Java—RDBMS & Database Programming with JDBC

1.Introduction to JDBC

- **Theory:**

o What is JDBC (Java Database Connectivity)?

->It is an java API which enable the java program to execute sql statements.

->It is an application programming interface that defines how a java programmer can access the database in tabular format from java code using a set of standard interfaces and classes written in the java programming languages.

o Importance of JDBC in Java Programming

- Database Connectivity: Enables Java applications to interact with relational databases.
- Platform Independence: Works across different databases without major changes.
- Simplified Communication: Abstracts complexities in database interactions.
- Versatile Features: Supports queries, transactions, and batch processing.
- Integration: Seamlessly works with Java frameworks like Spring and Hibernate.
- Enterprise Applications: Essential for building scalable, data-driven software.

o JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet

1. Driver Manager
   - Handles loading and managing database drivers.
   - Establishes connections between Java applications and databases by selecting the appropriate driver.
2. Driver
   - Acts as an interface for communicating with the database.
   - Each database has its own specific driver (e.g., MySQL driver, Oracle driver).
3. Connection
   - Represents a session between the Java application and the database.
   - Used to execute SQL commands and manage transactions.
4. Statement
   - Interface for executing SQL queries (e.g., Statement, Prepared Statement, and Callable Statement).
   - Allows sending queries to the database and returning results.
5. Result Set

- Represents the data retrieved from the database as a result of SQL queries.
- Provides methods to navigate and access the data (e.g., moving forward, backward, etc.).

2. JDBC Driver Types

• Theory:

o Overview of JDBC Driver Types:

♣ Type 1: JDBC-ODBC Bridge Driver

♣ Type 2: Native-API Driver

♣ Type 3: Network Protocol Driver

♣ Type 4: Thin Driver

- Type 1 (JDBC-ODBC Bridge): Uses ODBC drivers; easy to set up but slow and not portable.
- Type 2 (Native-API): Converts JDBC calls to native database API; faster but platform-dependent.
- Type 3 (Network Protocol): Uses middleware for database communication; good for networks but needs extra setup.
- Type 4 (Thin Driver): Directly connects to databases; fast, portable, and widely used.

o Comparison and Usage of Each Driver Type

Driver Type	Description	Pros	Cons	Usage
Type 1: JDBC-ODBC Bridge	Bridges Java to ODBC for database access	Easy to set up for legacy systems	Slow, not portable, and deprecated	Rarely used today, mainly for legacy systems
Type 2: Native-API Driver	Uses native database APIs through Java	Faster than Type 1	Requires database-specific libraries	Suitable for applications with fixed databases
Type 3: Network Protocol Driver	Uses middleware to translate JDBC calls	Network access enabled	Middleware configuration needed	Used in distributed systems and networks
Type 4: Thin Driver	Direct communication with the database	Portable, efficient, and fast	Specific to database being used	Most commonly used in

				modern applications

3. Steps for Creating JDBC Connections

- **Theory:**

o Step-by-Step Process to Establish a JDBC Connection:

1. Import the JDBC packages

2. Register the JDBC driver

3. Open a connection to the database

4. Create a statement

5. Execute SQL queries

6. Process the result set

7. Close the connection

->1. Import the JDBC Packages

- Include the required JDBC classes by importing java.sql.*
- Example: import java.sql.*;

2. Register the JDBC Driver

- Load the database-specific driver using Class.forName("DriverClassName").
- Example: Class.forName("com.mysql.cj.jdbc.Driver");

3. Open a Connection to the Database

- Establish a connection using the DriverManager.getConnection() method.

  E.g-> Connection connection = DriverManager.getConnection(

  "jdbc:mysql://localhost:3306/yourDatabaseName", "username", "password");

4. Create a Statement

- Use the Connection object to create a Statement for executing SQL commands.

- Example:

  Statement stmt = connection.createStatement();

5. Execute SQL Queries

- Run SQL queries using the Statement object, such as executeQuery() for SELECT or executeUpdate() for INSERT/UPDATE/DELETE.
- Example:

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

6. Process the Result Set

- Use the ResultSet object to retrieve and process the data returned by the query.
- Example:

```
while (rs.next()) {
   System.out.println(rs.getString("columnName"));
}
```

7. Close the Connection

- Release resources by closing the ResultSet, Statement, and Connection.
- Example:

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

4. Types of JDBC Statements

- **Theory:**

o Overview of JDBC Statements:

♣ Statement: Executes simple SQL queries without parameters.

♣ Prepared Statement: Precompiled SQL statements for queries with parameters.

♣ Callable Statement: Used to call stored procedures.

- o Statement:
  - ➢ Executes basic SQL queries like SELECT, INSERT, etc.
  - ➢ Does not support parameters.
  - ➢ Suitable for static SQL queries but less efficient for repeated executions.
- o Prepared Statement:

- Used for SQL queries with parameters (e.g., ? placeholders for dynamic input).
- Precompiled for efficiency, making it faster for repeated execution.
- Helps prevent SQL injection attacks, enhancing security.
    - Callable Statement:
- Used to call stored procedures in the database.
- Allows input, output, and input-output parameters.
- Ideal for complex database operations defined as procedures.

5. JDBC CRUD Operations (Insert, Update, Select, Delete)

• Theory:

o Insert: Adding a new record to the database.

->INSERT INTO tableName (column1, column2)

VALUES ('value1', 'value2');

o Update: Modifying existing records.

-> UPDATE tableName

SET column1 = 'newValue'

WHERE column2 = 'conditionValue';

o Select: Retrieving records from the database.

-> SELECT *

FROM tableName

WHERE column1 = 'searchValue';

o Delete: Removing records from the database.

-> DELETE FROM tableName

WHERE column1 = 'conditionValue';

6. ResultSet Interface

• **Theory:**

o What is ResultSet in JDBC?

o Navigating through ResultSet (first, last, next, previous)

o Working with ResultSet to retrieve data from SQL queries

-> In JDBC, ResultSet is an object that represents the data returned from the database as a result of executing a SQL query, typically a SELECT statement. It allows developers to access and navigate the retrieved records in a tabular format.

Key Features of ResultSet:

- Access Data: Retrieve data from rows and columns using methods like getString(), getInt(), etc.
- Navigation: Move through the result set using methods like next(), previous(), first(), last(), etc.

  e.g->String query = "SELECT * FROM tableName";

  Statement stmt = connection.createStatement();

  ResultSet rs = stmt.executeQuery(query);

  while (rs.next()) {

    System.out.println("Column1: " + rs.getString("column1"));

    System.out.println("Column2: " + rs.getInt("column2"));

}

Method	**Description**
next()	Moves the cursor to the next row. (Commonly used)
previous()	Moves the cursor to the previous row.
first()	Moves the cursor to the first row.
last()	Moves the cursor to the last row.

Method	**Usage**
getString(columnName)	Retrieves data from a column as a string.
getInt(columnName)	Retrieves data from a column as an integer.
getDate(columnName)	Retrieves data from a column as a date.

e.g String query = "SELECT id, name FROM employees";

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

```
    int id = rs.getInt("id");                    // Getting integer data
    String name = rs.getString("name");    // Getting string data
    System.out.println("ID: " + id + ", Name: " + name);
}
```

7. Database Metadata

• Theory:

o What is DatabaseMetaData?

-> DatabaseMetaData is an interface in Java's JDBC (Java Database Connectivity) API that provides comprehensive information about the database's structure and properties. It allows developers to retrieve metadata, such as details about the database's schema, tables, stored procedures, capabilities, and much more. Essentially, it acts as a bridge between the application and the database, helping applications understand the database's characteristics.

o Importance of Database Metadata in JDBC

- Database Independence: Supports cross-database compatibility.
- Runtime Insights: Retrieves database-specific information at runtime.
- Debugging and Maintenance: Aids in troubleshooting and understanding database configurations.
- Dynamic Queries: Enables creation of queries based on schema metadata.

o Methods provided by DatabaseMetaData (getDatabaseProductName, getTables, etc.)

- getDatabaseProductName(): Returns the database product name (e.g., MySQL, Oracle).
- getTables(): Provides information about tables in the database.
- getColumns(): Retrieves column details of a specified table.
- getPrimaryKeys(): Fetches primary key information.
- getIndexInfo(): Supplies index details for a table.

8. ResultSet Metadata

• Theory:

o What is ResultSetMetaData?

-> An interface in JDBC used to retrieve metadata about the columns in the ResultSet object of a SQL query.

o Importance of ResultSet Metadata in analyzing the structure of query results

- Structure Analysis: Helps analyze the structure and properties of query results, such as column names, types, and count.

- Dynamic Query Handling: Facilitates applications to handle query results dynamically without prior knowledge of the database schema.
- Error Prevention: Assists in verifying column details, reducing runtime errors and mismatched data types.

o Methods in ResultSetMetaData (getColumnCount, getColumnName, getColumnType)

- getColumnCount(): Returns the number of columns in the ResultSet.
- getColumnName(int column): Retrieves the name of a specified column.
- getColumnType(int column): Provides the SQL data type of a given column.

10.

1: Swing GUI for CRUD Operations

• Theory:

o Introduction to Java Swing for GUI development

-> Swing is a GUI widget toolkit in Java that allows you to create windows, buttons, text fields, and other interactive components. It is part of the Java Foundation Classes (JFC).

->Swing is platform-independent and written entirely in Java, unlike AWT which relies on native code.

Key Features of Swing

- Lightweight components: Don't rely on native OS GUI components.
- Pluggable Look and Feel: Customize the appearance of your application.
- Rich Set of Components: Includes tables, trees, tabbed panes, sliders, etc.
- MVC Architecture: Separates model (data), view (UI), and controller (interaction logic).

o How to integrate Swing components with JDBC for CRUD operations

- Java JDK installed
- A database (e.g., MySQL, SQLite, or PostgreSQL)
- A JDBC driver for your database
- IDE (e.g., IntelliJ, Eclipse, or NetBeans)

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.sql.*;
```

```
public class UserManager extends JFrame {
    JTextField nameField, emailField;
    JButton addButton, loadButton;
    JTextArea output;

    public UserManager() {
        setTitle("User Manager");
        setLayout(new FlowLayout());
        setSize(400, 300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        nameField = new JTextField(15);
        emailField = new JTextField(15);
        addButton = new JButton("Add User");
        loadButton = new JButton("Load Users");
        output = new JTextArea(10, 30);

        add(new JLabel("Name:"));
        add(nameField);
        add(new JLabel("Email:"));
        add(emailField);
        add(addButton);
        add(loadButton);
        add(new JScrollPane(output));

        // ADD user to DB
        addButton.addActionListener(e -> {
            try (Connection conn = DBConnection.getConnection()) {
                String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
                PreparedStatement stmt = conn.prepareStatement(sql);
```

```
            stmt.setString(1, nameField.getText());
            stmt.setString(2, emailField.getText());
            stmt.executeUpdate();
            JOptionPane.showMessageDialog(this, "User added!");
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    });

    // LOAD users from DB
    loadButton.addActionListener(e -> {
        try (Connection conn = DBConnection.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");
            output.setText("");
            while (rs.next()) {
                output.append(rs.getInt("id") + ": " +
                              rs.getString("name") + " - " +
                              rs.getString("email") + "\n");
            }
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    });

    setVisible(true);
}

public static void main(String[] args) {
    new UserManager();
```

```
    }
}
```