations:**

CRUD = Create, Read, Update, Delete:

Operation SQL Keyword What it Does • Create INSERT Add new data

into a table

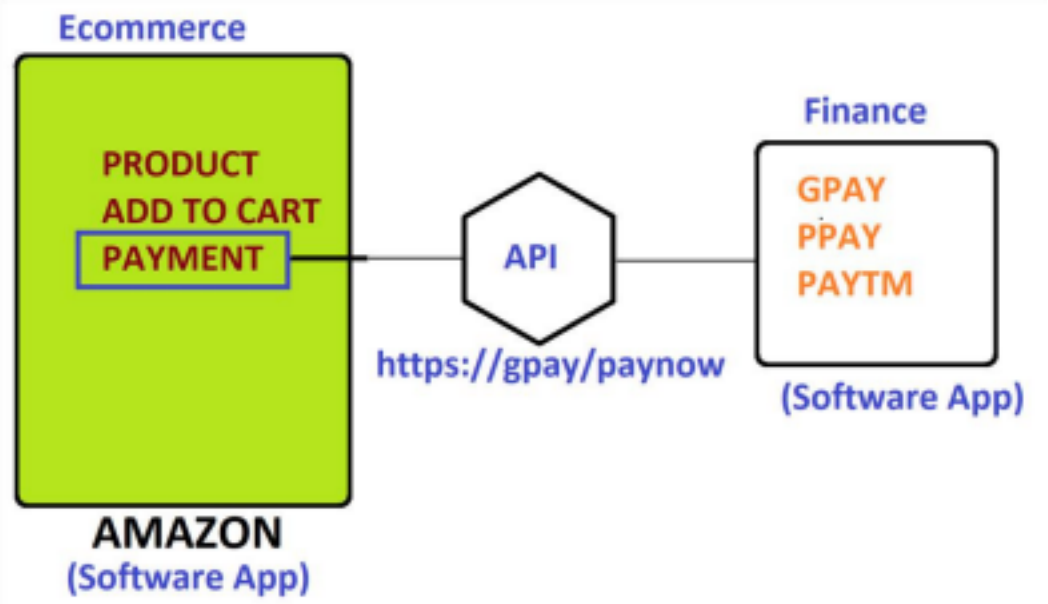
- Read SELECT Fetch data from a table
- Update UPDATE Modify existing data in a table
- Delete DELETE Remove data from a table

What is an API?

- **An API is a set of rules that allow different software applications to communicate with each other. Think of it like a bridge that connects two systems and lets them share data or services.**

- **An API (Application Programming Interface) is a software concept — it is a set of rules, classes, methods, or interfaces that allow two software systems to communicate.**

- **"API" refers to the contract/interface(ENG) that your code exposes — not the actual implementation.**
- **Sometimes an API is just a specification, and sometimes it comes with an implementation.**



- The JDBC API is a set of interfaces and classes in the java.sql and javax.sql packages that allow Java programs to connect to, communicate with, and perform operations on relational databases.

- JDBC API Doesn't Contain Implementation

Why JDBC API Doesn't Contain Implementation? - JDBC is just a set of interfaces, not a complete implementation. Why? Because Java wanted to support connectivity with all relational databases, but it doesn't know the internal communication protocols of each one.

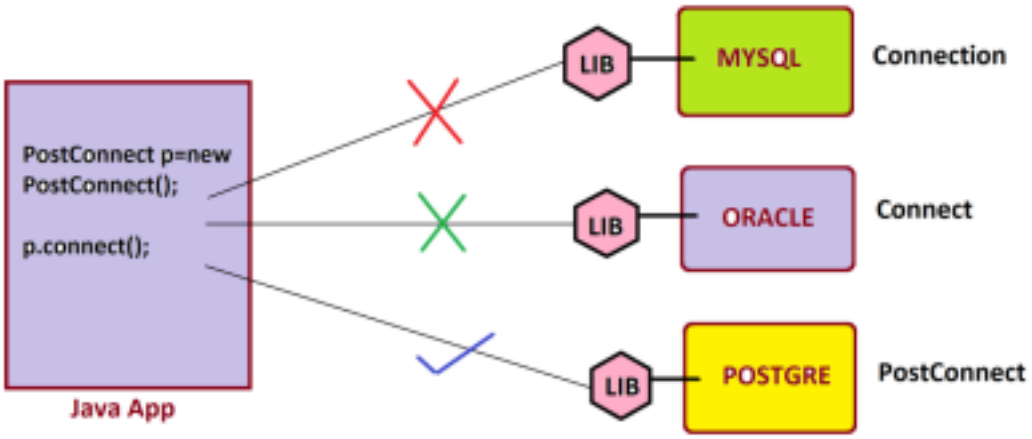
- Why Vendors Should Not Provide Random Libraries (Without a Standard) ?

1. Every vendor would have different class names, method names, and connection logic.

2. Developers would have to learn a new set of methods for each database.

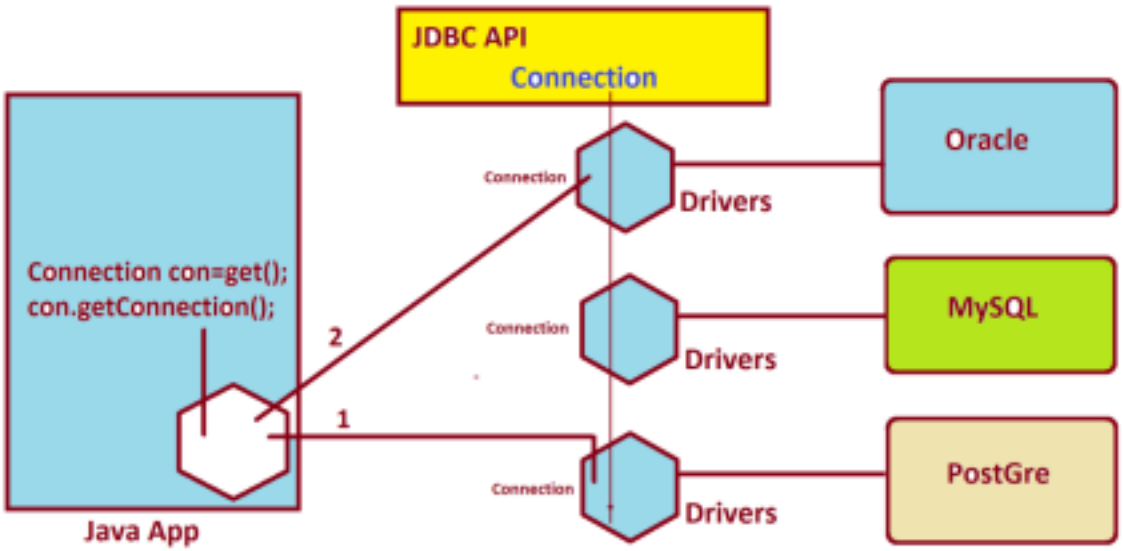
3. Code would break if you tried to switch from one database to another.

4. This creates tight coupling and no portability.



- ***If Java had to handle the implementation itself, it would need to support every database (like MySQL, Oracle, PostgreSQL, etc.) separately — which is not scalable.***
- **So instead, Java introduced the JDBC API as a standard set of interfaces (like Connection, Statement, ResultSet, etc.).**

- **Now, each database vendor provides its own JDBC driver — a library that implements these interfaces according to how their database works internally.**



This approach solves a big problem:

- **Earlier, if every vendor provided their own custom library with different methods, developers would have to learn and rewrite code for each database. There was no standard.**

With JDBC:

- **You write code using the standard JDBC interfaces.**
- **When switching databases, you just change the driver and connection URL.**
- **The rest of the code remains the same.**
- **This ensures database independence and reduces vendor lock-in.**

NOTE:

Why is it called a "Driver" and not just a "Library"?

- ***A driver is a special kind of library whose main job is to "drive" or***

mediate communication between two systems — usually, your program and an external resource (like a database or hardware).

DAY 2:

Disadvantages of File I/O (Compared to JDBC / DBMS)

1.Lack of Security

- **File I/O is not secure by default. Anyone with access to the file system can view or modify data easily.**

2.Data Redundancy

- **There may be duplicate data entries, as file systems do not enforce rules to prevent duplication.**

3.Data Inconsistency

- **If the same data is stored in multiple files or locations, it can become inconsistent.**

Example:

- **CIVIL Dept: 11 Raju 90888 CSK**
- **Library: 11 Raju 80888 CSK**
- **The phone numbers are different for the same person, leading to inconsistency.**

4.Atomicity Problem

- **File I/O does not support atomic transactions.**

```
int key = s.nextInt();
if(key < 100) {
 // Cancel file delete
}
```

NOTE:

- **An atomic transaction means a series of operations that are treated as a single, indivisible unit of work. It either completes entirely or doesn't happen at all — there's no in-between.**

5.Lack of Data Integrity

- File systems cannot enforce constraints like NOT NULL, UNIQUE, FOREIGN KEY, etc. As a result, invalid or inconsistent data may be entered.

6.Difficult Data Retrieval

- No query language (like SQL) to search/filter data.
- You must write custom logic to find even simple data.

7.Manual Backup & Recovery

- No automatic backup or recovery features.
- Restoring lost data is complex.

Why File I/O Was Created / Where It Is Still Useful ?

- File I/O is mainly used to work directly with files on your system. It helps you read from or write to files like:

a) Text files (.txt, .csv)
b) Image files (.jpg, .png)
c) PDFs, logs, documents, etc.

Common Use Cases:

a) Storing app configuration (e.g., config.properties)
b) Saving logs or reports
c) Uploading or saving images/files
d) Exporting/importing data (CSV, JSON, XML)

Simple Use Cases of JDBC:

- Store User Data----------->Save things like name, email, and password in a database.
- Fetch Data----------->Get data from the database like user details or product list.
- Update/Delete Data----------->Change or remove data, like updating a profile or deleting an order.
- Secure Data----------->Keep data safe and correct using rules (like no empty emails).

After understanding the advantages of JDBC (like security, data consistency, and scalability), the next step is to actually establish a connection to a database.

JDBC CONNECTION:

A JDBC (Java Database Connectivity) connection refers to the mechanism used by Java applications to interact with databases. It allows Java programs to connect to databases, execute SQL queries, and process the results.

- **But before we dive into that, let's keep in mind that:**
  The database must be installed and running.
- **We need the JDBC driver for the specific database we are using (like MySQL, Oracle, PostgreSQL, etc.).**
- **And of course, we need the connection details like:**
  a) Database URL
  b) Username
  c) Password

Download Driver Link (mysql connector jar):

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j

XAMPP(MySQL Installation Link):

https://youtu.be/Vfdw-6vYt2A?si=et0PLl_elT0EPu9O

(TIME STAMP: 00:02:00)

So, now let's explore the steps to connect to a database using JDBC.

- Load the JDBC Driver **(optional in JDBC 4.0+).**
- Establish a Connection using **DriverManager.getConnection().**

Before JDBC 4.0, developers had to explicitly load the JDBC driver using Class.forName("driver-class-name") to register it with the DriverManager. This was necessary for the Java application to recognize and use the driver. However, starting from JDBC 4.0 (supported from Java 6 onwards), this step is no longer required. Driver loading is now

automatic if the JDBC driver JAR is present in the classpath.

To View or Edit Classpath in Eclipse:

- Right-click on your project in the Project Explorer.
- Select "Build Path" → "Configure Build Path...".
- Go to the "Libraries" tab.

Here you'll see all JARs and libraries that are part of the classpath.

For example: JRE System Library, any JDBC JARs, other utility JARs, etc.

NOTE:

To check the JDBC version being used

- Find the JDBC Driver JAR file (e.g., mysql-connector-java 8.0.xx.jar):
- Open the JAR file using any ZIP tool or built-in file explorer.
- Navigate to the file: (META-INF/MANIFEST.MF)
- Open this MANIFEST.MF file and look for attributes like:

Specification-Version: 4.2

Implementation-Version: 8.0.33

==========PROGRAM: JDBC-CONNECTION==========

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DriverManager;
public class Launch {
    public static void main(String[] args) {
        try
        {
            //MYSQL DRIVER LOADED AND REGISTERED
//
    Class.forName("com.mysql.cj.jdbc.Driver"); //JDBC
4.0+
            String url="jdbc:mysql://localhost:3306";
            String username="root";
            String password="";
```

```
            //FACTORY DESIGN PATTERN
                                //JDBC API
            Connection con =
DriverManager.getConnection(url,username,password);
             System.out.println(con);
        }
        catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Explanation:

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

- **You're actually getting an object of a class that implements the Connection interface.**
- **This is a classic example of the Factory Design Pattern in action!**

Why is it a Factory Pattern?

In the Factory Pattern, you:

- **Don't create objects directly using new.**
- **Instead, you call a method (like DriverManager.getConnection()) which returns an object, typically of an interface type.**
- **The method decides which concrete class to instantiate based on parameters or configuration.**

==========

==========

After establishing the JDBC connection, the next step is to perform CRUD operations — that is, Create, Read, Update, and Delete data from the database.

To do this, you need to understand some basic SQL commands, such as:

- **CREATE DATABASE** – to create a new database
- **CREATE TABLE** – to define a table structure
- **INSERT INTO** – to add data
- **SELECT** – to read or retrieve data
- **UPDATE** – to modify existing records
- **DELETE** – to remove records

We've provided helpful video links below to help you learn these SQL basics before moving forward with JDBC CRUD operations.

SQL LECTURES:

Lect1:

- **https://youtu.be/FEuugXBe_7k**

Lect2:

- **https://youtu.be/7myDOv67lKk**

Lect3:

- **https://youtu.be/Vfdw-6vYt2A**

Lect4:

- **https://youtu.be/QpMNCDOM9_M**

Lect5:

- **https://youtu.be/PNV9vC7_2zo**

Lect6:

- **https://youtu.be/-ItBuLY3iwA**

Lect7:

- **https://youtu.be/JJq0dSE4gag**

Lect8:

- **https://youtu.be/-FqvJRPKoUk?si=VNYHZhWYu18JeILl**

Lect9:

- **https://youtu.be/epZVdXdns-U**

Lect10:

- **https://youtu.be/L9g-0neh_Gg**

Lect11:

- https://youtu.be/DzgiBCtAt8Q

DAY 3:

Now that we've successfully connected to the database, the next step is to interact with it — this means performing real operations like inserting data, retrieving records, updating existing entries, and deleting unnecessary data. In simple terms, we'll now start doing the actual CRUD operations (Create, Read, Update, Delete) using JDBC.

To interact with the database, the JDBC API provides three main interfaces:

- **Statement Interface**
- **PreparedStatement Interface**
- **CallableStatement Interface**

The implementations of all three interfaces — Statement, PreparedStatement, and CallableStatement — are provided by the JDBC driver vendor (like Oracle, MySQL, PostgreSQL, etc.).

Quick and short difference:

- **Statement: For simple, static SQL queries (no parameters).**
- **PreparedStatement: For parameterized queries (better performance & SQL injection protection).**
- **CallableStatement: For calling stored procedures with input/output parameters.**

Statement Interface

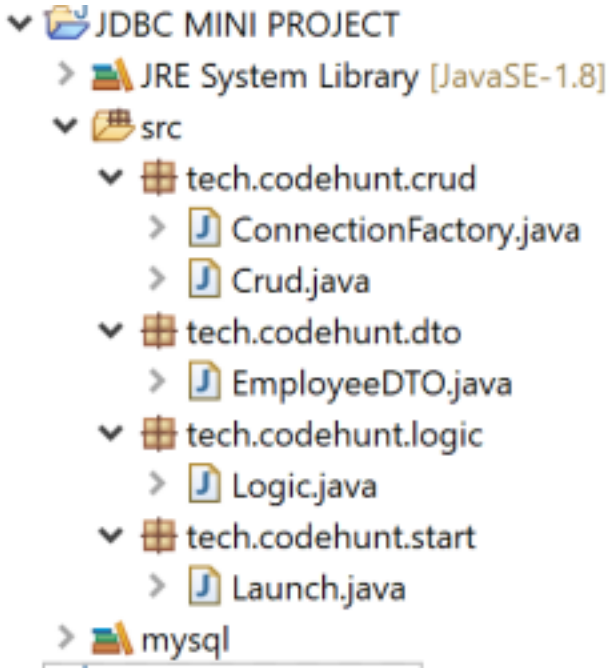
• The Statement interface in JDBC is used to execute static SQL queries (i.e., queries without parameters) against a database. • It doesn't accept input from the user/program at runtime through parameters — you build the full query as a string manually.

EX. " SELECT * FROM employees WHERE department = 'HR' "

Common Use Cases of Statement:

- **Executing simple SQL queries like SELECT, INSERT, UPDATE, or DELETE.**
- **When you do not need to pass dynamic values (parameters) to the SQL query.**
- **Useful for DDL statements (like CREATE TABLE, ALTER TABLE, DROP TABLE).**
- **Less secure for user input – vulnerable to SQL injection if used improperly.**
- **Slower for repeated queries as SQL is not precompiled by the database.**

===PROGRAM: CRUD USING STATEMENT



INTERFACE===

```
package tech.codehunt.start;
import tech.codehunt.logic.Logic;
public class Launch {

    public static void main(String[] args) {

        Logic logic = new Logic();
        logic.doStart();

    }
}
```

```java
package tech.codehunt.logic;
import java.util.Scanner;
import tech.codehunt.crud.Crud;
import tech.codehunt.dto.EmployeeDTO;
public class Logic {

    private final int INSERT_DATA=1;
    private final int READ_DATA=2;
    private final int UPDATE_DATA=3;
    private final int DELETE_DATA=4;
    private final int EXIT=5;
    private static final int MAX_ATTEMPTS=3;
    private String username;
    private String password;
    private String fullname;
    private String address;
    private int salary;
    private Crud crud;

    public Logic() {
        crud=new Crud();
    }


    public void doStart() {

        Scanner scanner = new Scanner(System.in);
        int attempt=0;

        while (true) {


    System.out.println("\n=========MENU=========");
                System.out.println("Press-1 : INSERT
DATA");
                System.out.println("Press-2 : READ
DATA");
                System.out.println("Press-3 : UPDATE
DATA");
                System.out.println("Press-4 : DELETE
DATA");
               System.out.println("Press-5 : EXIT\n");

               System.out.print("Enter Your Choice: ");
```

```
                int choice = 0;

                try {
                    choice = scanner.nextInt();
                } catch (Exception e) {
                     System.out.print("Invalid Input!
Please take a number from 1 to 5.\n");
                    scanner.nextLine();

                    attempt++;
                    if(attempt>=MAX_ATTEMPTS) {
                         System.out.print("You have
reached the limit");
                        scanner.close();
                        return;
                    }
                    continue;
                }

                if (choice < 1 || choice > 5) {
                     System.out.print("Invalid Input!
Please take a value from 1 to 5.\n");
                    attempt++;
                    if(attempt>=MAX_ATTEMPTS) {
                         System.out.print("You have
reached the limit");
                        scanner.close();
                        return;
                    }
                }

                switch (choice) {
                case INSERT_DATA:

    System.out.println("************INSERT
DATA***********");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    System.out.println("ENTER PASSWORD");
                    password=scanner.next();

                    scanner.nextLine();
```

```
                    System.out.println("ENTER FULLNAME");
                    fullname=scanner.nextLine();

                    System.out.println("ENTER ADDRESS");
                    address=scanner.nextLine();

                    System.out.println("ENTER SALARY");
                    salary=scanner.nextInt();

                    EmployeeDTO employeeDTO = new
EmployeeDTO(username, password, fullname, address,
salary);

                    crud.insert(employeeDTO);
                    break;

                case READ_DATA:

    System.out.println("******************READ
DATA*********************");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    System.out.println("ENTER PASSWORD");
                    password=scanner.next();

                    crud.read(username, password);
                    break;

                case UPDATE_DATA:
    System.out.println("**********************UPDATE
DATA*****************");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    System.out.println("ENTER SALARY");
                    salary=scanner.nextInt();

                    crud.update(username, salary);
                    break;

                case DELETE_DATA:
```

```
                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    crud.delete(username);

    System.out.println("**********************DELETE
DATA*****************");
                    break;

                case EXIT:
                    System.out.println("EXIT");
                    scanner.close();
                    return;

                }//switch
            }//while
        }//doStart
}

package tech.codehunt.crud;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import tech.codehunt.dto.EmployeeDTO;

public class Crud {

    public Crud() {
        createTable();

    }

    public void createTable() { // SN USER PASS
FULLNAME ADDRESS SALARY

        Connection connection = null;
        Statement statement = null;
        try {

            // WATER LEAK? -> PANI WASTE
            // MEMORY LEAK? -> NO USE

            connection =
```

```
ConnectionFactory.getConnection();// CONNECTION MANAGE
            String sql = "CREATE TABLE IF NOT EXISTS
employee(sn INT AUTO_INCREMENT PRIMARY KEY," +
                    "USERNAME varchar(30) NOT NULL
UNIQUE," + "PASSWORD varchar(30)," + "FULLNAME
varchar(100),"
                        + "ADDRESS varchar(200)," +
"SALARY int)";
            statement = connection.createStatement();
            statement.executeUpdate(sql);
            System.out.println("TABLE CREATED");

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }

    public void insert(EmployeeDTO employeeDTO) {

        Connection connection = null;
        Statement statement = null;

        try {
             String sql = "insert into
employee(username,password,fullname,address,salary)
" + "values ('"
                        + employeeDTO.getUsername() +
"','" + employeeDTO.getPassword() + "','" +
employeeDTO.getFullname()
                        + "','" +
employeeDTO.getAddress() + "'," +
employeeDTO.getSalary() + ")";

            connection =
ConnectionFactory.getConnection();
            statement = connection.createStatement();
            statement.executeUpdate(sql);
            System.out.println("DATA INSERTED");

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

```
                ConnectionFactory.close(statement);
                ConnectionFactory.close(connection);
            }
        }

        public void read(String username, String password)
{

            Connection connection = null;
            Statement statement = null;
            ResultSet resultSet=null;

            try {

                 String sql = "select * from employee
 where username='" + username + "' and password='" +
   password + "'"; // "select * from employee where
 username='raju123' and password=' anything 'OR' // 1
                        '=' 1 '";

                connection =
 ConnectionFactory.getConnection();
                statement = connection.createStatement();
                 resultSet = statement.executeQuery(sql);
 // NO DATA
                if (resultSet.next()) {
                    int getSn = resultSet.getInt("sn");
                     String getUsername =
 resultSet.getString("username");
                     String getPassword =
 resultSet.getString("password");
                     String getFullname =
 resultSet.getString("fullname");
                     String getAddress =
 resultSet.getString("address");
                     int getSalary =
 resultSet.getInt("salary");

                    System.out.println("SN: " + getSn);
                     System.out.println("USERNAME: " +
 getUsername);
                     System.out.println("PASSWORD: " +
 getPassword);
                     System.out.println("FULLNAME: " +
 getFullname);
```

```
                    System.out.println("ADDRESS: " +
getAddress);
                    System.out.println("SALARY: " +
getSalary);
                } else {
                    System.out.println("USER NOT FOUND");
                }

            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                ConnectionFactory.close(resultSet);
                ConnectionFactory.close(statement);
                ConnectionFactory.close(connection);
            }
        }

        public void update(String username, int salary) {

            Connection connection = null;
            Statement statement = null;

            try {
                 String sql = "update employee set
salary=" + salary + " where username='" + username +
"'";
                connection =
ConnectionFactory.getConnection();
                statement =
                connection.createStatement(); int
                executeUpdate =
statement.executeUpdate(sql); // NO DATA
                if (executeUpdate > 0) {
                    System.out.println("DATA UPDATED");
                } else {
                    System.out.println("USER NOT FOUND");
                }

            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                ConnectionFactory.close(statement);
                ConnectionFactory.close(connection);
            }
```

```
    }

    public void delete(String username) {

        Connection connection = null;
        Statement statement = null;

        try {

            String sql = "delete from employee where
username='" + username + "'";
            connection =
ConnectionFactory.getConnection();
            statement =
            connection.createStatement(); int
            executeUpdate =
statement.executeUpdate(sql); // NO DATA
            if (executeUpdate > 0) {
                System.out.println("DATA DELETED");
            } else {
                System.out.println("USER NOT FOUND");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }
}

package tech.codehunt.dto;
public class EmployeeDTO {

    private String username;
    private String password;
    private String fullname;
    private String address;
    private int salary;

    public EmployeeDTO() {
        // TODO Auto-generated constructor stub
    }
```

```
    public EmployeeDTO(String username, String
password, String fullname, String address, int salary)
{
        super();
        this.username = username;
        this.password = password;
        this.fullname = fullname;
        this.address = address;
        this.salary = salary;
    }

    public String getUsername() {
        return username;
    }

     public void setUsername(String username)
           { this.username = username;
    }

    public String getPassword() {
        return password;
    }
     public void setPassword(String password)
           { this.password = password;
    }

    public String getFullname() {
        return fullname;
    }

     public void setFullname(String fullname)
           { this.fullname = fullname;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
            this.address = address;
    }

    public int getSalary() {
        return salary;
```

```
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

}

package tech.codehunt.crud;
import java.sql.Connection;
import java.sql.DriverManager;
public class ConnectionFactory {

    private static final String
DB_URL="jdbc:mysql://localhost:3306/maybatch";
    private static final String
    DB_USER="root"; private static final
    String DB_PASSWORD="";

     public static Connection getConnection()

          { Connection connection=null;

        try {

    connection=DriverManager.getConnection(DB_URL,DB_
U SER,DB_PASSWORD);
        } catch (Exception e) {
            e.printStackTrace();
        }

        return connection;

    }

    public static void close(AutoCloseable resource) {
        try {
            if(resource!=null) {
                resource.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

CODE EXPLANATION

NUMBER-1: Why is it important to close Connection, Statement, and ResultSet in JDBC?

It's important to close these JDBC resources for the following reasons:

- **Prevent Resource Leakage – These objects consume system and database resources. Not closing them can lead to memory leaks.**
- **Avoid Connection Pool Exhaustion – If connections aren't closed, they're not returned to the pool, causing "Too many connections" errors.**
- **Release Database Locks – Open statements or result sets can hold database locks, blocking other operations.**
- **Improve Performance and Stability – Promptly closing resources ensures smoother application behaviour and avoids crashes.**

WHAT IS MEMORY LEAK?

In the context of JDBC:

- **When you create objects like Connection, Statement, or ResultSet, memory is allocated for them. If you don't close them, Java thinks you're still using them, so it won't clean them up — even if you're done with them.**

Over time:

- **These unused objects accumulate in memory.**
- **Memory usage keeps increasing.**
- **Eventually, your app may slow down or crash due to OutOfMemoryError.**

✅ So where is the memory leak?

- **The memory leak is in the part where JDBC objects are no longer needed but never closed — and thus never cleaned up by the JVM**

or OS.

- **Water Leak Analogy:** Imagine you have a water tank with limited water. You use a pipe (like a JDBC connection) to draw water. But after using it, you forget to turn off the tap — and water keeps leaking. Even though you're not using the water, it keeps draining.

NOTE:

Closing the Connection will automatically close all associated Statement and ResultSet objects.

So technically, it won't cause a memory leak, and no exception will be thrown.

BUT…

Why you still should close the Statement or ResultSet explicitly:

- **Best Practice / Readability:**
- **It's clear and intentional.**
- **Prevents bugs if someone reuses the connection later in the code.**

NUMBER-2: What is SQL Injection?

SQL Injection is a code injection technique that attackers use to exploit vulnerabilities in an application's interaction with a database. It occurs when malicious SQL code is inserted into an input field, which is then executed by the application's database. This can allow attackers to:

- **Retrieve unauthorized data (e.g., passwords, personal info)**
- **Modify or delete data**
- **Execute administrative operations (e.g., shut down the DB)**
- **Bypass authentication**

In the case of the Statement interface, we can say that input data is treated as part of the query — and that's exactly why SQL injection becomes possible.

✅ **What Happens in Statement**

When using Statement, user input is concatenated directly into the SQL string:

- **String query = "SELECT * FROM users WHERE username = '" + userInput + "'";**

If userInput is something like:

- ' OR '1'='1

Then the final SQL query becomes:

- SELECT * FROM users WHERE username = '' OR '1'='1'

In this case, the input is no longer just data — it's actually being treated as part of the SQL query. That's dangerous.

That's Why You Should Use Statement Only for Static queries (Queries that don't involve user input Ex. DDL)

NUMBER-3: Why Statement Interface?

Why Can't We Directly Execute SQL on Connection?

- The Connection interface in JDBC is designed to manage the connection to the database itself — not to execute SQL commands directly. It serves as the link between your Java program and the database, handling transaction management, connection pooling, and other aspects of the connection lifecycle.

Purpose of the Statement Interface

- The Statement interface is specifically responsible for executing STATIC SQL queries or updates and retrieving the results. It acts as a tool or interface that uses the underlying database connection to send SQL commands and handle the results.

NUMBER-4: execute() vs. executeUpdate() vs. executeQuery()

1. execute()

Purpose:

- Executes any SQL query (could be a SELECT, INSERT, UPDATE, DELETE, etc.) and returns a boolean indicating whether the query produced a ResultSet.

Return Type: boolean

- true: If the SQL query produces a ResultSet (usually SELECT). - false:

If the SQL query does not produce a ResultSet (e.g., INSERT, UPDATE, DELETE).

Use Case:

- When you don't know in advance if the SQL will return a result set or not. It's a "catch-all" method that can be used for any SQL command.

```
String sql = "SELECT * FROM users";
boolean isResultSet = stmt.execute(sql); //
returns true for SELECT queries
if (isResultSet) {
 ResultSet rs = stmt.getResultSet();
 // Handle result set here
}
```

2. executeUpdate()

Purpose:

- Executes SQL commands that modify the database (e.g., INSERT, UPDATE, DELETE, CREATE, DROP). It does not return a ResultSet. Instead, it returns an integer representing the number of rows affected by the SQL query.

Return Type:

- int (the number of rows affected)

Use Case:

- When you're executing SQL that modifies the database (but doesn't return data, like a SELECT query would). It's specifically designed for Data Manipulation Language (DML) commands.

```
String sql = "UPDATE users SET password =
'newpassword' WHERE username = 'john_doe'"; int
rowsAffected = stmt.executeUpdate(sql); //
Returns number of rows affected
System.out.println("Rows updated: " +
rowsAffected);
```

3. executeQuery()

Purpose:

- Executes SQL queries that return a ResultSet (i.e., SELECT statements). It is specifically designed for querying the database.

Return Type:

- ResultSet (the data returned by the query)

Use Case:

- When you know your SQL statement is a SELECT query that will return data, and you want to retrieve that data in a ResultSet.

```
String sql = "SELECT username, email FROM users";
ResultSet rs = stmt.executeQuery(sql); // Executes the query and returns a ResultSet
while (rs.next()) {
 System.out.println("Username: " + rs.getString("username")); }
```

NUMBER-5: Working of ResultSet

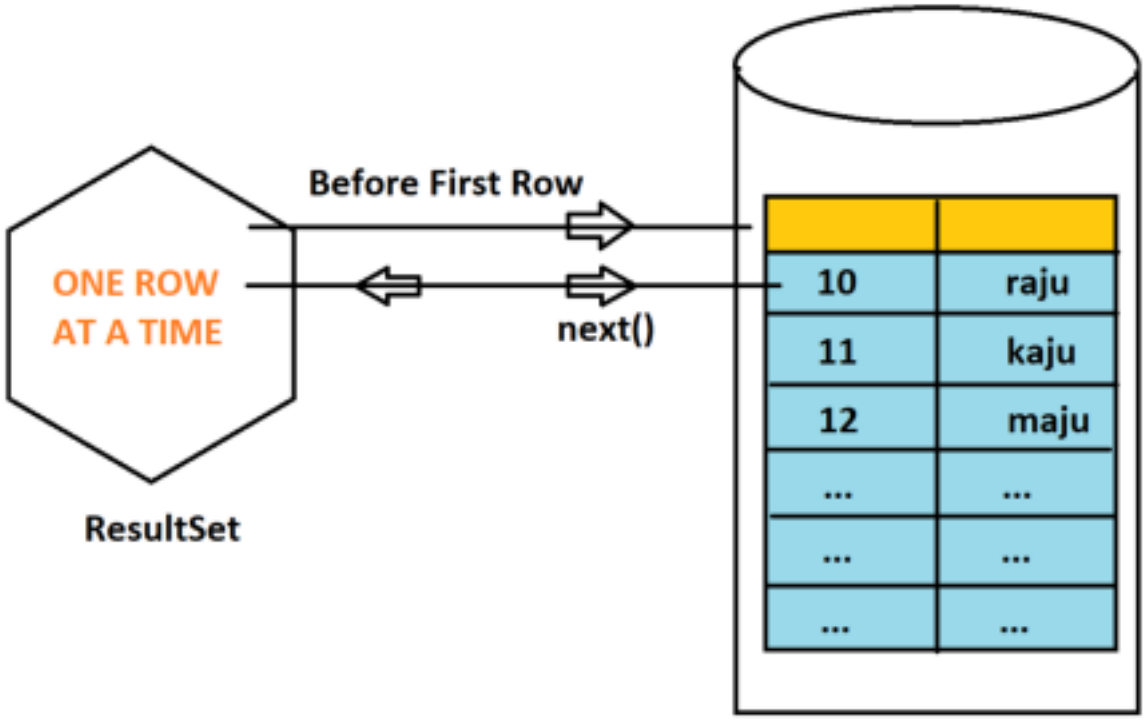
What is ResultSet?

A ResultSet in JDBC is an object that holds the data retrieved from a database after executing a query. Think of it as a table-like structure that contains rows and columns of data from your query.

Working:

In JDBC, the ResultSet does not fetch all the data from the database initially when the query is executed. Instead, it uses a cursor-based approach, where the data is retrieved one row at a time as you navigate through the result set. When the query is executed, the ResultSet is created, but the cursor is positioned before the first row. Calling the rs.next() method moves the cursor to the next row and retrieves the data for that row. This means that the entire result set is not loaded into memory at once, which helps optimize performance, especially when dealing with large datasets. The data is fetched only when needed, minimizing memory usage and allowing for more efficient processing of

the results.



So before calling rs.next():

The ResultSet does not point to any actual row yet.

It holds:

- Metadata about the result (column names, data types, number of columns, etc.).
- A cursor that is set before the first row.
- A connection to the actual query result on the database server.

DAY 4:

Up to this point, we've explored the Statement interface in JDBC, understood its usage, and identified its major drawback—it is not secure for handling user input, as it's vulnerable to SQL injection. Now, it's time to move ahead and learn about the PreparedStatement interface, which is specifically designed for secure and dynamic data insertion.

PreparedStatement:

PreparedStatement is a sub-interface of Statement in JDBC that is used to execute parameterized SQL queries.

Precompiled Query:

- **The SQL query is compiled once and can be reused multiple times with different parameters.**

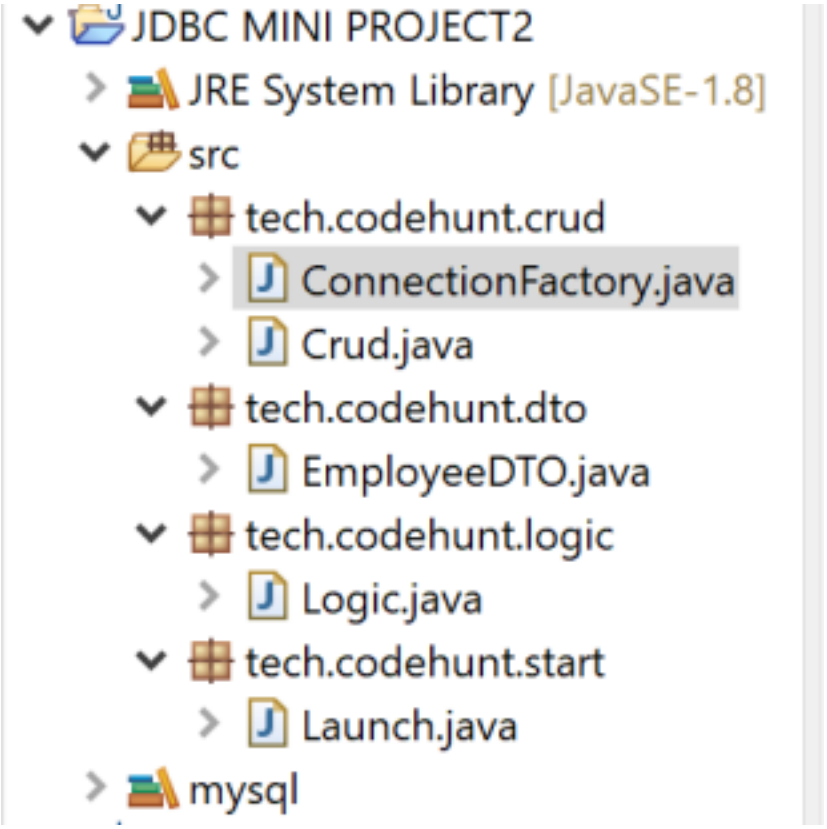
Supports Dynamic Input:

- **Uses ? as placeholders for values, which can be set at runtime.**

Protects Against SQL Injection:

- **JDBC automatically escapes input data, making it injection safe.**

=========PROGRAM: PREPARED STATEMENT=========




```java
package tech.codehunt.start;
import tech.codehunt.logic.Logic;
public class Launch {
    public static void main(String[] args) {

        Logic logic = new Logic();
        logic.doStart();

    }
}
```

```java
package tech.codehunt.logic;
import java.util.Scanner;
import tech.codehunt.crud.Crud;
import tech.codehunt.dto.EmployeeDTO;
public class Logic {

    private final int INSERT_DATA=1;
    private final int READ_DATA=2;
    private final int UPDATE_DATA=3;
    private final int DELETE_DATA=4;
    private final int TEST_DATA=5;
    private final int EXIT=6;
    private static final int MAX_ATTEMPTS=3;
    private String username;
    private String password;
```

```
    private String fullname;
    private String address;
    private int salary;
    private Crud crud;

    public Logic() {
        crud=new Crud();
    }


    public void doStart() {

        Scanner scanner = new Scanner(System.in);
        int attempt=0;

        while (true) {


    System.out.println("\n=========MENU=========");
            System.out.println("Press-1 : INSERT
DATA");
             System.out.println("Press-2 : READ
DATA");
             System.out.println("Press-3 : UPDATE
DATA");
             System.out.println("Press-4 : DELETE
DATA");
             System.out.println("Press-5 : TEST
DATA");
            System.out.println("Press-6 : EXIT\n");

            System.out.print("Enter Your Choice: ");
            int choice = 0;

            try {
                choice = scanner.nextInt();
            } catch (Exception e) {
                 System.out.print("Invalid Input!
Please take a number from 1 to 6.\n");
                scanner.nextLine();

                attempt++;
                if(attempt>=MAX_ATTEMPTS) {
                    System.out.print("You have
reached the limit");
```

```
                    scanner.close();
                    return;
                }
                continue;
            }

            if (choice < 1 || choice > 6) {
                 System.out.print("Invalid Input!
Please take a value from 1 to 6.\n");
                attempt++;
                if(attempt>=MAX_ATTEMPTS) {
                     System.out.print("You have
reached the limit");
                    scanner.close();
                    return;
                }
            }

            switch (choice) {
            case INSERT_DATA:

    System.out.println("************INSERT
DATA***********");

                System.out.println("ENTER USERNAME");
                username=scanner.next();

                System.out.println("ENTER PASSWORD");
                password=scanner.next();

                scanner.nextLine();

                System.out.println("ENTER FULLNAME");
                fullname=scanner.nextLine();

                System.out.println("ENTER ADDRESS");
                address=scanner.nextLine();

                System.out.println("ENTER SALARY");
                salary=scanner.nextInt();

                EmployeeDTO employeeDTO = new
EmployeeDTO(username, password, fullname, address,
salary);
```

```
                    crud.insert(employeeDTO);
                    break;

                case READ_DATA:

    System.out.println("******************READ
DATA********************");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    System.out.println("ENTER PASSWORD");
                    password=scanner.next();

                    crud.read(username, password);
                    break;

                case UPDATE_DATA:
    System.out.println("**********************UPDATE
DATA******************");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    System.out.println("ENTER SALARY");
                    salary=scanner.nextInt();

                    crud.update(username, salary);
                    break;

                case DELETE_DATA:


    System.out.println("**********************DELETE
DATA******************");

                    System.out.println("ENTER USERNAME");
                    username=scanner.next();

                    crud.delete(username);
                    break;

 case TEST_DATA:
```

```
    System.out.println("**********************TEST
DATA*****************");

                    String sql1="select * from employee";
                     String sql2="insert into
 employee(username,password,fullname,address,salary)
                              " +
"values('test','test123','testname','testadd',2000)";

                    System.out.println("P1->SELECT\nP2-
>NON SELECT");
                    choice=scanner.nextInt();
                    if(choice==1) {
                         crud.test(sql1);
                    }else {
                        crud.test(sql2);
                    }

                    break;

                case EXIT:
                    System.out.println("EXIT");
                    scanner.close();
                    return;

                }//switch
            }//while
    }//doStart
}
```

```
package tech.codehunt.crud;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import tech.codehunt.dto.EmployeeDTO;

public class Crud {

    public Crud() {
        createTable();
    }
```

```
    public void createTable() { // SN USER PASS
FULLNAME ADDRESS SALARY

        Connection connection = null;
        Statement statement = null;
        try {

            // WATER LEAK? -> PANI WASTE
            // MEMORY LEAK? -> NO USE

            connection =
ConnectionFactory.getConnection();// CONNECTION MANAGE
 String sql = "CREATE TABLE IF NOT EXISTS  employee(sn
           INT AUTO_INCREMENT PRIMARY KEY,"
                       + "USERNAME varchar(30) NOT NULL
UNIQUE," + "PASSWORD varchar(30)," + "FULLNAME
varchar(100),"
                       + "ADDRESS varchar(200)," +
"SALARY int)";
            statement = connection.createStatement();
            statement.executeUpdate(sql);
            System.out.println("TABLE CREATED");

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            ConnectionFactory.close(statement);
            ConnectionFactory.close(connection);
        }
    }

    public void insert(EmployeeDTO employeeDTO) {

        Connection connection = null;
        PreparedStatement preparedStatement = null;

        try {
             String sql = "insert into
employee(username,password,fullname,address,salary)
values(?,?,?,?,?)";

            connection =
ConnectionFactory.getConnection();
            preparedStatement =
```

```
connection.prepareStatement(sql); // COMPILE->DATABASE

            int count = 1;
            while (count <= 5) {

                preparedStatement.setString(1,
employeeDTO.getUsername()+count);
                preparedStatement.setString(2,
employeeDTO.getPassword());
                preparedStatement.setString(3,
employeeDTO.getFullname());
                preparedStatement.setString(4,
employeeDTO.getAddress());
                preparedStatement.setInt(5,
employeeDTO.getSalary());

                preparedStatement.executeUpdate();

                System.out.println("DATA INSERTED");
                count++;
            }

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

    ConnectionFactory.close(preparedStatement);
            ConnectionFactory.close(connection);
        }
    }

    public void read(String username, String password)
{

        Connection connection = null;
        PreparedStatement preparedStatement =
        null; ResultSet resultSet = null;

        try {

             String sql = "select * from employee
where username=? and password=?";

            connection =
ConnectionFactory.getConnection();
```

```
            preparedStatement =
connection.prepareStatement(sql);

            preparedStatement.setString(1, username);
            preparedStatement.setString(2, password);

            resultSet =
preparedStatement.executeQuery();

            if (resultSet.next()) {
                int getSn = resultSet.getInt("sn");
                 String getUsername =
resultSet.getString("username");
                 String getPassword =
resultSet.getString("password");
                 String getFullname =
resultSet.getString("fullname");
                 String getAddress =
resultSet.getString("address");
                 int getSalary =
resultSet.getInt("salary");

                System.out.println("SN: " + getSn);
                 System.out.println("USERNAME: " +
getUsername);
                 System.out.println("PASSWORD: " +
getPassword);
                 System.out.println("FULLNAME: " +
getFullname);
                 System.out.println("ADDRESS: " +
getAddress);
                 System.out.println("SALARY: " +
getSalary);
            } else {
                System.out.println("USER NOT FOUND");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            ConnectionFactory.close(resultSet);

    ConnectionFactory.close(preparedStatement);
            ConnectionFactory.close(connection);
        }
```

```
    }

    public void update(String username, int salary) {

        Connection connection = null;
        PreparedStatement preparedStatement =

        null; try {

             String sql = "update employee set
salary=? where username=?";
            connection =
ConnectionFactory.getConnection();
            preparedStatement =
connection.prepareStatement(sql);

            preparedStatement.setInt(1, salary);
            preparedStatement.setString(2, username);

            int executeUpdate =
  preparedStatement.executeUpdate(); // NO
      DATA if (executeUpdate > 0) {
                System.out.println("DATA UPDATED");
            } else {
                System.out.println("USER NOT FOUND");
            }

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

    ConnectionFactory.close(preparedStatement);
            ConnectionFactory.close(connection);
        }

    }

    public void delete(String username) {

        Connection connection = null;
        PreparedStatement preparedStatement =

        null; try {

            String sql = "delete from employee where
```

```
username=?";
                connection =
ConnectionFactory.getConnection();
                preparedStatement =
connection.prepareStatement(sql);

               preparedStatement.setString(1, username);
                int executeUpdate =
   preparedStatement.executeUpdate(); // NO
         DATA if (executeUpdate > 0) {
                     System.out.println("DATA DELETED");
                } else {
                     System.out.println("USER NOT FOUND");
                }

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

     ConnectionFactory.close(preparedStatement);
                ConnectionFactory.close(connection);
           }
     }


     public void test(String sql) { //EMPLOYEE: READ
/ INSERT UPDATE DELETE

           Connection connection = null;
           PreparedStatement preparedStatement =

           null; try {


                connection =
ConnectionFactory.getConnection();
                preparedStatement =
connection.prepareStatement(sql);

                boolean execute =
preparedStatement.execute();

                if(execute) {
                     ResultSet resultSet =
preparedStatement.getResultSet();
```

```
                if (resultSet.next()) {
                    int getSn =
resultSet.getInt("sn");
                    String getUsername =
resultSet.getString("username");
                    String getPassword =
resultSet.getString("password");
                    String getFullname =
resultSet.getString("fullname");
                    String getAddress =
resultSet.getString("address");
                    int getSalary =
resultSet.getInt("salary");

                    System.out.println("SN: " +
getSn);
                   System.out.println("USERNAME: "
+ getUsername);
                    System.out.println("PASSWORD: "
+ getPassword);
                    System.out.println("FULLNAME: "
+ getFullname);
                    System.out.println("ADDRESS: " +
getAddress);
                    System.out.println("SALARY: " +
getSalary);
                } else {
                    System.out.println("USER NOT
FOUND");
                }
            }
            else {
                int k =
preparedStatement.getUpdateCount();
                System.out.println(k);
            }

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

    ConnectionFactory.close(preparedStatement);
            ConnectionFactory.close(connection);
        }
    }
```

```
}
package tech.codehunt.dto;
public class EmployeeDTO {

	private String username;
	private String password;
	private String fullname;
	private String address;
	private int salary;

	public EmployeeDTO() {
		// TODO Auto-generated constructor stub
	}

	public EmployeeDTO(String username, String
password, String fullname, String address, int salary)
{
		super();
		this.username = username;
		this.password = password;
		this.fullname = fullname;
		this.address = address;
		this.salary = salary;
	}

	public String getUsername() {
		return username;
	}

	 public void setUsername(String username)
			{ this.username = username;
	}

	public String getPassword() {
		return password;
	}

	 public void setPassword(String password)
			{ this.password = password;
	}

	public String getFullname() {
		return fullname;
```

```
    }
     public void setFullname(String fullname)
           { this.fullname = fullname;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public int getSalary() {
        return salary;
    }

    public void setSalary(int salary) {
        this.salary = salary;
    }

}

package tech.codehunt.crud;
import java.sql.Connection;
import java.sql.DriverManager;
public class ConnectionFactory {

    private static final String
DB_URL="jdbc:mysql://localhost:3306/maybatch";
    private static final String
    DB_USER="root"; private static final
    String DB_PASSWORD="";

     public static Connection getConnection()

          { Connection connection=null;

        try {

    connection=DriverManager.getConnection(DB_URL,DB_
U SER,DB_PASSWORD);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
            return connection;

        }

        public static void close(AutoCloseable resource) {
            try {
                if(resource!=null) {
                    resource.close();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

==========

==========

Number1: Why PreparedStatement is Safe from SQL Injection ?

SQL Injection happens when malicious users insert or "inject" SQL code into your query by exploiting input fields. This usually works when you dynamically build queries using string concatenation—like with the Statement interface.

✅ **With PreparedStatement, input is treated as data, not code.**

How it works internally:

Query is Precompiled:

- When you write a PreparedStatement like this:
- String sql = "SELECT * FROM users WHERE username = ? AND password = ?";

- `PreparedStatement ps = conn.prepareStatement(sql);`
- The SQL query with placeholders (?) is sent to the database server in advance and compiled. At this stage, the DB knows the structure of the query, but not the actual data.

Parameters are Bound Safely: (You then set the parameters like this)

- `ps.setString(1, userInputUsername);`
- `ps.setString(2, userInputPassword);`

- These inputs are not merged into the SQL string. Instead, they're sent separately to the database driver, which treats them strictly as values, not part of the SQL syntax.

Special Characters are Escaped: (Even if a user enters something like)

- `'OR'1'='1`

- It is treated as a plain string literal (data), not part of the SQL code. The database doesn't interpret it as a logical expression because the query is already compiled with fixed placeholders.

DAY 5:

CallableStatement

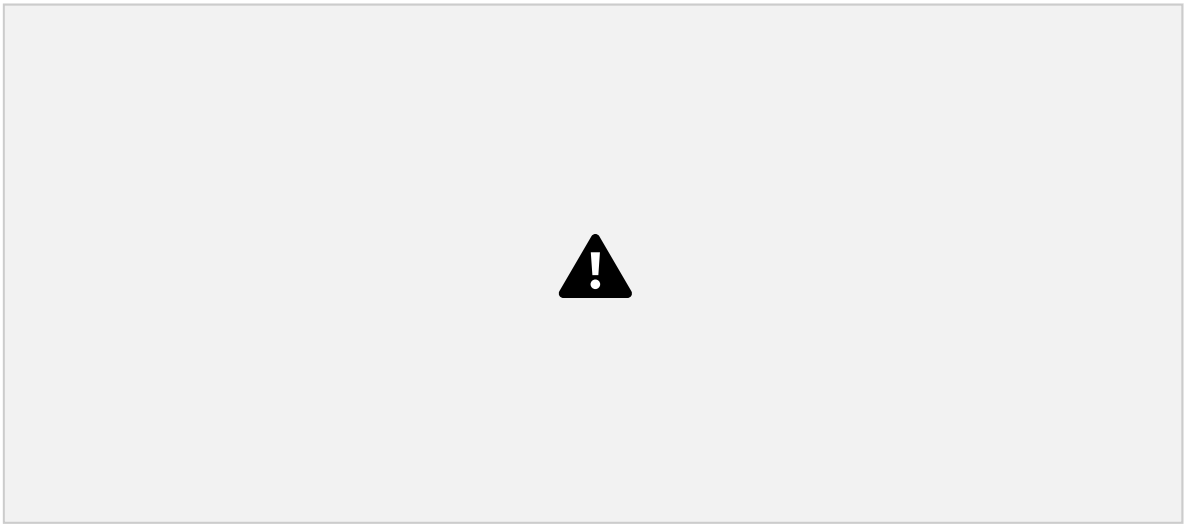
What is CallableStatement in JDBC?

- A CallableStatement is a special type of JDBC statement used to execute stored procedures in a relational database. It is part of the java.sql package and is an extension(child) of PreparedStatement.
- Stored procedures are precompiled SQL code stored in the database that can accept input parameters, return output parameters, and even return result sets.

Why Use CallableStatement?

- Encapsulate complex logic: Shift business logic from the application layer to the database.

- **Reusability: Stored procedures can be reused by multiple applications.**
- **Performance: Stored procedures are compiled once and stored in the DBMS.**
- **Security: Users can be granted permission to execute procedures without direct access to underlying tables.**
- **Maintainability: Changing logic in one stored procedure is easier than updating application logic in multiple places.**



==========PROGRAM: CALLABLE STATEMENT==========

```
package com.mainapp;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
public class Launch {
    public static void main(String[] args) {

        try
```

```
        {
            String
url="jdbc:mysql://localhost:3306/maybatch"
        ; String username="root";
            String password="";

            Connection con =
DriverManager.getConnection(url,username,password);
             CallableStatement callableStatement =
con.prepareCall("{CALL xyz(?,?)}");

            callableStatement.setString(1, "j%");
            callableStatement.setInt(2, 1000);

            ResultSet rs =
callableStatement.executeQuery();
            while(rs.next()) {

    System.out.println(rs.getString("fullname"));

    System.out.println(rs.getInt("salary"));
            }

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

==========

==========

QUESTION: **Both PreparedStatement and CallableStatement help with performance by precompiling SQL queries. If I call a method that creates a PreparedStatement or a CallableStatement with the same SQL or stored procedure every time, will the SQL or procedure be recompiled each time? What's the difference in terms of compilation and performance when using PreparedStatement vs CallableStatement for repeated executions?**

IN CASE OF PREPARED STATEMENT

```
▪ Connection
  connection=ConnectionFactory.getConnection(); ▪
String SQL = "SELECT * FROM employees WHERE id  =
?";
▪ PreparedStatement ps =
  con.prepareStatement(SQL);
```

and this code is inside a method, and you call that method twice, then: The SQL will be compiled (parsed + planned) by the database each time you call the method.

Why?

- **PreparedStatement is precompiled, but only per JDBC connection and per statement object.**
- **If you create a new PreparedStatement each time, the DB sees it as a new statement and recompiles it.**
- **So yes, compilation happens each time you call the method if you're not reusing the PreparedStatement.**

How to Avoid Recompilation?

- **If you want to avoid SQL recompilation, you should: Reuse the same PreparedStatement object:**

```
PreparedStatement ps =
connection.prepareStatement("SELECT
..."); for (int i = 0; i < 2; i++) {
 ps.setInt(1, id);
 ps.executeQuery();
}
```

IN CASE OF CALLABLE STATEMENT

• You have a stored procedure in DB: getEmployee(IN empId INT) • You define a String SQL = "{call getEmployee(?)}"; • You call a method like this:

```
public void callProcedure(int id) {
 String SQL = "{call getEmployee(?)}";
```

```
CallableStatement cs =
connection.prepareCall(SQL);
 cs.setInt(1, id);
 cs.execute();
}
```

You call callProcedure(101) and then callProcedure(102) then What Happens Internally?

• Stored procedures are precompiled and stored in the database. • The database compiles it once when you create it. • Every call to it just executes the already-compiled logic. • When you're calling connection.prepareCall(SQL) every time. • That creates a new statement object and sends it to the DB. • JDBC driver parses the call and prepares a statement each time.

But this preparation is very lightweight **because the actual SQL logic is already compiled in the DB.**

JDBC TRANSCTION MANAGEMENT

What is a Transaction?

- **A transaction is a group of one or more SQL statements that are executed as a single unit of work. It ensures data integrity and follows the principle of:**

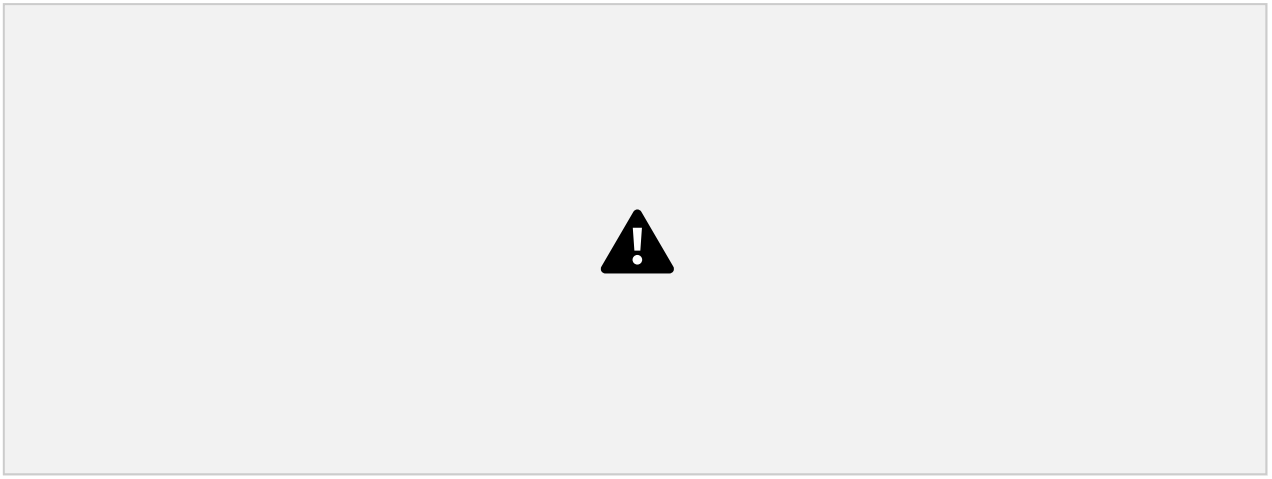
Atomicity

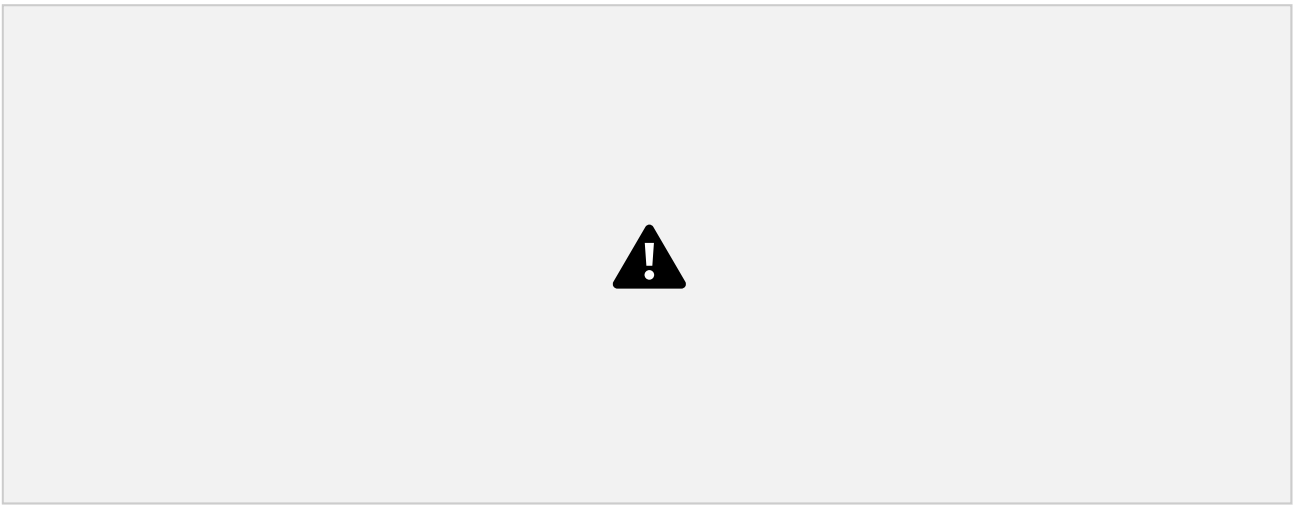
- **Atomicity means all operations in a transaction either complete successfully together, or none of them take effect at all. If one part fails, the entire transaction is rolled back, leaving the database unchanged — as if nothing happened.**

What is Transaction Management in JDBC?

- **Transaction management in JDBC is the process of controlling a set of SQL operations so that they either all succeed together or fail together, ensuring the consistency and integrity of the database.**

It is used to achieve atomicity in database operations, where a group of SQL statements must be treated as a single unit of work.





=======PROGRAM: TRANSACTION MANAGEMENT=======

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;
public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con=null;
        try
        {
            String
url="jdbc:mysql://localhost:3306/maybatch";
            String username="root";
            String password="";
```

```
            con =
DriverManager.getConnection(url,username,password);
            con.setAutoCommit(false);

             String sql="update employee set salary=?
where username='ttt'";
             PreparedStatement preparedStatement =
con.prepareStatement(sql);

             System.out.println("ENTER NEW
SALARY");//1000
             preparedStatement.setInt(1,
scanner.nextInt() );

            int executeUpdate1 =
preparedStatement.executeUpdate();
            System.out.println(executeUpdate1);

            Thread.sleep(10000);

             System.out.println("ENTER NEW
SALARY");//2000
             preparedStatement.setInt(1,
scanner.nextInt() );

            int executeUpdate2 =
preparedStatement.executeUpdate();
            System.out.println(executeUpdate2);

            Thread.sleep(10000);

             System.out.println("ENTER NEW
SALARY");//3000
             preparedStatement.setInt(1,
scanner.nextInt() );

            int executeUpdate3 =
preparedStatement.executeUpdate();
            System.out.println(executeUpdate3);

            con.commit();

        }
        catch (Exception e) {
```

```
                try {
                    con.rollback();
                } catch (SQLException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
                e.printStackTrace();
            }
        }
}
```

==========

==========

= *DAY 6:*

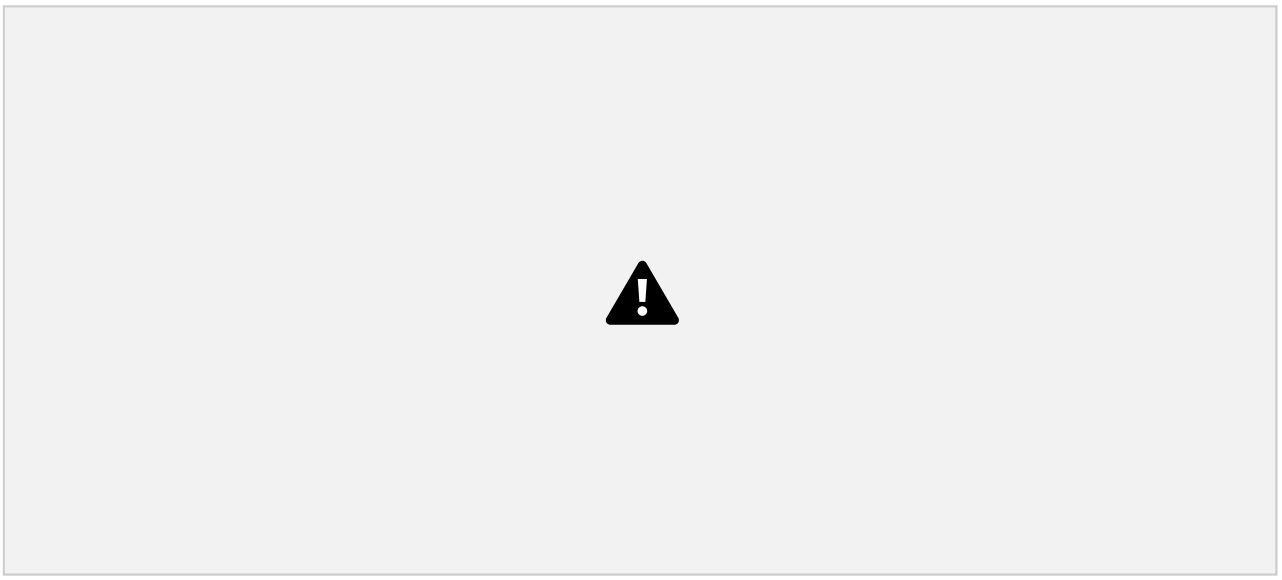
Batch Processing

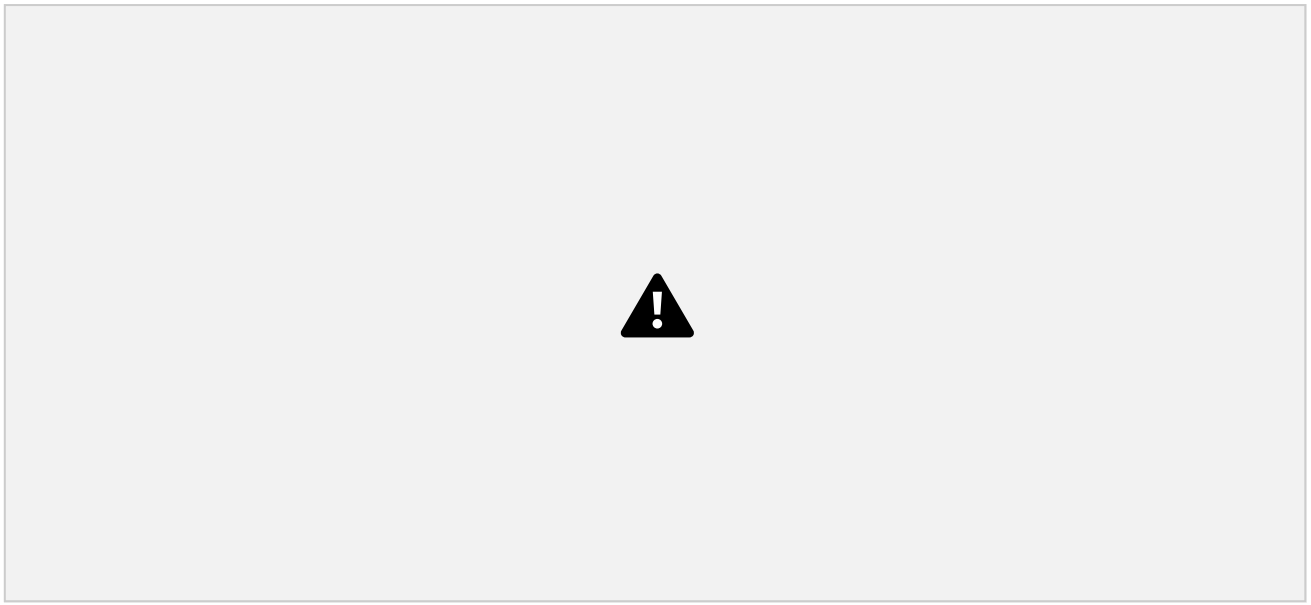
What is Batch Processing in JDBC?

Batch Processing allows you to group multiple SQL statements and send them to the database in one go, instead of executing them one by one. This reduces:

- **Network calls**
- **Execution time**

Main Goal: Improve performance when running multiple SQL statements (INSERT, UPDATE, or DELETE).





Common Use Case:

Imagine inserting 1000 records:

- **Without batch: 1000 SQL calls (slow)**
- **With batch: 1 SQL call with 1000 statements (faster)**

Steps to Perform Batch Processing:

1. Create Connection

2. Disable Auto-Commit (optional but recommended) 3. Create Statement or PreparedStatement

4. Add SQL statements to batch (MULTIPLE)

5. Execute the batch

6. Commit the transaction

7. Handle exceptions & close resources

==========PROGRAM: BATCH PROCESSING========== package com.mainapp;

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;

public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con = null;
        try {
            String url =
"jdbc:mysql://localhost:3306/maybatch";
```

```
            String username = "root";
            String password = "";

             con = DriverManager.getConnection(url,
username, password);
            con.setAutoCommit(false);
            System.out.println(con);

             String sql = "insert into
employee(username,password,fullname,address,salary)
values(?,?,?,?,?)";
             PreparedStatement preparedStatement =
con.prepareStatement(sql);

            while (true) {

                System.out.println("ENTER USERNAME");
                String user = scanner.next();

                System.out.println("ENTER PASSWORD");
                String pass = scanner.next();

                scanner.nextLine();

                System.out.println("ENTER FULLNAME");
                String fullname = scanner.nextLine();

                System.out.println("ENTER ADDRESS");
                String address = scanner.nextLine();

                System.out.println("ENTER SALARY");
                int salary = scanner.nextInt();

                preparedStatement.setString(1, user);
                preparedStatement.setString(2, pass);
                 preparedStatement.setString(3,
fullname);
                 preparedStatement.setString(4,
address);
                preparedStatement.setInt(5, salary);

                preparedStatement.addBatch();

                 System.out.println("DO U WANT 2
INSERT MORE:(Y)");
```

```
                    String
choice=scanner.next().trim().toUpperCase()
                    ; if(!choice.equals("Y"))
                    {
                         int[] i =
preparedStatement.executeBatch();
 //Arrays.stream(i).forEach(n-
>System.out.println(n));
                         System.out.println("ROWS
INSERTED:-"+i.length);
                        con.commit();
                        break;
                    }
                }
            } catch (Exception e) {
                try {
                    con.rollback();
                } catch (SQLException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
                e.printStackTrace();
            }
        }
}
```

==========

==========

How many SQL statements can we put in a single JDBC batch?

- There is no fixed limit defined by JDBC itself, but there are practical limits depending on Database Configuration.
- In MySQL 8(XAMPP), the number of SQL statements you can include in a single JDBC batch is not limited by the number of statements, but by the total size of the batch, which is controlled

by the max_allowed_packet setting on the server. By default, max_allowed_packet is set to 1MB, but it can be increased up to 1GB if needed. This means you can add as many statements to a batch as you want, as long as the combined size of all SQL statements in the batch doesn't exceed the max_allowed_packet limit. For example, if each statement is about 10KB and the limit is 1MB, you could batch roughly 100

- To check the current max_allowed_packet value in MySQL, you can run the SQL command:

```
SHOW VARIABLES LIKE 'max_allowed_packet';
```

- To change (in my.ini file):

```
max_allowed_packet=2M
```

When NOT to Use Batch Processing in JDBC?

❌ 1. When Each Query Depends on the Previous Result

If you need to:

- Get the result of one query before running the next
- Make decisions based on intermediate results

Example:

```
ResultSet rs = stmt.executeQuery("SELECT stock
FROM products WHERE id = ?");
if (rs.getInt("stock") < 5) {
 stmt.executeUpdate("ORDER MORE STOCK"); }
```

❌ 2. When Using Non-DML Queries (e.g., SELECT, DDL)

Batching is mainly for:

- INSERT
- UPDATE
- DELETE

You cannot batch SELECT or CREATE TABLE statements.

❌ 3. When Batch Size is Too Small to Matter

If you're only doing:

- 2 or 3 queries
- Or only one-time operations

➡ Using a batch adds unnecessary code complexity with no real performance benefit.

DAY 7:

MetaData

Metadata means "data about data".

In JDBC, metadata helps us get information about:

- **The database (like DB name, version, tables)**
- **The ResultSet (columns, types, etc.)**

JDBC Metadata is a feature in Java that allows you to get information about the database, tables, and columns at runtime, without hardcoding anything

✅ JDBC Metadata Summary with Examples

Metadata Type Purpose

Interface What You Can Get (Examples)

DatabaseMetaData Info about the database, tables, and JDBC driver java.sql.DatabaseMetaData ✅ Database name (e.g., MySQL, Oracle)

✅ Database version (e.g., 8.0.29)

✅ Logged-in username (e.g., root)

✅ List of all tables in the database

✅ Table's primary keys and foreign keys

✅ Supported SQL features (like batch updates, stored procedures)
✅ JDBC driver name and version

ResultSetMetaData Info about columns in the result of a SELECT query java.sql.ResultSetMetaData ✅ Total number of columns

✅ Name of each column (e.g., "id", "name")

✅ Data type of each column (e.g., INT, VARCHAR)

✅ **Whether the column is nullable**

✅ **Whether the column is auto-increment**

✅ **Column size and precision**

========PROGRAM-1: DATABASE META DATA========

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Scanner;
public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con=null;
        try
        {
            String
url="jdbc:mysql://localhost:3306/maybatch"
        ; String username="root";
            String password="";

            con =
DriverManager.getConnection(url,username,password);

             DatabaseMetaData metaData =
con.getMetaData();//FACTORY DESIGN PATTERN

            String url2 = metaData.getURL();
            System.out.println(url2);

            String userName2 =
metaData.getUserName();
            System.out.println(userName2);

             String databaseProductName =
metaData.getDatabaseProductName();
             String databaseProductVersion =
```

```
metaData.getDatabaseProductVersion();
                int databaseMajorVersion =
metaData.getDatabaseMajorVersion();
                int databaseMinorVersion =
metaData.getDatabaseMinorVersion();

                System.out.println(databaseProductName);

        System.out.println(databaseProductVersion);
                System.out.println(databaseMajorVersion);
                System.out.println(databaseMinorVersion);
                String driverName =
metaData.getDriverName();
                String driverVersion =
metaData.getDriverVersion();
                int driverMajorVersion =
metaData.getDriverMajorVersion();
                int driverMinorVersion =
metaData.getDriverMinorVersion();

                System.out.println(driverName);
                System.out.println(driverVersion);
                System.out.println(driverMajorVersion);
                System.out.println(driverMinorVersion);

                int maxUserNameLength =
metaData.getMaxUserNameLength();
                System.out.println(maxUserNameLength);

                int maxColumnsInTable =
metaData.getMaxColumnsInTable();
                System.out.println(maxColumnsInTable);

                int maxRowSize =
metaData.getMaxRowSize();
                System.out.println(maxRowSize);


            }
            catch (Exception e) {
                try {
                    con.rollback();
                } catch (SQLException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
```

```
            }
            e.printStackTrace();
        }
    }
}
```

==========
=
==========
=

========PROGRAM-2: RESULTSET META DATA=======

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Scanner;
public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con=null;
        try
        {
            String
url="jdbc:mysql://localhost:3306/maybatch"
        ; String username="root";
            String password="";

            con =
DriverManager.getConnection(url,username,password);

            String sql="select * from employee";
            PreparedStatement ps =
con.prepareStatement(sql);
            ResultSet resultSet = ps.executeQuery();

             ResultSetMetaData metaData =
resultSet.getMetaData();
```

```
            int columnCount =
metaData.getColumnCount();
            System.out.println(columnCount);

            for(int i=1;i<=columnCount;i++) {
             String columnName =
metaData.getColumnName(i);
               System.out.println(columnName);
            }

            for(int i=1;i<=columnCount;i++) {
                    String columnName =
metaData.getColumnClassName(i);
                   System.out.println(columnName);
            }

            for(int i=1;i<=columnCount;i++) {
                 int size=
metaData.getColumnDisplaySize(i);
                   System.out.println(size);
            }

            String tableName =
metaData.getTableName(1);
            System.out.println(tableName);

            String dbname =
metaData.getCatalogName(1);
            System.out.println(dbname);

        }
        catch (Exception e) {
            try {
                con.rollback();
            } catch (SQLException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
    }
}
```

==========
==========

DAY 8:

Connection with Properties

What is a JDBC Connection with Properties?

- In JDBC, a connection using a Properties object means passing the database username, password, and other settings separately using key-value pairs instead of writing everything inside the connection URL. This makes the code cleaner, easier to manage, and more flexible for future changes. Typically, these properties are loaded from an external .properties file, which helps in keeping configuration separate from code and simplifies environment-based setups.

✅ Advantages of Using Properties in JDBC

1. Better Security

• You avoid hardcoding sensitive data (username/password) directly in the code.

• You can load credentials from encrypted or external .properties files.

2. Easier Configuration Management

• All connection settings are stored in a central place. • Easy to change properties without touching the Java code.

❌ Disadvantages of Using Properties in JDBC

1. External File Handling

• If you're loading from a .properties file, you must manage file paths properly.

• Errors like FileNotFoundException can occur if the path is incorrect.

✅ When to Use Properties Object in JDBC

- Situation :- You want to change DB config without touching code
  Reason :- Makes configuration external and flexible
- Situation :- App runs in multiple environments (dev/test/prod)
  Reason :- Load different .properties files per environment •
  Situation :- You want better separation of code and credentials
  Reason :- Avoids hardcoding sensitive info

❌ When Not to Use Properties Object

- Situation :- Just a small or demo project
  Reason :- Simpler to write everything in one line

CAN WE PUT CONFIGURATION IN A TEXT FILE (.txt) ???

Example: (config.txt)

user=root
password=pass123

YES, technically it looks just like a .properties file — and Java's Properties.load() can read it perfectly fine.

So what's the difference then?

The file extension .properties is a convention telling developers and tools:

- "This file contains configuration properties."
- .txt is a generic text file extension with no special meaning.

(.properties vs .txt)

Feature	Properties file	Text file
• Format	Key-value pairs (standardized)	Free-form text (no structure enforced)
• Readable by Java	✅ Directly with Properties.load()	❌ Must read manually and parse(IF NOT IN KEY VALUE)
• Purpose	Configuration settings	Any general-purpose content
• File extension	.properties	.txt
• Usage in Java projects	Widely used (for config)	Rare for storing credentials

Why does the extension matter?

1.Tools & Frameworks:

- Most Java tools (Maven, Spring, IDEs) expect config files to be .properties.
- Using .properties helps those tools recognize and handle the file correctly.

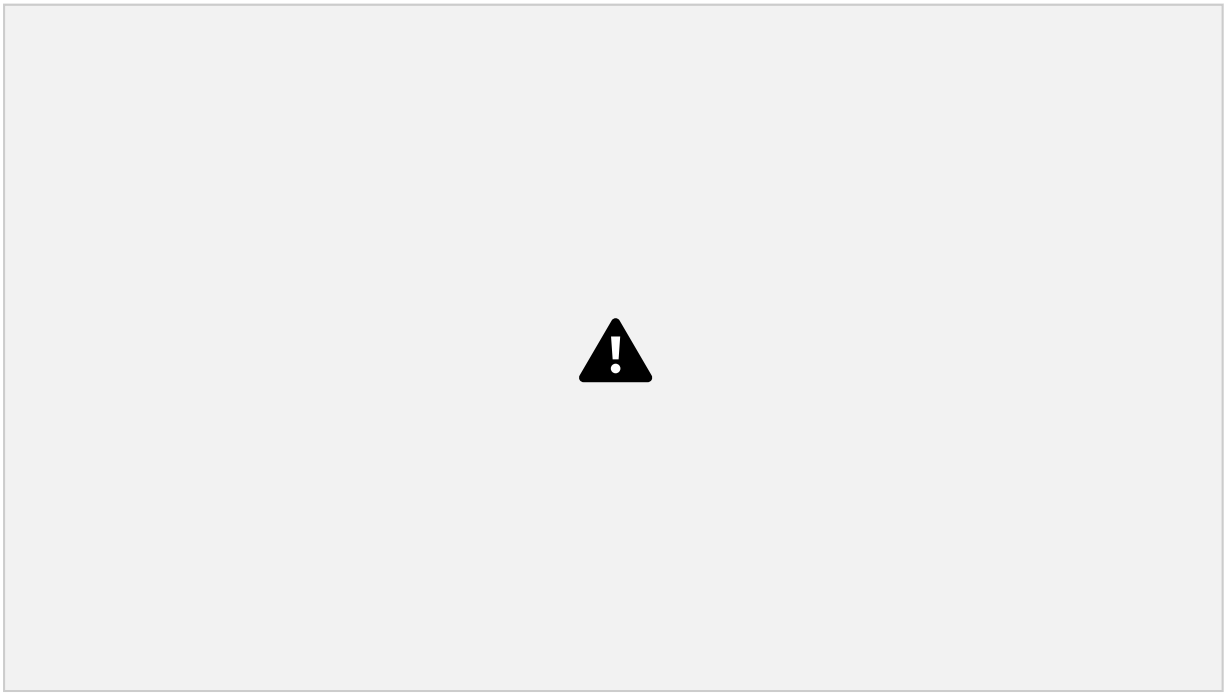
2.Readability & Maintenance:

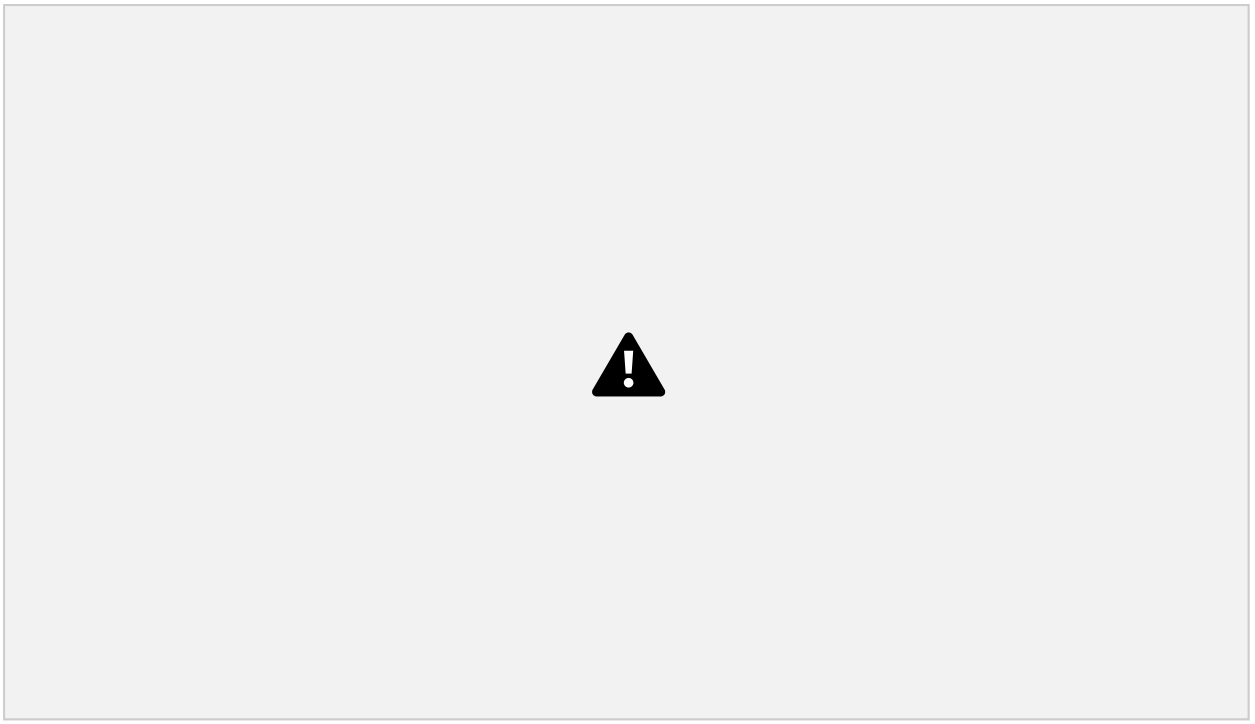
- Developers immediately know what the file is for by its extension.

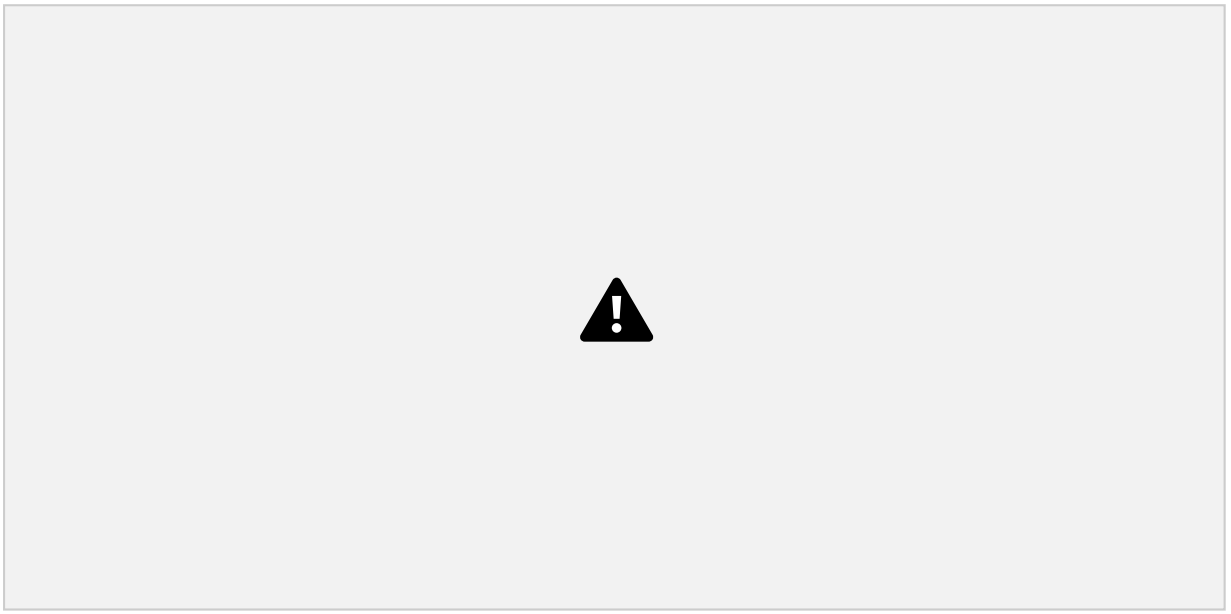
3.Best Practice & Standards:

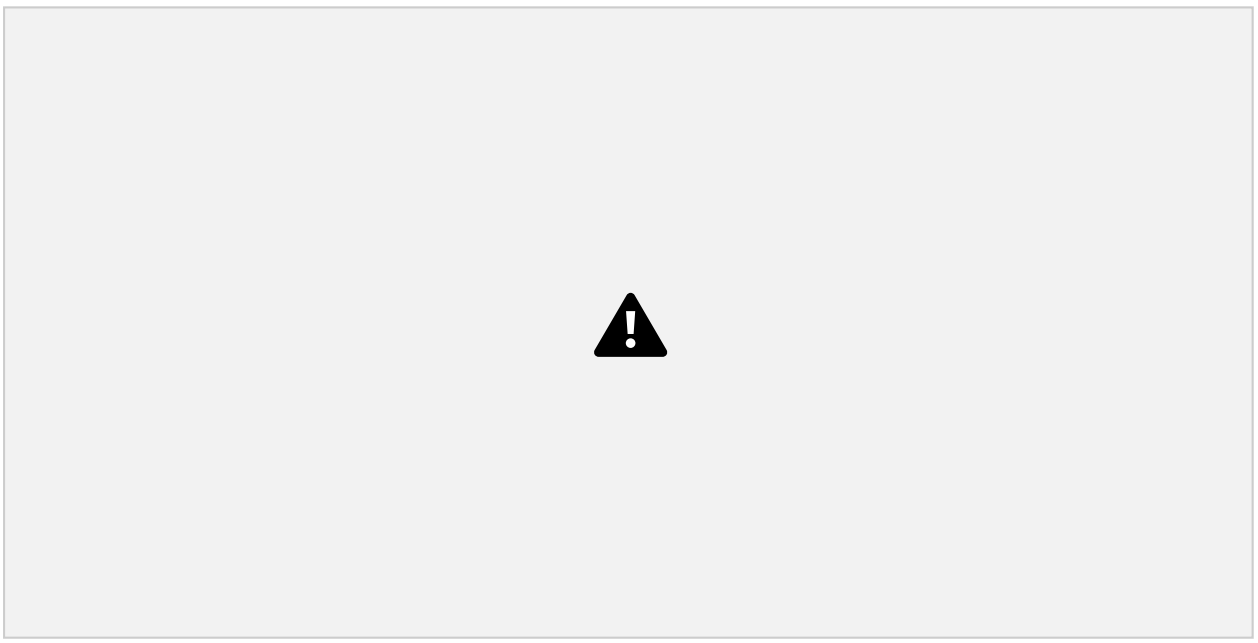
- Following conventions helps others understand and maintain your project easily.

Scenario	Works?	Recommended?
Key-value pairs in .properties file	✅ Java reads easily	✅ Yes, industry standard
Key-value pairs in .txt file	✅ Java reads if formatted	❌ No, avoid confusing use

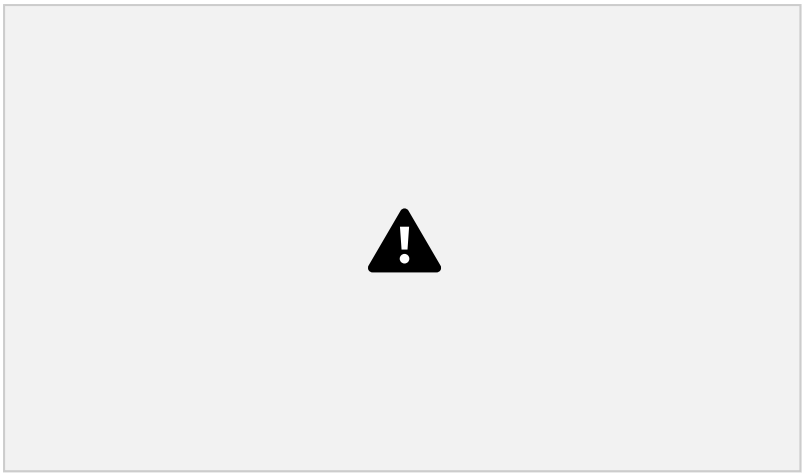


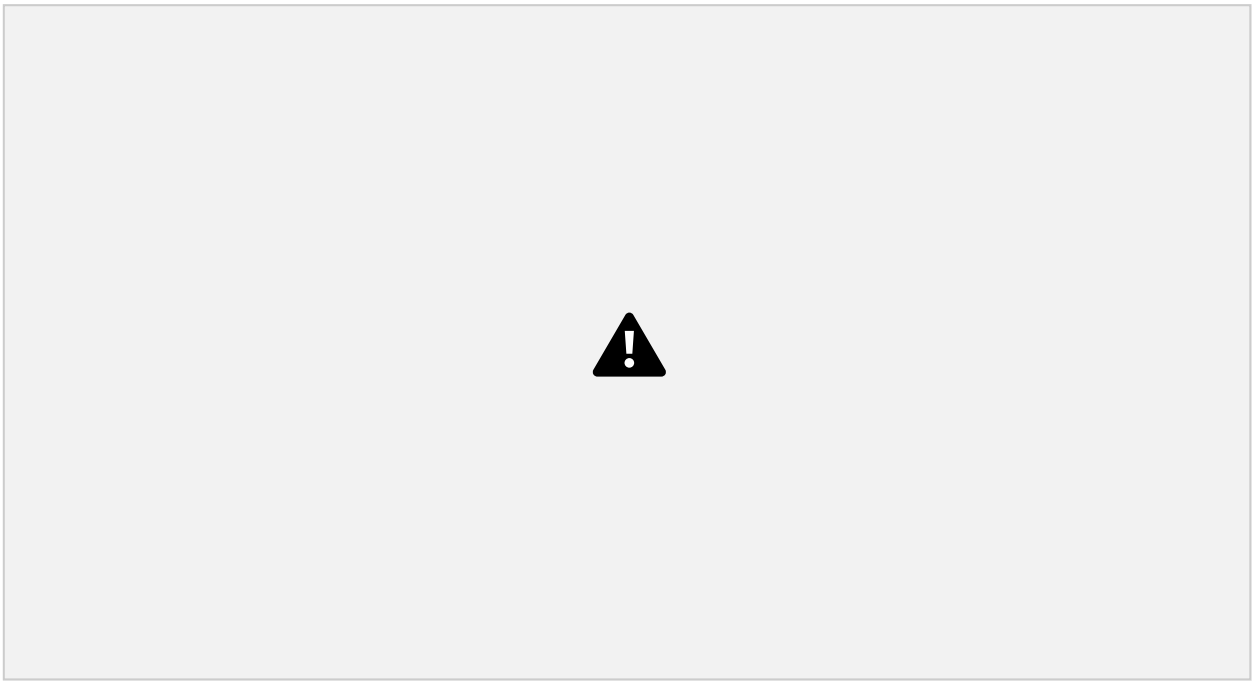






=====PROGRAM: CONNECTION WITH PROPERTIES=====





```
package com.mainapp;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class Launch {

    public static void main(String[] args) {

        try {

            while (true) {
                //FileInputStream fis = new
FileInputStream("C:\\Users\\DELL\\Desktop\\config\\my
c onfig.properties");
                FileInputStream fis = new
FileInputStream("myconfig.properties");
                Properties properties = new
Properties(); // KEY VALUE LOAD
```

```
                properties.load(fis);

                String url = (String)
properties.get("mysql.url");
                String user = (String)
properties.get("mysql.user");
                String pass = (String)
properties.get("mysql.pass");

                Connection connection =
DriverManager.getConnection(url, user, pass);
                System.out.println(connection + " ->
" + connection.getCatalog());
                Thread.sleep(2000);
            }

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

==========
==========

Connection Pooling

What is Connection Pooling?

Connection pooling is a technique used to manage and optimize database connections in applications.

Why is Connection Pooling needed?

- **Opening and closing a database connection is expensive and slow — it involves network overhead, authentication, and resource allocation. • In web apps or enterprise apps, multiple requests often need to access the database concurrently and repeatedly.**

• Creating a new connection for every request hurts performance and scalability.

What does Connection Pooling do?

• It creates a fixed number of connections upfront (a “pool”). • When your app needs a connection, it borrows one from the pool instead of creating a new one.

• After using, the connection is returned to the pool for reuse.

When is Connection Pooling Suitable?

Connection pooling is ideal for applications that:

- Are web-based or enterprise-level, where multiple users or threads frequently access the database.
- Require high performance and scalability, such as in online shopping platforms, financial applications, or REST APIs.
- Have a limited number of database connections available and need to reuse them efficiently.
- Connections are built using multi-threaded environments where simultaneous database access is common.

When Not to Use Connection Pooling?

Connection pooling may not be suitable in cases like:

- Simple desktop or command-line applications that make occasional database queries and don’t run continuously. • Testing

How Connection Pooling Works (Step by Step):

1. Initialization: When the app starts, a pool of DB connections (e.g., 5-20) is created and kept open.
2. Borrowing: When your code calls getConnection(), the pool hands over an available connection.
3. Usage: Your app uses the connection to run queries, updates, etc.
4. Return: Instead of closing, the connection is returned to the pool (via connection.close() internally overridden).
5. Reuse: The next request borrows the same connection again — no new physical connection is created.

Advantages of Connection Pooling

Advantage Explanation • Improved Performance Avoids connection creation overhead • Reuse of connections Connections are reused instead of recreated • Resource Management Limits max DB connections to avoid overload • Better Scalability Supports many concurrent DB users • Connection Monitoring Pool can detect and remove broken/stale connections

Is a Connection Pool Multithreaded?

Yes!

A connection pool is designed to be thread-safe and support multithreaded access because:

- **In typical applications (like web servers), multiple threads run concurrently and all need to access the database.**
- **The pool must handle simultaneous requests for connections from different threads safely.**

How does it work?

- **The connection pool internally manages a thread-safe collection of connections.**
- **When multiple threads call getConnection(), the pool allocates free connections without conflicts or race conditions.**

What about the connections themselves?

• Connections are NOT shared simultaneously between threads. • Each thread gets its own connection instance from the pool. • It's not safe to share a single Connection object across threads at the same time.

EXAMPLE:

- **If the pool has, say, 10 connections, then up to 10 threads can get connections simultaneously.**
- **If the 11th thread asks for a connection while all 10 are in use, it will either:**
  - **Wait until a connection is returned (blocking call), or**
  - **Get an error/timeout/Pool Grow, depending on pool config.**

Analogy:

Imagine the pool as a locker room with 10 lockers (connections): • Multiple people (threads) come to get a locker at the same time. • The locker room has a system (thread-safety) to ensure no two people get assigned the same locker simultaneously.

• If all lockers are taken, newcomers wait in line.

What happens when all pooled connections are in use? 1.

The 11th request tries to get a connection but all 10 connections are currently checked out (busy).

2. How the pool handles this depends on its configuration

Possible behaviours:

Wait (blocking):

- The 11th request waits for a connection to be returned to the pool within a configured timeout. If a connection is returned in time, the request gets it.

Timeout & throw error:

- If no connection becomes free within the timeout, the pool throws an exception (e.g., SQLTimeoutException) indicating no connection available.

Grow pool (if allowed):

- If configured, the pool may create additional connections beyond initial size, up to max pool size, to serve the request.

Reject immediately:

- Some pools can be configured to reject new requests immediately if the pool is exhausted.

NOTE:

• JDBC itself does not have a built-in connection pool. • The core JDBC API (java.sql) only provides basic classes like Connection, Statement, ResultSet, etc., and leaves connection pooling up to the developer or external libraries.

✅ How Connection Pooling is Typically Handled in JDBC

To implement connection pooling in JDBC, you usually rely on third party libraries or application servers, such as:

- **HikariCP – widely used with Spring Boot**
- **Apache DBCP (Database Connection Pooling)**
- **C3P0**

✅ What is HikariCP?

HikariCP is a fast, lightweight, and reliable JDBC connection pooling library (third party). It's the default connection pool in Spring Boot (since version 2.x) due to its performance and low overhead.

Download link:

- **Download jar file(search hikaricp without space in mvnrepository.com(3.1.0))**

  https://mvnrepository.com/artifact/com.zaxxer/HikariCP
- **Downlaod jar file(search slf4j without space in mvnrepository.com (1.7) (INTERNALLY HIKARI USES SLF4J)**

  https://mvnrepository.com/artifact/org.slf4j/slf4j-api/1.7.5

How Connections Are Handled by Default

When you use HikariCP (e.g., in Spring Boot), it automatically:

- **Creates a pool of database connections**
- **Reuses connections for multiple requests**

By default, it uses:

- **maximumPoolSize = 10 connections**
- **idle connection = 10 connections**

What is an Idle Connection?

- **An idle connection is a database connection that is open and ready to use, but not currently being used by any application thread.**

- Idle connections sit in the pool waiting to be borrowed for new database requests.

NOTE:

If you configure:

- dataSource.setMinimumIdle(50); // Idle (ready-to-use) connections
- dataSource.setMaximumPoolSize(100); // Total connections allowed

✅ **At Startup – How Many Connections Are Ready?**

- HikariCP does NOT create all 50 idle connections immediately at startup by default.
- It creates connections lazily, i.e., as needed when requests come in.
- Over time, if demand requires, HikariCP will gradually increase the number of idle connections until it reaches minimumIdle = 50.

What Happens When the 51st Request Comes?

- HikariCP maintains up to 50 idle connections (ready to use). • The pool can grow up to 100 total connections.

If 50 connections are already in use and there are no idle connections available, when the 51st request comes in, HikariCP checks whether the current pool size is below the configured maximumPoolSize (which is 100 in this case). Since the pool has only 50 connections in use, it is still under the limit. Therefore, HikariCP creates a new connection to handle the 51st request, increasing the total pool size to 51. This process continues, growing the pool one connection at a time as needed, until it reaches the maximum of 100 connections.

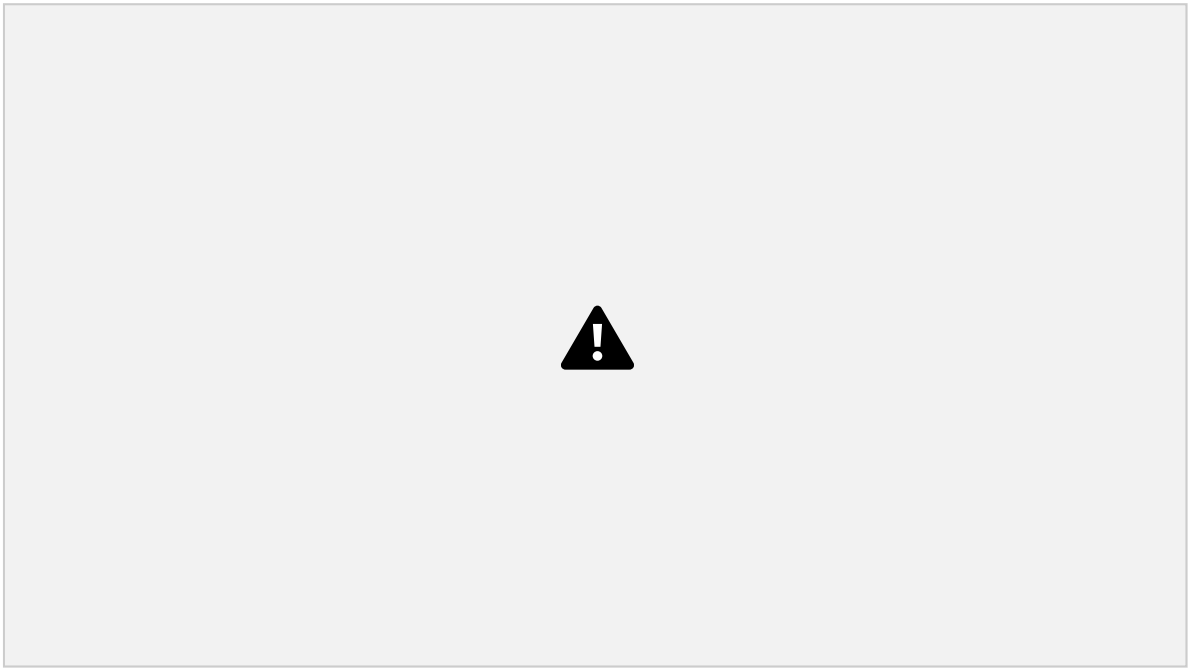
How to Close a Connection Pool

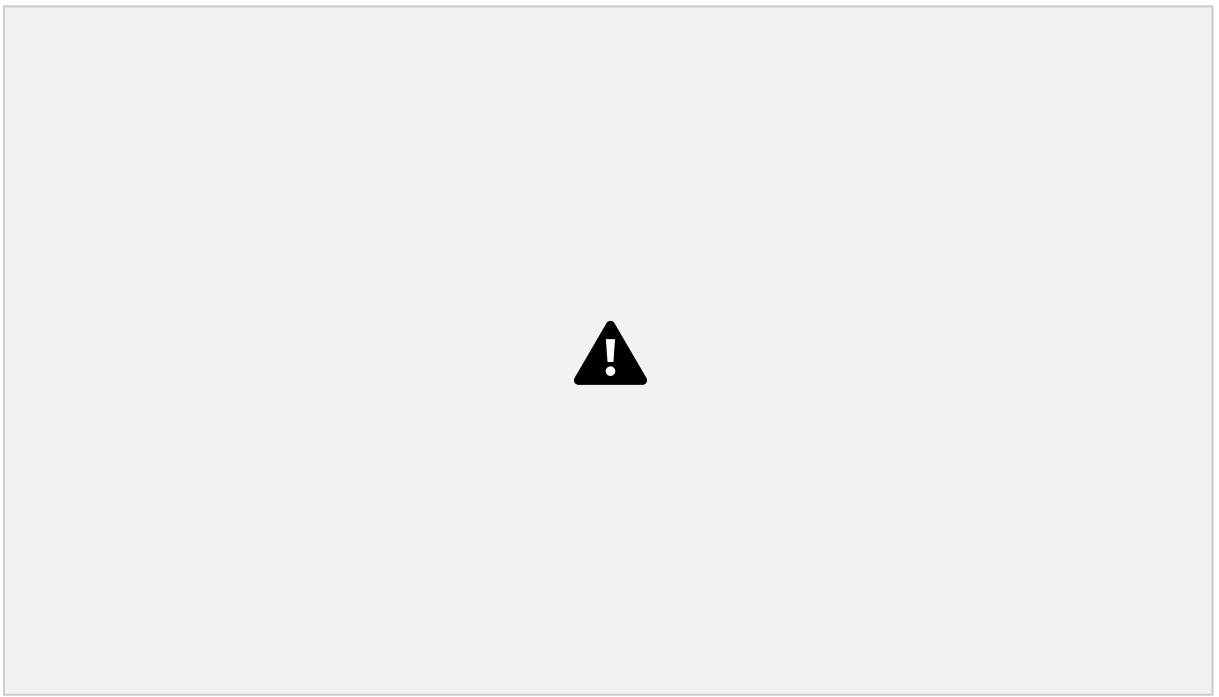
- To close a connection pool, you don't just close individual database connections — you shut down the entire pool, which properly releases all resources held by the pool.

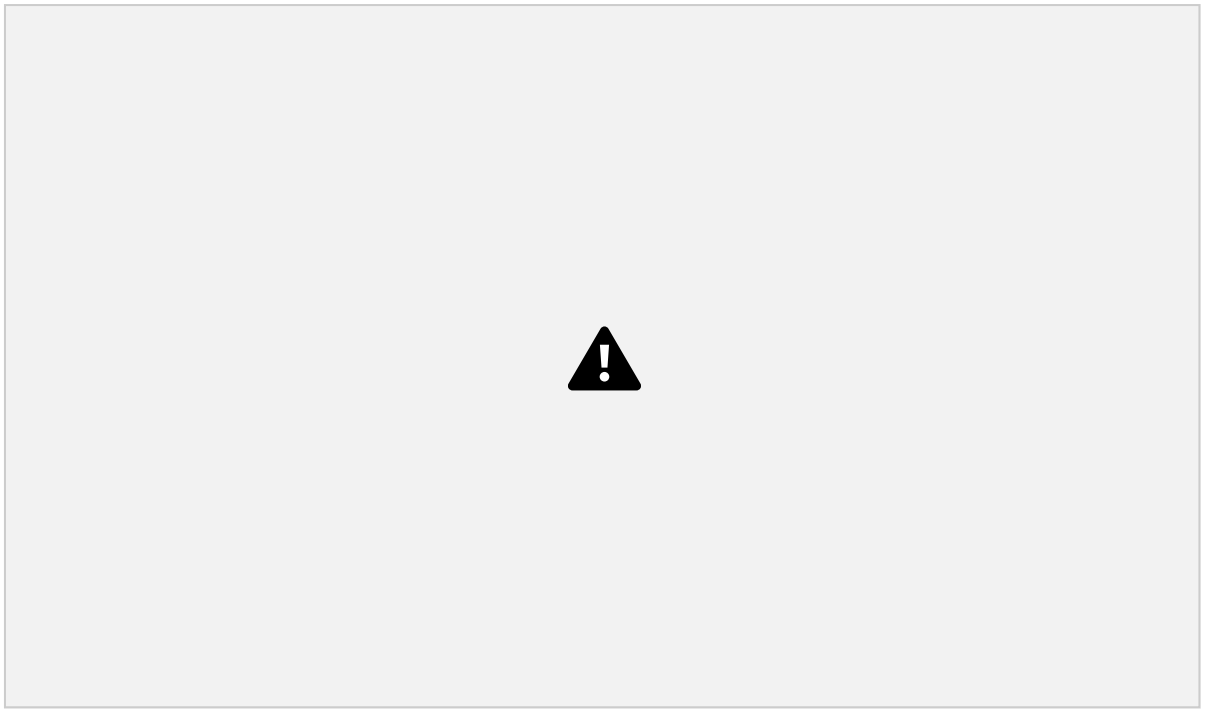
- ***Important: If you only close individual Connection objects (using connection.close()), it does NOT actually close the physical database connection — it just returns it to the pool for reuse.***

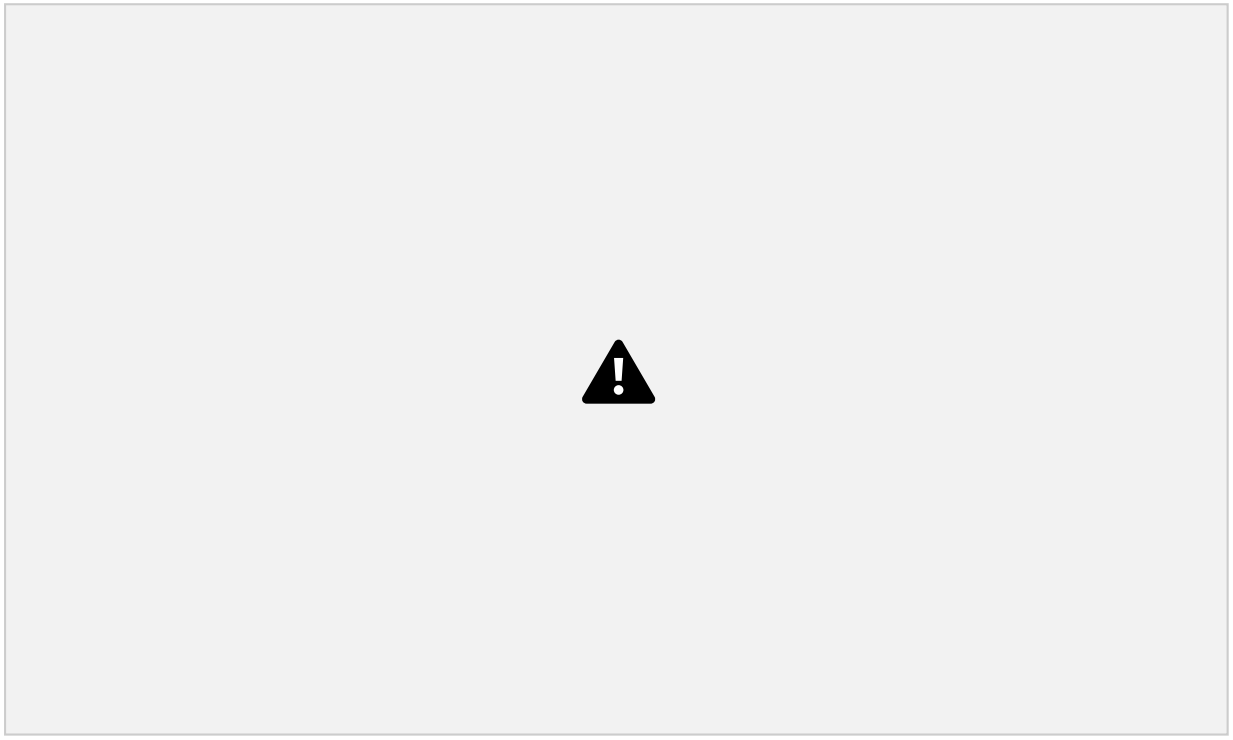
- **To fully release all resources and shut down the pool, you must close the data source itself.**

```
 HikariDataSource dataSource = new
HikariDataSource();
dataSource.close(); // Properly shuts down
the entire connection pool
```

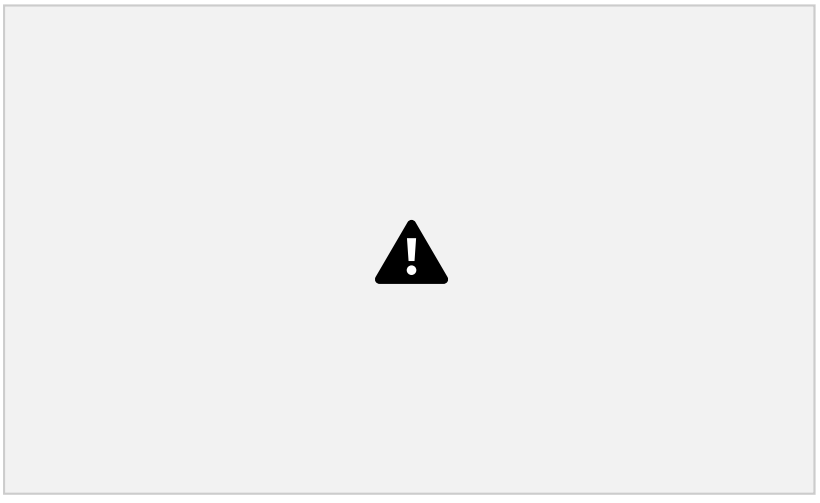








==========PROGRAM: CONNECTION POOLING==========



```
package com.mainapp;
import java.io.FileInputStream;
import java.sql.Connection;
import java.util.Properties;
import com.zaxxer.hikari.HikariConfig;
import

com.zaxxer.hikari.HikariDataSource;

public class Launch {

    public static void main(String[] args) {

        HikariDataSource hikariDataSource=null;

        try {

            FileInputStream fis = new
FileInputStream("myconfig.properties");
            Properties properties = new Properties();
// KEY VALUE LOAD
            properties.load(fis);

            //CONNECTION DETAILS---->HIKARI

            HikariConfig hikariConfig = new
HikariConfig();

    hikariConfig.setJdbcUrl((String)properties.get("m
y sql.url"));

    hikariConfig.setUsername((String)properties.get("
m ysql.user"));
    hikariConfig.setPassword((String)properties.get("
m ysql.pass"));

            hikariConfig.setMinimumIdle(20);
            hikariConfig.setMaximumPoolSize(100);

         hikariDataSource=new
HikariDataSource(hikariConfig);//POOL READY

            Connection connection =
  hikariDataSource.getConnection();//GET FROM THE
```

```
            POOL System.out.println(connection);

                connection.close();//BACK TO POOL

            } catch (Exception e) {
                // TODO: handle exception
            }
            finally {

        hikariDataSource.close();//CLOSING(COMPLETE
            POOL) }
        }
}
```

==========

=

==========

= DAY 9:

RowSet

- **ResultSet always works with an active database connection.**

- **It is child interface of ResultSet**

- **When you use ResultSet, you are directly working with the table in the database. So, if you delete or update data using ResultSet, it immediately affects the table in the database — because you're connected live.**

- ResultSet always works with an active database connection.

- RowSet is like an advanced version of ResultSet that lets you work with database data offline. You can fetch the data once, disconnect from the database, and then read, modify, or even add new records while offline. Later, you can reconnect and sync those changes back to the database.

✅ When to Use RowSet

Use RowSet when:

- You want to disconnect from the database after fetching data.
- You want to work offline (e.g., update, insert, delete while disconnected).
- You need to cache data temporarily, modify it, and sync it later.

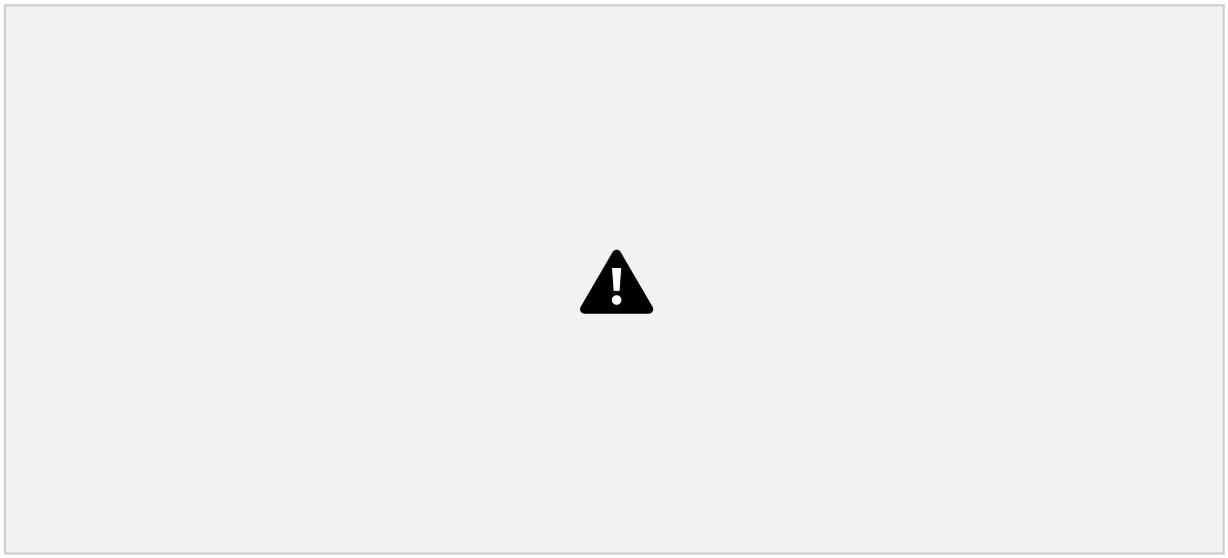
✅ When to Use ResultSet

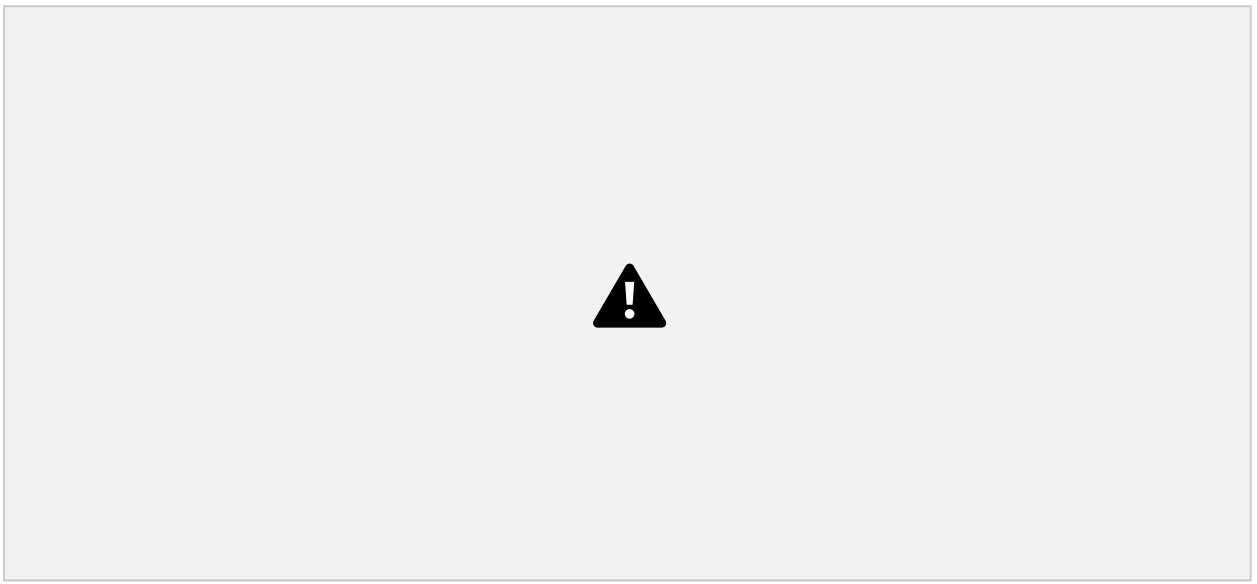
Use ResultSet when:

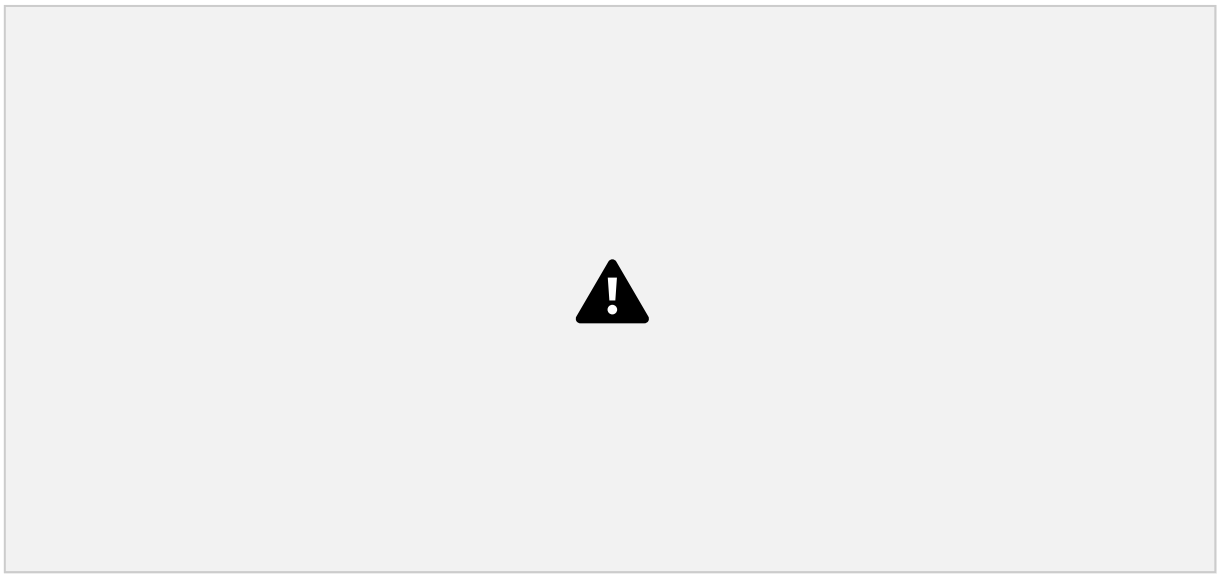
- You are okay with keeping the database connection open during the entire operation.
- You need real-time access to the database.

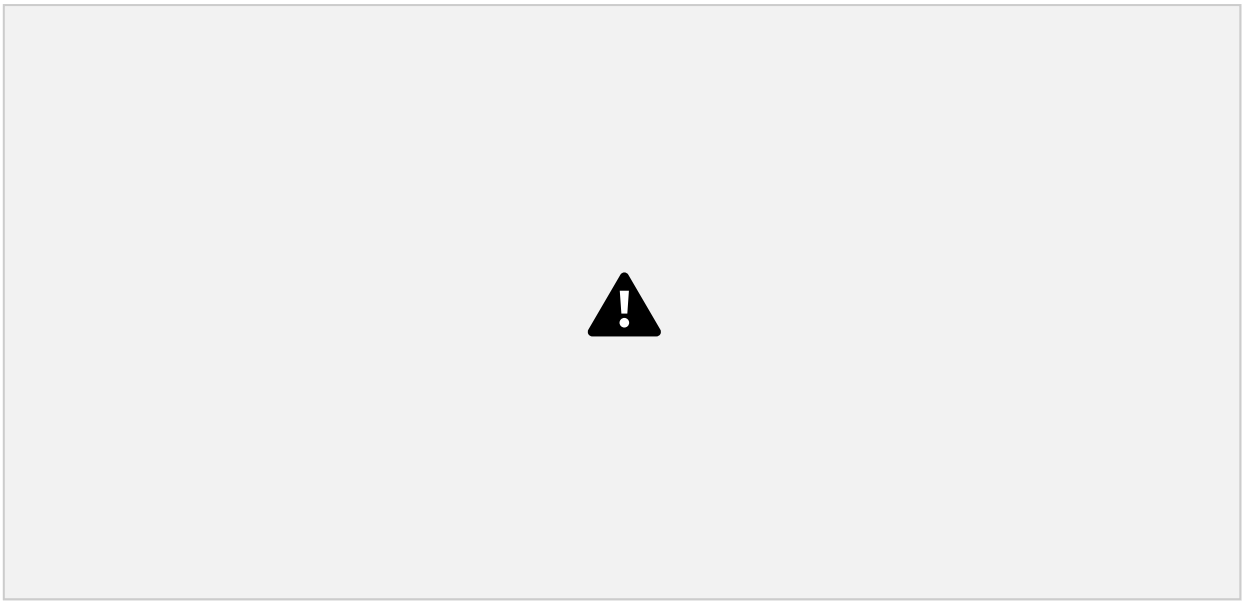
NOTE:

The ResultSet in JDBC is not serializable because it maintains a live connection with the database while accessing or modifying data. This connection-dependent nature makes it unsuitable for scenarios where the data needs to be transferred over a network or saved to a file. Since ResultSet directly interacts with the database, any attempt to serialize it would result in issues due to its dependency on an active database connection. In contrast, RowSet implementations like CachedRowSet are designed to be disconnected and serializable, allowing developers to fetch data, work offline, and later sync changes—making them ideal for distributed applications or offline data manipulation.









==========PROGRAM-1:

ResultSet==========

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Scanner;
public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con = null;
        try {
            String url =
"jdbc:mysql://localhost:3306/maybatch";
            String username = "root";
            String password = "";

             con = DriverManager.getConnection(url,
username, password);
            System.out.println(con);

            PreparedStatement ps =
con.prepareStatement("select * from employee",
ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {

                int sn = rs.getInt("sn");
```

```
                    String fullname =
rs.getString("fullname");

                if (fullname.equals("wwww")) {
                    rs.deleteRow();
                }
            }

            test(rs);

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

    private static void test(ResultSet rs) {

        try {
            rs.beforeFirst();
            while (rs.next()) {
                int sn = rs.getInt("sn");
                 String fullname =
rs.getString("fullname");
                System.out.println(sn);
                System.out.println(fullname);

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

    }
}
```

==========

==========

==========PROGRAM-2:

RowSet==========

```
package com.mainapp;
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Scanner;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetProvider;

public class Launch {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        Connection con = null;
        try {
            String url =
"jdbc:mysql://localhost:3306/maybatch";
            String username = "root";
            String password = "";

             con = DriverManager.getConnection(url,
username, password);
            System.out.println(con);
            PreparedStatement ps =
con.prepareStatement("select * from employee",
ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = ps.executeQuery();

             CachedRowSet cachedRowSet =
RowSetProvider.newFactory().createCachedRowSet()
            ; cachedRowSet.populate(rs); //
            CACHE
MEMORY //OFFLINE(NOT CONNECTED)

            while (cachedRowSet.next()) {

                int sn = cachedRowSet.getInt("sn");
                 String fullname =
cachedRowSet.getString("fullname");

                if (fullname.equals("wwww")) {
                    cachedRowSet.deleteRow();
                }
            }

            test(cachedRowSet);
```

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

    private static void test(CachedRowSet
cachedRowSet) {

        try {
            cachedRowSet.beforeFirst();
            while (cachedRowSet.next()) {

                int sn = cachedRowSet.getInt("sn");
                 String fullname =
cachedRowSet.getString("fullname");
                System.out.println(sn);
                System.out.println(fullname);

            }
        } catch (Exception e) {
```