JDBC Notes

1. Why JDBC?

- JDBC is needed to **connect Java applications with databases** like MySQL, Oracle, PostgreSQL, etc.
- Before JDBC, there were platform-dependent APIs (like ODBC) that were hard to maintain.
- JDBC provides:
  - **Platform Independence** (Write Once, Run Anywhere)
  - **Secure and Reliable** data access
  - **Standard API** to perform CRUD operations (Create, Read, Update, Delete)
  - **Bridges the gap** between Java application and relational databases

---

2. What is JDBC?

- JDBC stands for **Java Database Connectivity**.
- It is a **Java API** that allows Java programs to **interact with relational databases** using SQL.
- It is part of **Java SE** (Standard Edition).
- It includes:
  - Interfaces like Connection, Statement, ResultSet
  - Classes like DriverManager
  - SQL Exception Handling (SQLException)

 Definition:

JDBC is an API that enables Java applications to connect and interact with relational databases using SQL statements.

---

3. What is a JDBC Jar?

- A **JAR (Java Archive)** is a file containing compiled Java classes.
- For JDBC to work, we need a **specific JDBC driver JAR** provided by the database vendor (e.g., mysql-connector-j.jar for MySQL).

- This JAR provides the implementation of JDBC interfaces for a specific database.

 Examples:

- MySQL → mysql-connector-j-8.0.xx.jar
- Oracle → ojdbc8.jar
- PostgreSQL → postgresql-xx.x.x.jar

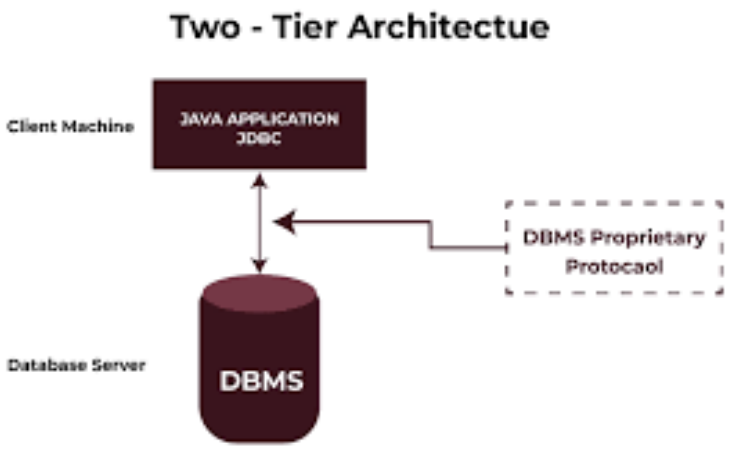
 How to Add JAR in IDE:

- In Eclipse/IntelliJ → Right-click project → Build Path → Add External JARs

---

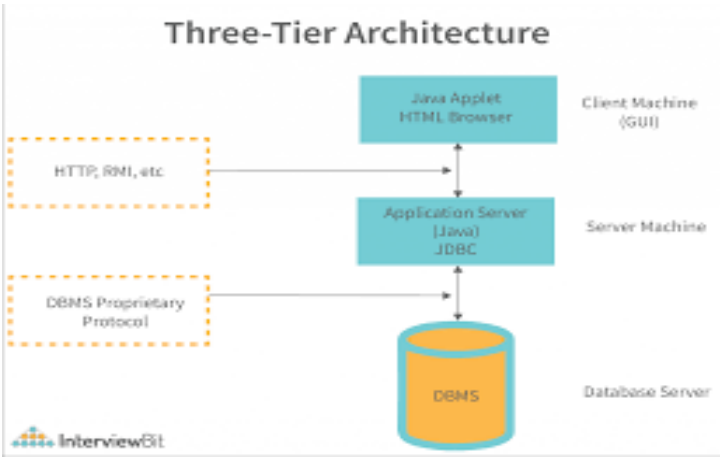
4. JDBC Architecture

▶ Two-Tier Architecture:



- The client application communicates **directly** with the database.

▶ Three-Tier Architecture:



- The client does **not connect directly** to the database. Instead, it goes through a server or middleware.

---

5. JDBC Drivers

Drivers are necessary to **translate Java calls to database-specific calls**. JDBC defines 4 types of drivers:

Type	Name	Description
1	JDBC-ODBC Bridge Driver	Converts JDBC calls to ODBC. **Deprecated**.
2	Native-API Driver	Converts JDBC to DB-specific native calls.
3	Network Protocol Driver	Uses middleware to translate requests.
4	**Thin Driver (Pure Java)**	Converts JDBC to DB protocol directly. Most commonly used (e.g., MySQL, Oracle).

 Recommended: Type 4 driver → mysql-connector-j.jar

 Example Code Snippet:

```
Class.forName("com.mysql.cj.jdbc.Driver"); // Load JDBC Driver
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/db_name", "user", "password");
```

7. Steps to Connect to Database Using JDBC

Connecting a Java application to a database involves the following **5 standard steps**:

Step 1: Import JDBC Packages

```
import java.sql.*;
```

Step 2: Load and Register the Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Step 3: Establish the Connection

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/db_name", "username", "password");
```

Step 4: Create a Statement

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

Step 5: Execute the Query

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

Step 6: Process the Result

```
while(rs.next()) {
   System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
```

Step 7: Close the Resources

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

 These steps are common for executing any SQL operation — SELECT, INSERT, UPDATE, DELETE.

8. JDBC API Overview

The JDBC API consists of a set of **interfaces and classes** provided in the java.sql package. Here are the most important ones:

□ DriverManager

- A class used to **manage JDBC drivers**.
- Establishes a **connection** between Java app and DB.
- Acts like a **central service** for getting Connection objects.

Connection con = DriverManager.getConnection(DB_URL, USER, PASS);

□ Connection

- Represents a **connection** to the database.
- Obtained from DriverManager.
- Used to create Statement, PreparedStatement, or CallableStatement.

Connection con = DriverManager.getConnection(...);

□ Statement

- Used to **execute SQL queries** (static queries).
- Created from a Connection object.

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

□ **Vulnerable to SQL Injection**, not recommended for dynamic queries.

□ ResultSet

- Holds data **returned by SQL SELECT queries**.
- Acts like a **cursor** pointing to a current row of data.

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

 Methods like .getInt(), .getString(), .next() help retrieve data.

SQLException

- Handles all **JDBC-related exceptions**.
- Belongs to java.sql package.
- Can provide:
  - **Error message**: e.getMessage()
  - **SQL state**: e.getSQLState()
  - **Error code**: e.getErrorCode()

```
try {
   ...
} catch(SQLException e) {
   System.out.println("Error: " + e.getMessage());
}
```

9. Executing SQL Queries in JDBC

JDBC provides **three primary methods** in the Statement interface to execute SQL queries.

---

 executeQuery() – For SELECT

- Used when the SQL query returns a **ResultSet**, typically SELECT.
- Returns a ResultSet object containing data.

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

---

 executeUpdate() – For INSERT, UPDATE, DELETE

- Used for **modification queries**.
- Returns an int value representing the **number of rows affected**.

```
int rows = stmt.executeUpdate("UPDATE employees SET salary = 50000
WHERE id = 101");
```

---

 execute() – General Purpose

- Used when you **don't know** if the query returns a ResultSet (SELECT) or just an update count (INSERT/UPDATE).
- Returns boolean:
  - true → ResultSet is returned (e.g., SELECT)
  - false → Update count is returned (e.g., INSERT, UPDATE)

```
boolean status = stmt.execute("SELECT * FROM employees");
if (status) {
   ResultSet rs = stmt.getResultSet();
   // process ResultSet
}
```

---

10. ResultSet Navigation

A ResultSet is like a **cursor** to move through the rows returned by a SELECT query.

 Cursor Movement Methods:

Method	Description
next()	Moves to the next row (most used)
previous()	Moves to the previous row
first()	Moves to the first row
last()	Moves to the last row

 Default ResultSet type allows only **forward movement** (next() only).
For full navigation (first(), last()), create with scrollable ResultSet:

```
Statement stmt = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY
);
```

□ Reading Data using getXXX() methods

Method	Description
getInt(column)	Gets int value from specified column
getString(column)	Gets string value from specified column
getDouble(column)	Gets double value from column
getDate(column)	Gets date from column

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

---

11. Handling Exceptions in JDBC

Using try-catch-finally block

JDBC operations often throw SQLException. Use try-catch-finally for proper handling and resource management.

```
Connection con = null;
Statement stmt = null;

try {
   con = DriverManager.getConnection(...);
   stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

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

```
    }

} catch (SQLException e) {
    System.out.println("Error: " + e.getMessage());
    System.out.println("SQLState: " + e.getSQLState());
    System.out.println("ErrorCode: " + e.getErrorCode());
} finally {
    try {
        if (stmt != null) stmt.close();
        if (con != null) con.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

 SQLException Handling

Method	Description
getMessage()	Returns detailed error message
getSQLState()	Returns the SQLState string
getErrorCode()	Returns vendor-specific error code
printStackTrace()	Prints the complete exception trace

 Always **log exceptions** and **close resources** in finally or use **try-with-resources** (Java 7+).

12. PreparedStatement – Secure & Efficient Query Execution

 What is PreparedStatement?

- A **precompiled** SQL statement used to execute parameterized queries.
- Prevents **SQL Injection attacks**.
- Improves performance for **repeated executions** of the same query with different parameters.

□ Key Advantages:

- **Faster Execution** (query compiled once, used multiple times)
- **Security** (protects against SQL injection)
- **Readability** (clearly separates SQL and parameters)

□ Syntax and Example:

```
String query = "INSERT INTO employees (id, name, salary) VALUES (?, ?, ?)";
PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setInt(1, 101);
pstmt.setString(2, "John");
pstmt.setDouble(3, 50000);

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

□ Common Methods:

Method	Description
setInt(index, value)	Sets an int value at given placeholder
setString(index, value)	Sets a String value
executeUpdate()	Executes INSERT/UPDATE/DELETE
executeQuery()	Executes SELECT query

□ SQL Injection Prevention Example

```
// Unsafe
Statement stmt = con.createStatement();
stmt.executeQuery("SELECT * FROM users WHERE username='" + user + "'");

// Safe
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM users
WHERE username = ?");
```

```
pstmt.setString(1, user);
```

---

13. Scrollable and Updatable ResultSet

By default, a ResultSet is:

- **Forward-only**: can only move from top to bottom.
- **Read-only**: cannot update DB data.

To make it **scrollable** and/or **updatable**, use special ResultSet types and concurrency levels.

□ Types of ResultSet:

Constant	Description
TYPE_FORWARD_ONLY	Default. Moves only forward.
TYPE_SCROLL_INSENSITIVE	Scrollable, but **not sensitive** to DB changes after the query is run.
TYPE_SCROLL_SENSITIVE	Scrollable and **sensitive** to DB changes after the query is run.

Concurrency Modes:

Constant	Description
CONCUR_READ_ONLY	Cannot modify the ResultSet (default)
CONCUR_UPDATABLE	Can **insert, update, delete** rows via ResultSet

---

□ Creating a Scrollable & Updatable ResultSet:

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

□ Cursor Navigation Methods:

Method	Action
rs.next()	Move to next row
rs.previous()	Move to previous row
rs.first()	Move to first row
rs.last()	Move to last row
rs.absolute(n)	Jump to nth row

Updating Data via ResultSet:

```
rs.absolute(2); // Move to 2nd row
rs.updateString("name", "Updated Name");
rs.updateRow(); // Apply update
```

Inserting New Row:

```
rs.moveToInsertRow();
rs.updateInt("id", 105);
rs.updateString("name", "New Employee");
rs.updateDouble("salary", 60000);
rs.insertRow();
```

Deleting Row:

```
rs.absolute(3); // Move to 3rd row
rs.deleteRow();
```

14. CallableStatement – Calling Stored Procedures in JDBC

What is CallableStatement?

- A **JDBC interface** used to **call stored procedures** in a relational database.
- Extends PreparedStatement.

- Allows executing database-side logic with **IN, OUT, and INOUT parameters**.
- Useful when logic is **encapsulated inside stored procedures or functions** for reuse, performance, or security.

 Syntax:

CallableStatement cstmt = con.prepareCall("{call procedure_name(?, ?, ?)}");

 Parameter Types in Stored Procedures:

Type	Direction	Usage
IN	Input only	Pass value from Java to DB
OUT	Output only	Return value from DB to Java
INOUT	Input and Output	Pass value and get modified

Example: Stored Procedure in MySQL

Let's say we have the following stored procedure in MySQL:

```
DELIMITER //

CREATE PROCEDURE getEmployeeName (
   IN empId INT,
   OUT empName VARCHAR(100)
)
BEGIN
   SELECT name INTO empName FROM employees WHERE id = empId;
END //

DELIMITER ;
```

Calling it from JDBC:

```
CallableStatement cstmt = con.prepareCall("{call getEmployeeName(?, ?)}");

// Set IN parameter
cstmt.setInt(1, 101);

// Register OUT parameter
cstmt.registerOutParameter(2, Types.VARCHAR);

// Execute the stored procedure
cstmt.execute();

// Retrieve the OUT parameter
String name = cstmt.getString(2);
System.out.println("Employee Name: " + name);
```

CallableStatement Methods:

Method	Description
setInt(index, value)	Set an IN parameter
setString(index, value)	Set an IN parameter
registerOutParameter(index, SQLType)	Register an OUT parameter
getXXX(index)	Get OUT parameter after execution

Example with INOUT Parameter

```
DELIMITER //

CREATE PROCEDURE increaseSalary (
   INOUT empId INT
)
BEGIN
   UPDATE employees SET salary = salary + 5000 WHERE id = empId;
END //
```

```
DELIMITER ;
```

```
CallableStatement cstmt = con.prepareCall("{call increaseSalary(?)}");

// Register INOUT
cstmt.setInt(1, 101);
cstmt.registerOutParameter(1, Types.INTEGER);

// Execute
cstmt.execute();

int updatedEmpId = cstmt.getInt(1);
System.out.println("Updated for Employee ID: " + updatedEmpId);
```

 When to Use CallableStatement?

- When business logic is stored in the **database**.
- When procedures involve **complex joins**, **batch operations**, or **transactions**.
- For **better performance** (precompiled procedures).
- For **code reusability** and **data security**.

15. JDBC Transaction Management

 What is a Transaction?

A **transaction** is a **sequence of SQL operations** that must be executed as a **single unit**:

- All operations **must succeed**, or
- If one fails, **everything should be rolled back**

 Key Properties (ACID):

- **A**tomicity: All operations complete or none
- **C**onsistency: DB remains valid before/after
- **I**solation: No interference between transactions
- **D**urability: Once committed, changes are permanent

Auto-commit vs Manual Commit

Auto-commit (Default Mode)

- Every SQL statement is **automatically committed** once it is executed.
- You **cannot group statements into a single transaction**.

```
Connection con = DriverManager.getConnection(...);
System.out.println(con.getAutoCommit()); // true
```

Manual Commit (Recommended for Transactions)

- Set autoCommit(false) to control transaction manually.

```
con.setAutoCommit(false); // disable auto-commit
```

- You can now group multiple queries and either:

commit() – Save all changes permanently

```
con.commit();
```

rollback() – Undo all uncommitted changes

```
con.rollback();
```

Example: Transaction with Manual Commit

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

   Statement stmt = con.createStatement();

   stmt.executeUpdate("UPDATE accounts SET balance = balance - 1000
WHERE id = 1");
   stmt.executeUpdate("UPDATE accounts SET balance = balance + 1000
WHERE id = 2");

   con.commit(); // Both updates succeed
```

```
    System.out.println("Transaction committed successfully.");
} catch (SQLException e) {
    con.rollback(); // If any query fails
    System.out.println("Transaction rolled back due to error: " + e.getMessage());
}
```

 Key Methods Summary

Method	Description
setAutoCommit(false)	Disable auto-commit to begin manual control
commit()	Finalize and save all SQL operations
rollback()	Undo all SQL operations since last commit

Use Case Example:

 Fund Transfer System

- Debit from one account
- Credit to another
- Both should happen together → Use transaction

16. Batch Processing in JDBC

 What is Batch Processing?

- Batch processing allows **grouping multiple SQL statements** and executing them together in one go.
- Significantly **improves performance**, especially during **bulk inserts/updates**.
- Reduces **network round-trips** to the database.

 Methods Involved:

Method	Description
addBatch()	Adds a SQL command to the batch
executeBatch()	Executes all commands added in the batch
clearBatch()	Clears the current batch of commands

Example: Inserting Multiple Records Efficiently

```
Connection con = DriverManager.getConnection(...);
PreparedStatement pstmt = con.prepareStatement(
   "INSERT INTO students (id, name, marks) VALUES (?, ?, ?)");

con.setAutoCommit(false); // Optional but improves performance

for (int i = 1; i <= 5; i++) {
   pstmt.setInt(1, i);
   pstmt.setString(2, "Student" + i);
   pstmt.setInt(3, 70 + i);
   pstmt.addBatch(); // Add to batch
}

int[] counts = pstmt.executeBatch(); // Execute all inserts at once
con.commit(); // Commit transaction

System.out.println("Inserted rows: " + counts.length);
```

Why Use Batch Processing?

- Speeds up bulk inserts/updates
- Reduces DB calls
- Helps in large-scale data migration, logs, etc.

---

17. Metadata in JDBC

Metadata provides **information about the database** or the **structure of a result set** — very useful for building **generic or dynamic applications**.

---

1. DatabaseMetaData

Used to retrieve **information about the database itself**.

```
DatabaseMetaData dbmd = con.getMetaData();

System.out.println("Database Product Name: " +
dbmd.getDatabaseProductName());
System.out.println("Database Version: " + dbmd.getDatabaseProductVersion());
System.out.println("Driver Name: " + dbmd.getDriverName());
System.out.println("User Name: " + dbmd.getUserName());
```

 You can retrieve:

- All table names
- Column details
- Supported SQL features
- Driver info

---

2. ResultSetMetaData

Used to get **info about the columns in a ResultSet**.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
ResultSetMetaData rsmd = rs.getMetaData();

int columnCount = rsmd.getColumnCount();
System.out.println("Total Columns: " + columnCount);

for (int i = 1; i <= columnCount; i++) {
   System.out.println("Column " + i + ": " + rsmd.getColumnName(i));
   System.out.println("Type: " + rsmd.getColumnTypeName(i));
```

}

 Helpful When:

- Dynamically displaying column names
- Creating generic DB tools
- Handling unknown table structures

18. Connection Pooling in JDBC – Basic Introduction

What is Connection Pooling?

- **Connection Pooling** is a technique to **reuse** database connections instead of creating a new one every time.
- JDBC Connection objects are **expensive** to create — pooling improves performance.
- A **pool** is a collection of pre-created Connection objects that are reused across requests.

Why Use Connection Pooling?

Without Pooling	With Pooling
New connection is created every time	Connection is reused from the pool
High overhead, slow performance	Fast and efficient
Resource leakage possible	Managed and monitored automatically

---

How It Works:

1. A **pool** of connections is created at startup (e.g., 10 connections).
2. When a DB connection is needed, a connection is **borrowed** from the pool.
3. After use, it is **returned** to the pool (not closed).
4. Connections can be **reused** multiple times.

□ Popular Connection Pooling Libraries

Library	Features / Notes
Apache DBCP (Database Connection Pooling)	Stable, simple to configure. Part of Apache Commons.
HikariCP	Fastest, most efficient. Default in Spring Boot.
C3P0	Easy setup, slower compared to HikariCP.

□ Basic Example with Apache DBCP (Using BasicDataSource)

```
import org.apache.commons.dbcp2.BasicDataSource;
import javax.sql.DataSource;

BasicDataSource ds = new BasicDataSource();
ds.setUrl("jdbc:mysql://localhost:3306/mydb");
ds.setUsername("root");
ds.setPassword("password");
ds.setMinIdle(5);
ds.setMaxIdle(10);
ds.setMaxOpenPreparedStatements(100);

Connection con = ds.getConnection(); // Borrow connection
// Use con here
con.close(); // Returns connection to pool
```

□ HikariCP Example (Spring Boot – auto-configured)

```
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.hikari.maximum-pool-size=10
```

□ Spring Boot uses **HikariCP by default** for blazing-fast connection management.

□ Benefits of Connection Pooling

- **Reusability**: No need to create new connections every time
- **Performance**: Faster response times, especially under high load
- **Resource Management**: Controlled number of DB connections
- **Secure & Scalable**: Ideal for enterprise-grade systems

19. JNDI and DataSource (Enterprise JDBC Connection Management)

□ What is JNDI?

- **JNDI** stands for **Java Naming and Directory Interface**.
- It is an **API used to look up resources** (like database connections, EJBs, configuration) using names.
- In enterprise applications (like in **Tomcat**, **JBoss**, **WebLogic**), **DataSource objects** are bound to a JNDI name, and Java apps use this name to obtain DB connections.

□ What is DataSource?

- An **alternative to DriverManager** to get JDBC connections.
- Supports **connection pooling** and **distributed transactions**.
- Often used in **enterprise servers** and **Spring applications**.

□ Why Use DataSource over DriverManager?

Feature	DriverManager	DataSource
Connection Pooling	Not supported	Fully supported
JNDI Lookup	Not possible	Can lookup via JNDI
Configuration Flexibility	Hardcoded	External, XML/config-based
Recommended in Prod	No	Yes (Enterprise standard)

□ JNDI Lookup Example (in Servlet)

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb");
Connection con = ds.getConnection();
```

□ **Note:** java:comp/env/jdbc/mydb is configured in context.xml (Tomcat) or web.xml.

□ Example context.xml (Apache Tomcat)

```
<Context>
  <Resource name="jdbc/mydb"
            auth="Container"
            type="javax.sql.DataSource"
            maxTotal="20"
            maxIdle="10"
            driverClassName="com.mysql.cj.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/mydb"
            username="root"
            password="password"/>
</Context>
```

---

20. RowSet Interface – Disconnected, Scrollable ResultSet

□ What is RowSet?

- A **wrapper around ResultSet**, providing **scrollable**, **serializable**, and **disconnected** capabilities.
- Part of javax.sql package.
- Can be used in **JavaBeans**, **GUI**, **offline data access**, or **web services**.

□ Types of RowSet:

RowSet Type	Description
JdbcRowSet	Connected, scrollable ResultSet
CachedRowSet	Disconnected, in-memory ResultSet
WebRowSet	XML-based representation of RowSet
FilteredRowSet	Apply filter logic on data
JoinRowSet	Combine data from multiple RowSets (like SQL JOIN)

 Why Use RowSet?

Feature	Benefit
Disconnected access	No need to keep DB connection open
Scrollable	Move forward, backward, randomly
Serializable	Can send over network / save to disk
GUI Friendly	Easily bind to Swing or UI components

CachedRowSet Example

```
CachedRowSet crs = RowSetProvider.newFactory().createCachedRowSet();
crs.setUrl("jdbc:mysql://localhost:3306/mydb");
crs.setUsername("root");
crs.setPassword("password");
crs.setCommand("SELECT * FROM students");
crs.execute(); // Executes & stores data in memory

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

 CachedRowSet works **offline** after execution — good for batch processing or GUI forms.

21. Handling Large Objects (LOBs) in JDBC

□ What are LOBs?

- LOB = **Large Object** used to store large binary/text data in databases.

Type	Full Form	Used For
BLOB	Binary Large Object	Images, audio, video, PDFs
CLOB	Character Large Object	Large text files, documents

□ Inserting a BLOB (e.g., Image)

```
FileInputStream fis = new FileInputStream("image.jpg");
PreparedStatement pstmt = con.prepareStatement("INSERT INTO images (id, photo) VALUES (?, ?)");
pstmt.setInt(1, 101);
pstmt.setBinaryStream(2, fis, fis.available());
pstmt.executeUpdate();
```

□ Retrieving a BLOB

```
ResultSet rs = stmt.executeQuery("SELECT photo FROM images WHERE id = 101");
if (rs.next()) {
   Blob blob = rs.getBlob("photo");
   InputStream is = blob.getBinaryStream();
   // Save or display image
}
```

□ Handling CLOB (e.g., Text File)

```
FileReader fr = new FileReader("notes.txt");
PreparedStatement pstmt = con.prepareStatement("INSERT INTO documents (id, content) VALUES (?, ?)");
pstmt.setInt(1, 1);
pstmt.setCharacterStream(2, fr);
```

```
pstmt.executeUpdate();
```

---

22. JDBC in Web Applications

□ Where is JDBC Used?

- **Servlets** and **JSP pages** use JDBC to interact with the database in a **web application**.
- Common operations: login, registration, form processing, CRUD, session tracking.

□ Typical Workflow in a Servlet

```
public void doPost(HttpServletRequest req, HttpServletResponse res) {
   String username = req.getParameter("username");
   String password = req.getParameter("password");

   try {
      Connection con = DriverManager.getConnection(...);
      PreparedStatement pstmt = con.prepareStatement("SELECT * FROM users
WHERE username=? AND password=?");
      pstmt.setString(1, username);
      pstmt.setString(2, password);

      ResultSet rs = pstmt.executeQuery();
      if (rs.next()) {
         // valid user → forward to home.jsp
      } else {
         // invalid login
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

□ Best Practices in Web Apps:

Practice	Why Important
Use **Connection Pooling**	Avoid performance issues
Close resources properly	Prevent memory leaks
Use **DAO classes**	Separation of concerns
Never expose JDBC code in JSP	JSP is for UI only
Share connection across request/session wisely	Avoid redundant DB hits

---

23. ORM vs JDBC – Why Hibernate?

□ What is ORM?

- **ORM (Object Relational Mapping)** is a technique to **map Java objects to database tables**.
- Example: **Hibernate**, **JPA (Java Persistence API)**

□ Why Use ORM Frameworks?

JDBC (Manual)	ORM (Hibernate)
Manually write SQL queries	Auto-generate SQL from Java objects
No caching by default	Built-in caching support
No object-table mapping	Maps classes to tables directly
Tedious relationship management	Easy with annotations
No transaction abstraction	Full transaction support

□ JDBC Limitations:

- Verbose and repetitive code
- Difficult to manage complex joins
- No caching or lazy loading
- Not database-agnostic (SQL is hardcoded)

- No built-in support for relationships (e.g., One-to-Many)

 When to Use JDBC vs ORM?

Use JDBC When...	Use ORM When...
App is small or SQL-specific	App is enterprise-grade
You need fine-grained SQL control	You want rapid dev with less SQL
You're building a JDBC learning project	You need to focus on business logic

---

 All major **Spring Boot** applications use **Hibernate** (JPA) under the hood but may use JDBC for **read-only**, **audit**, or **batch** modules