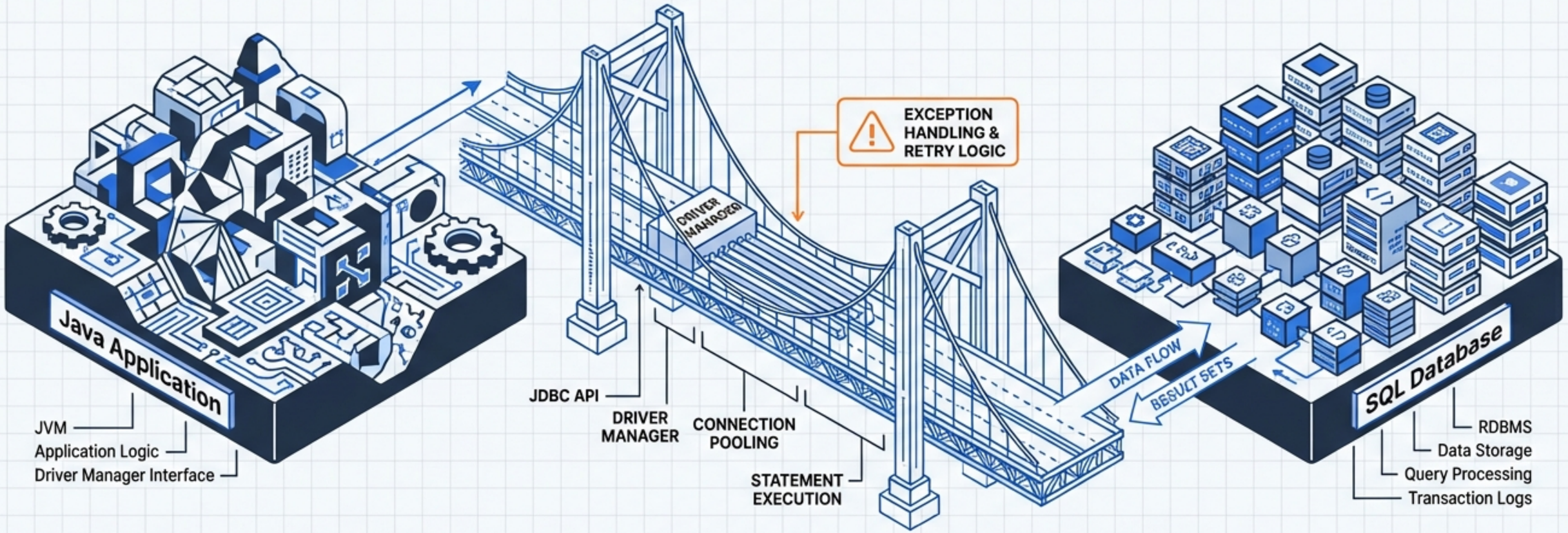
Master JDBC: The Complete Developer's Blueprint

A comprehensive guide to Java Database Connectivity—from architecture to transaction management.



Architecture & Drivers

- JDBC Core Components
- Type 1, 2, 3, & 4 Drivers
- Driver Manager & DataSource
- Connection Lifecycle

Statement vs. PreparedStatement

- Execution Differences
- SQL Injection Prevention ⚠
- Performance Optimization
- Parameterized Queries

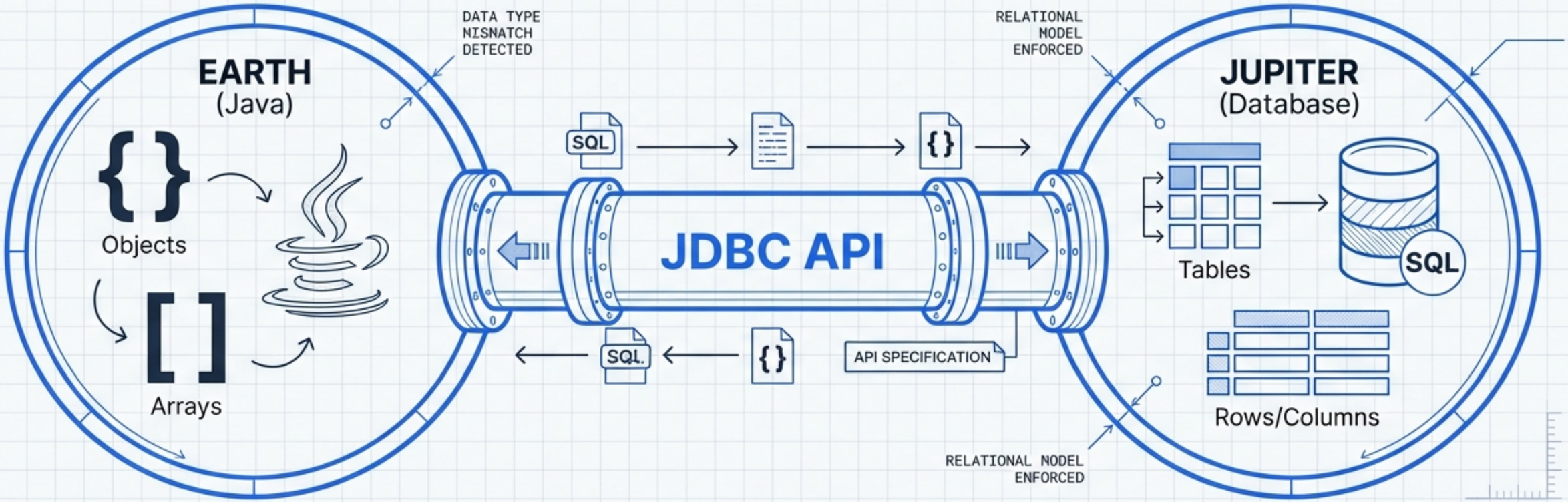
Batch Processing

- addBatch() & executeBatch()
- Reducing Network Roundtrips
- Handling Batch Updates
- Efficient Data Insertion

ACID Transactions

- Atomicity, Consistency, Isolation, Durability
- commit() & rollback()
- Transaction Isolation Levels
- Savepoints & Deadlocks </>

The Planetary Bridge: Java vs. Database



The Problem

Java speaks **Objects**. Databases speak **Tables**. They are **alien worlds** with **different rules**.

The Solution

JDBC acts as the translator—an API specification that allows Java to **talk to any SQL database**.

Key Takeaway

JDBC **is not the database**. ✕
It is the bridge. ←

DOC: TECH-JAVA-DB-BRIDGE-001 | REV: 1.2 | DATE: OCT 26, 2024

Evolution of the JDBC Driver

Type 1: JDBC-ODBC Bridge

Obsolete. Dependent on native OS libraries.
Slow performance.
Written in C.

Type 2: Native API

Vendor-specific.
Requires native client installation.
Hard to maintain.

Type 3: Network Protocol

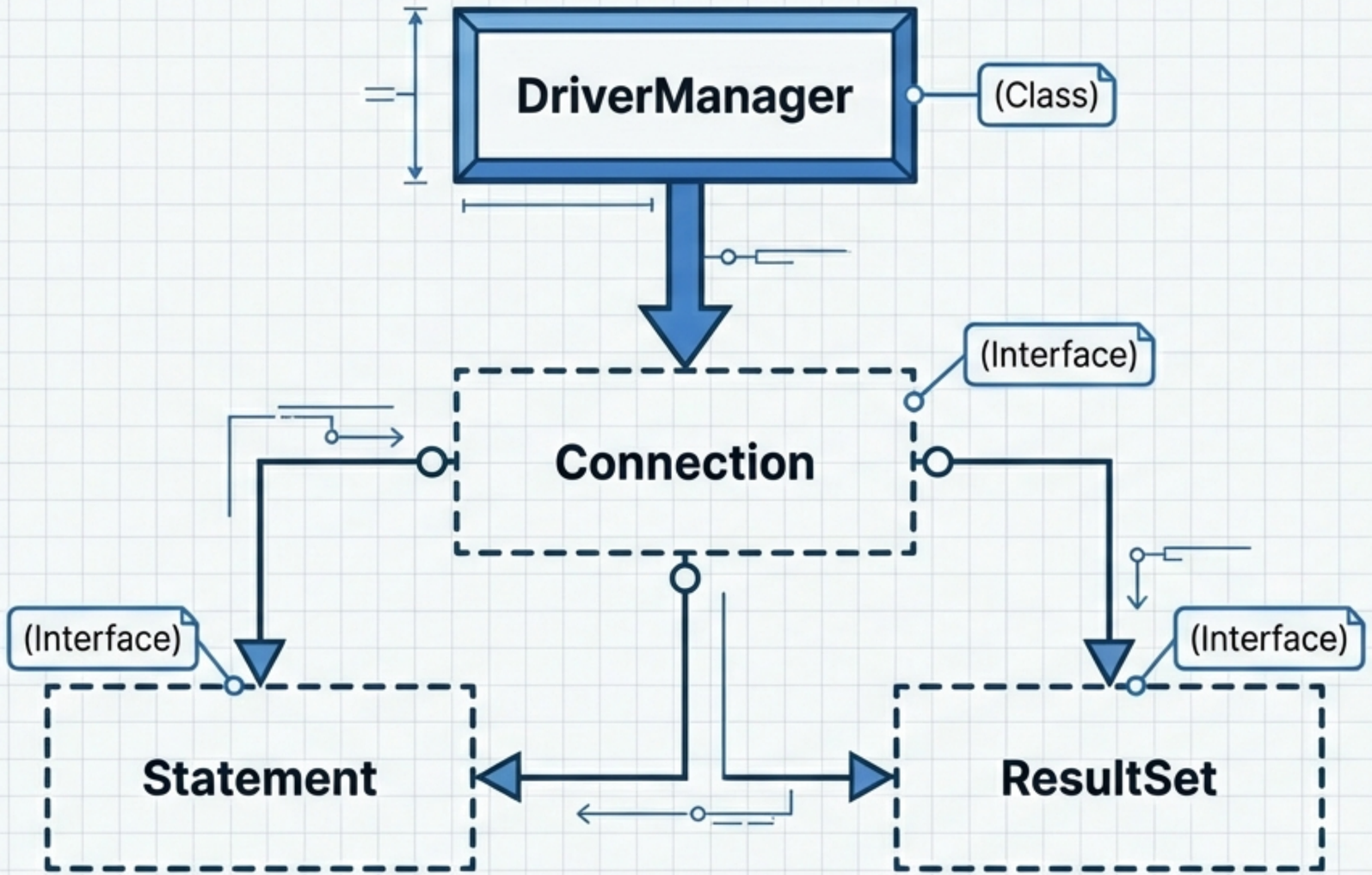
Middleware-based.
Adds latency layer.
Flexible but complex.

Type 4: Thin Driver (The Standard)

Pure Java.
Direct-to-Database.
Lightweight and portable.
No native code required.

We use **Type 4** (e.g., MySQL Connector/J) for direct, efficient communication.

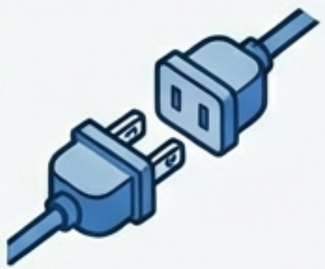
The Big 4: Classes vs. Interfaces



Definitions

- **DriverManager:** The manager that oversees drivers and establishes the connection.
- **Connection:** Represents the active session with the database.
- **Statement**: The vehicle for sending SQL queries.
- **ResultSet**: The container for returned data (the Table).

The 5-Step JDBC Lifecycle

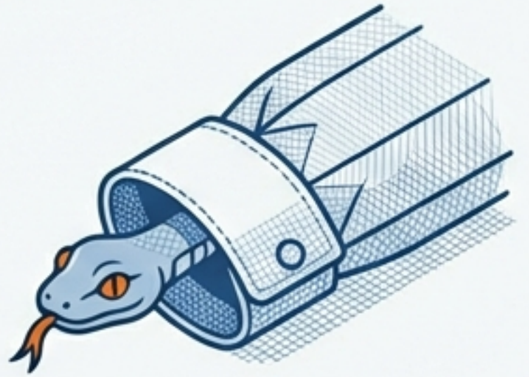
Connect IDE	Load Driver	Create Connection	Create Statement	Execute Query	Close
					
Import JAR / Connector	Register Driver Class	Establish Tunnel	Prepare SQL Vehicle	Send SQL, Get Data	Clean Up Resources

This immutable flow is the backbone of every JDBC application, from simple scripts to enterprise systems.

Step 1 & 2: Loading the Driver

```java
import java.sql.*;

public class Main {
    public static void main(String[] args) {
        try {
            // Load the MySQL Driver
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Exception Handling: The Snake in the Sleeve



- `Class.forName()` dynamically loads the driver class.
- **Risk**: If the JAR is missing, the program crashes.
- **Fix**: The code must be wrapped in a **Try-Catch Block** to handle `ClassNotFoundException`.

Step 3: Establishing the Connection

```
String url = "jdbc:mysql://localhost:3306/myDB";
String user = "root";
String pass = "password123";

// The Connection Object
Connection con = DriverManager.getConnection(url, user, pass);
```

Deconstructed:

Security Note

Best Practice: Avoid hardcoding credentials in production. Use environment variables.

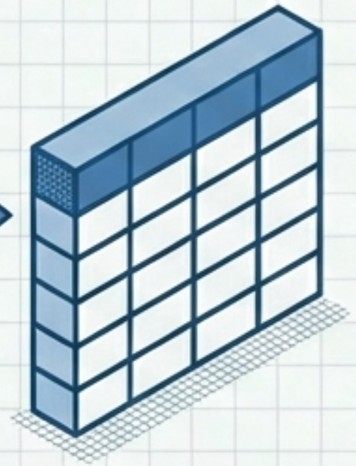
Step 4: Executing Static Queries

Using the `Statement` Interface

Reading Data (SELECT)

Method: `executeQuery(sql)`
Returns: `ResultSet` (A Table of data)

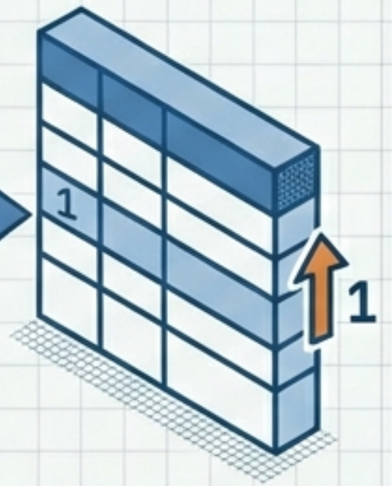
```
Statement stmt =
con.createStatement();
con.createStatement();
ResultSet rs =
stmt.executeQuery(
"SELECT * FROM
students");
```



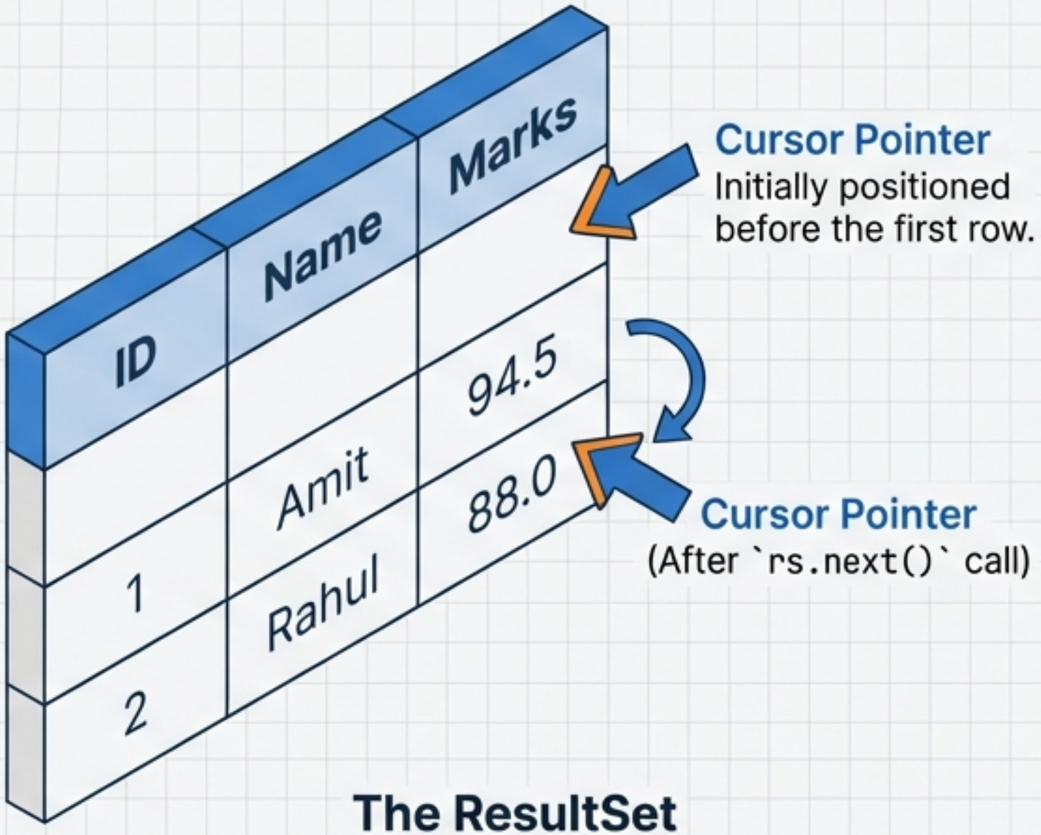
Modifying Data (INSERT, UPDATE, DELETE)

Method: `executeUpdate(sql)`
Returns: `int` (Number of rows affected)

```
Statement stmt =
con.createStatement();
con.createStatement();
int count =
stmt.executeUpdate(
"INSERT INTO students
VALUES('Amit', 21)");
```



Step 5: Decoding the ResultSet



The ResultSet

ResultSet maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row.

```
while (rs.next()) {
    // Move cursor forward
    int id = rs.getInt("id");
    String name = rs.getString("name");

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

- `next()`: Moves cursor to the next row. Returns `false` when no rows remain.
- Getters: `getInt()`, `getString()`, `getDouble()` retrieve specific column data.

Optimization: The PreparedStatement

Why we move away from `Statement`

The Problem

Standard `Statement` requires re-compilation for every query and uses messy string concatenation.

```
"INSERT INTO students VALUES
('" + name + "', " + age + ")"
```

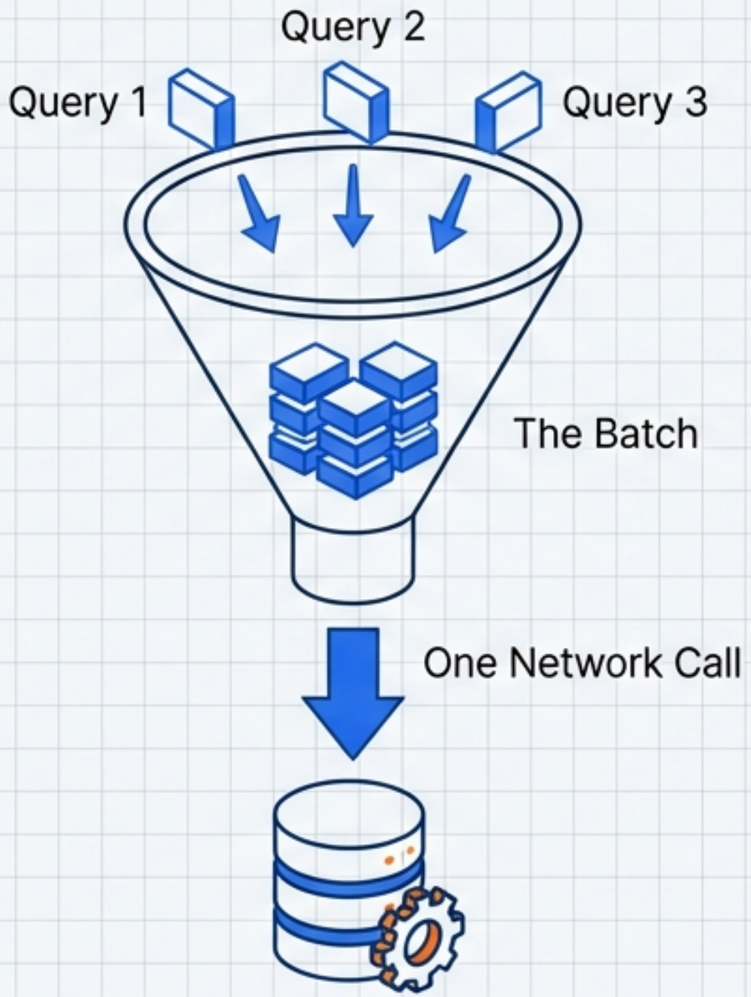
The Solution: PreparedStatement

Pre-compiled once, executed many times. Uses placeholders (`?`) for dynamic data.

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

// Set values for placeholders
pstmt.setString(1, "Rahul"); // First ?
pstmt.setInt(2, 25);         // Second ?

pstmt.executeUpdate();
```

Placeholders

Advanced Pattern: Batch Processing



Instead of executing queries one by one (inefficient), we group them into a batch and ship them together.

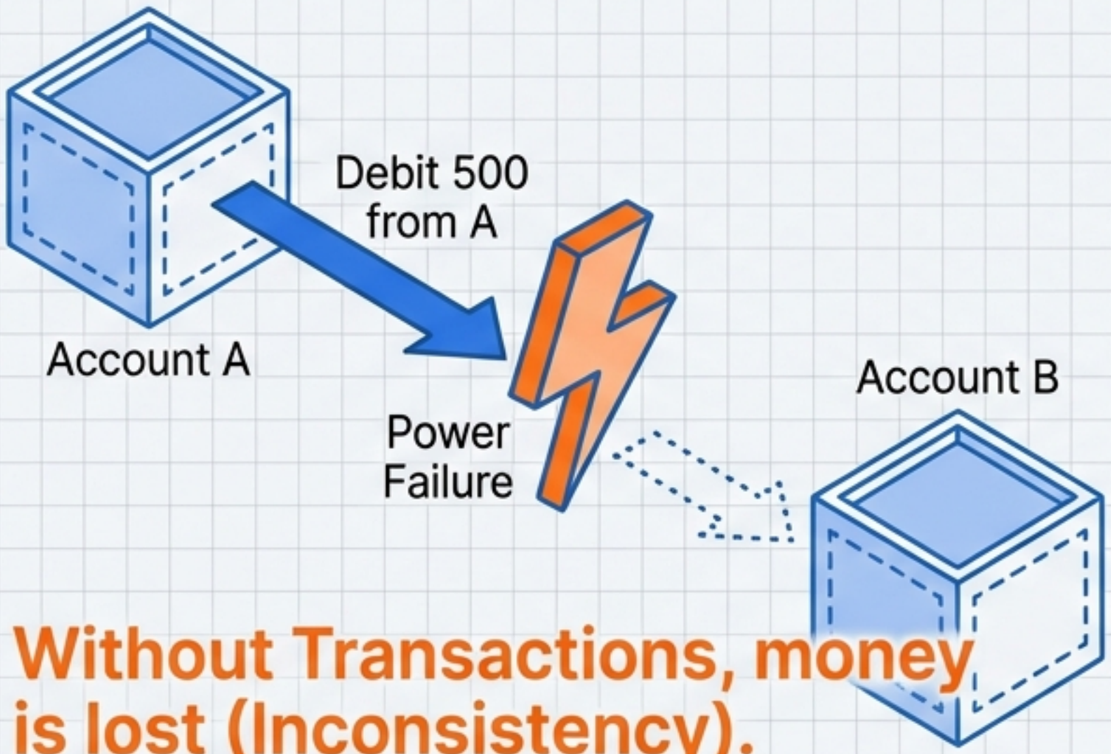
```
// Adding to the bucket
pstmt.setString(1, "Aman");
pstmt.addBatch();

pstmt.setString(1, "Bina");
pstmt.addBatch();

// Firing the bucket
int[] results = pstmt.executeBatch();
```

- Returns an array of status codes (1 = Success, 0 = Fail).

Data Consistency: Transactions

Why we need ACID properties to maintain integrity.

The Bank Transfer



Without Transactions, money is lost (Inconsistency).

The 500 is deducted from A but never reaches B.

The Fix: ACID Properties

Atomicity: All operations must succeed, or none should happen.

Key Command:

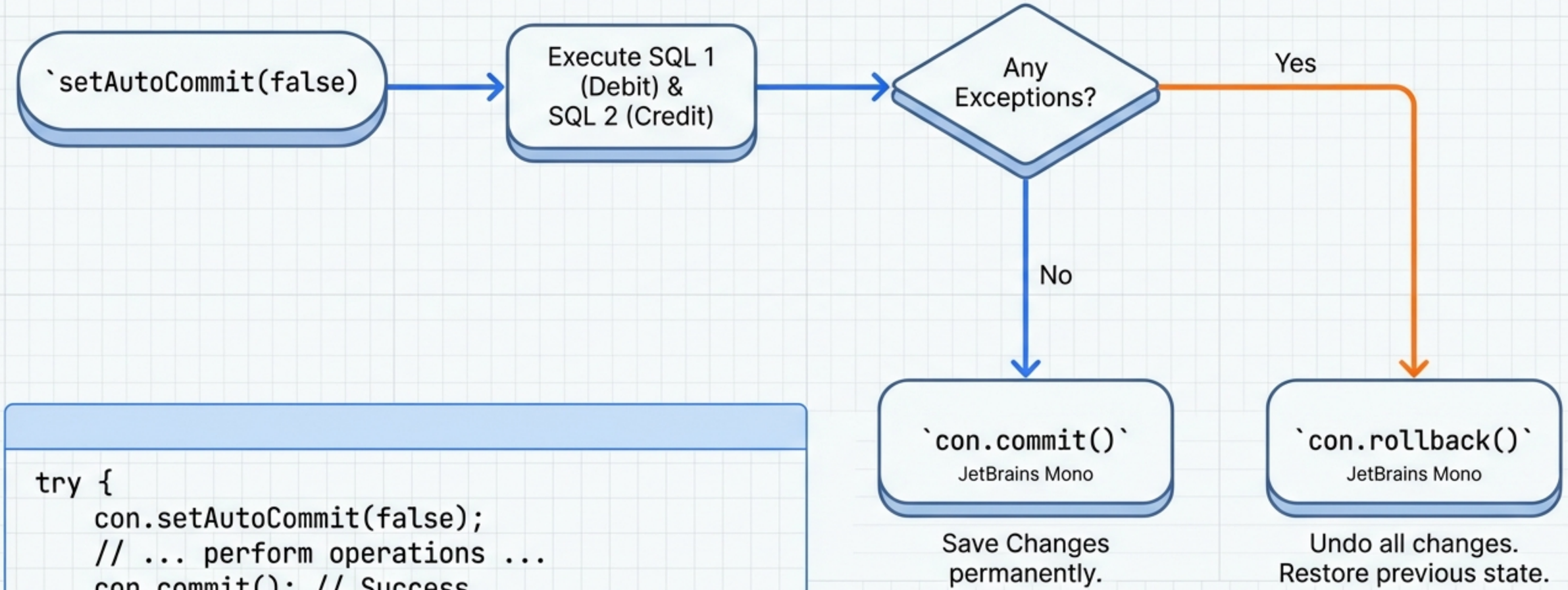
```
con.setAutoCommit(false);
```

↳ By default, JDBC auto-commits. We must disable this to manage the transaction manually.

This allows us to execute multiple queries and either `commit()` them all together or `rollback()` in case of failure, ensuring consistency.

✓ Atomicity, ✓ Consistency, ✓ Isolation ✓ Durability

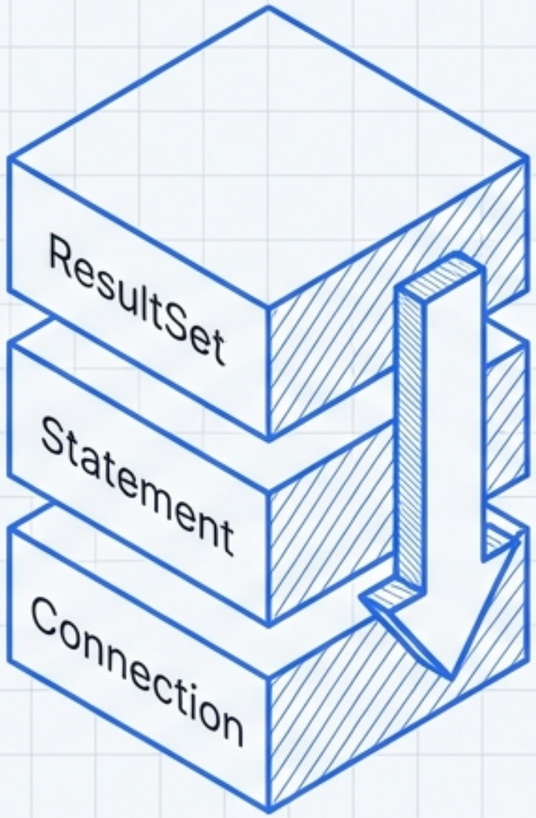
Transaction Logic: Commit & Rollback




```
try {
    con.setAutoCommit(false);
    // ... perform operations ...
    con.commit(); // Success
} catch (SQLException e) {
    con.rollback(); // Failure
}
```

Best Practice: Resource Management

Closing resources is mandatory to prevent memory leaks.



LIFO Cleanup

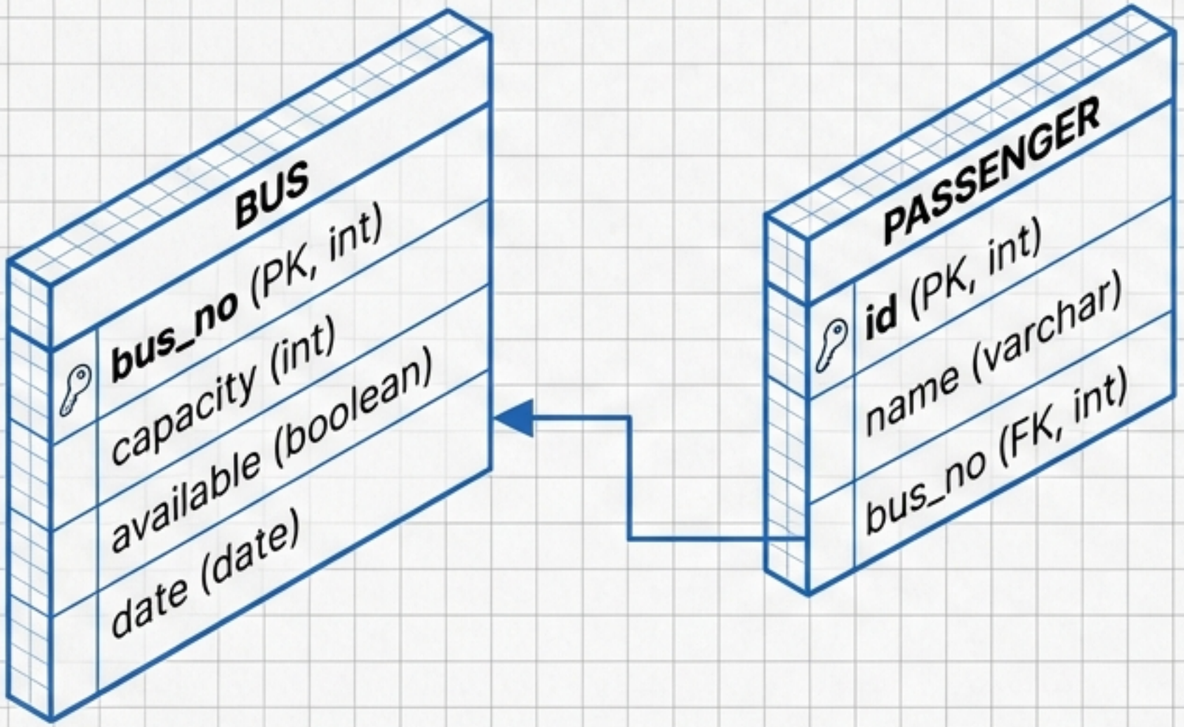
1. ResultSet.close()
2. Statement.close()
3. Connection.close()

```
finally {
  if (rs != null) rs.close();
  if (stmt != null) stmt.close();
  if (con != null) con.close();
}
```

⚠ Modern Java uses 'Try-with-Resources' to automate this.

Capstone Project: The Bus Traveling Agency

Apply your skills to build a booking system



Requirement Checklist

- [ ] Use Type 4 Driver (MySQL Connector)
- [ ] Input details via Scanner
- [ ] Check capacity before booking
- [ ] Use Transactions: Update Passenger + Decrement Capacity atomically.
- [ ] Close all resources.