Java Module-2 Assignment
JDBC-RDBMS

Introduction to JDBC
Theory:

What is JDBC?

JDBC (Java Database Connectivity) is a standard Java API that enables Java applications to interact with databases. It provides a set of classes and interfaces to connect to a database, execute SQL queries, retrieve results, and manage database transactions in a **database-independent** manner.

Using JDBC, a Java program can:

- Connect to relational databases (MySQL, Oracle, PostgreSQL, etc.)
- Execute SQL statements (SELECT, INSERT, UPDATE, DELETE)
- Retrieve and process results
- Handle database transactions and exceptions

Importance of JDBC in Java Programming

JDBC plays a vital role in Java-based applications because:

1. **Database Independence**
   Java applications can work with different databases by simply changing the JDBC driver.
2. **Standard API**
   JDBC provides a uniform interface for database access, making development easier and consistent.
3. **Enterprise Application Support**
   Widely used in enterprise applications such as banking, e-commerce, and management systems.

4. **Secure Data Access**
   Supports prepared statements, which help prevent SQL injection.

5. **Transaction Management**
   Allows control over transactions using commit and rollback operations.

JDBC Architecture

JDBC follows a **layered architecture** that connects a Java application to a database.

1. Driver Manager

- Acts as a **controller** between the Java application and JDBC drivers.
- Loads and manages database drivers.
- Establishes a connection with the database using a connection URL.

Example:

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

---

2. JDBC Driver

- A software component that enables Java applications to communicate with the database.
- Converts JDBC calls into database-specific calls.
- Types of JDBC drivers:
  - **Type 1:** JDBC-ODBC Bridge
  - **Type 2:** Native API Driver
  - **Type 3:** Network Protocol Driver
  - **Type 4:** Thin Driver (Pure Java, most commonly used)

---

3. Connection

- Represents a session between the Java application and the database.
- Used to create `Statement`, `PreparedStatement`, or `CallableStatement` objects.
- Manages transactions.

Example:

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

---

4. Statement

- Used to execute SQL queries.
- Types of statements:
  - **Statement:** Simple SQL queries
  - **PreparedStatement:** Precompiled SQL queries (more secure and efficient)
  - **CallableStatement:** Used to call stored procedures

Example:

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

---

5. ResultSet

- Stores the result of a SELECT query.
- Allows navigation through database records.
- Data can be accessed using column names or indexes.

Example:

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

Summary

- JDBC enables Java programs to interact with databases.
- It provides database independence and a standard way to execute SQL.
- Core components of JDBC architecture include:
  - **Driver Manager**
  - **Driver**
  - **Connection**
  - **Statement**
  - **ResultSet**

Overview of JDBC Driver Types

JDBC drivers act as a bridge between a Java application and a database. Based on how they communicate with the database, JDBC drivers are classified into **four types**.

Type 1: JDBC–ODBC Bridge Driver

Overview:

- Translates JDBC calls into ODBC calls.
- Requires an ODBC driver to be installed on the client machine.
- Uses the JDBC-ODBC Bridge.

Architecture:

```
Java Application → JDBC → ODBC → Database
```

Advantages:

- Easy to use for learning and testing.
- No need to write native code.

Disadvantages:

- Slow performance.
- Platform dependent.
- Requires ODBC configuration.
- **Deprecated and removed in Java 8**.

Usage:

- Used only for **educational purposes**.
- Not recommended for real-world applications.

Type 2: Native-API Driver

Overview:

- Converts JDBC calls into database-specific native calls.
- Requires database native libraries on the client machine.

Architecture:

```
Java Application → JDBC → Native API → Database
```

Advantages:

- Better performance than Type 1.
- Direct access to database features.

Disadvantages:

- Platform dependent.
- Requires native libraries installation.
- Less portable.

Usage:

- Used in **intranet applications** where client environment is controlled.

Type 3: Network Protocol Driver

Overview:

- JDBC calls are sent to a **middleware server**.
- Middleware converts calls to database-specific protocols.

Architecture:

```
Java Application → JDBC → Middleware Server → Database
```

Advantages:

- No database-specific code on client side.
- Platform independent.

- Can connect to multiple databases.

Disadvantages:

- Requires middleware setup.
- Slower than Type 4 due to extra network layer.

Usage:

- Suitable for **enterprise applications** using multiple databases.

Type 4: Thin Driver

Overview:

- Converts JDBC calls directly into database protocol.
- Written entirely in Java (pure Java driver).

Architecture:

```
Java Application → JDBC → Database
```

Advantages:

- Best performance.
- Platform independent.
- No additional software required.
- Secure and easy to deploy.

Disadvantages:

- Database-specific driver required.

Usage:

- **Most widely used** driver in real-world applications.
- Preferred for web and enterprise applications.

Comparison of JDBC Driver Types

Feature	Type 1	Type 2	Type 3	Type 4
Platform Independent	❌	❌	✅	✅
Performance	Low	Medium	Medium	High
Middleware Required	❌	❌	✅	❌
Native Code Required	❌	✅	❌	❌
Ease of Deployment	Low	Medium	Medium	High
Java Version Support	≤ Java 7	All	All	All
Real-world Usage	Rare	Limited	Limited	Very High

Conclusion

- **Type 1** is obsolete and used only for learning.
- **Type 2** is faster but platform dependent.
- **Type 3** is useful for multi-database enterprise systems.
- **Type 4** is the **best and most commonly used JDBC driver** due to high performance and portability.

Step-by-Step Process to Establish a JDBC Connection

To interact with a database using JDBC, a Java program follows a well-defined sequence of steps. Each step plays an important role in ensuring proper database connectivity and data handling.

---

1. Import the JDBC Packages

- Required JDBC classes and interfaces are available in `java.sql` and `javax.sql`.
- These packages provide classes like `Connection`, `Statement`, `ResultSet`, etc.

Example:

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

---

2. Register the JDBC Driver

- The JDBC driver must be loaded so that the `DriverManager` can use it.
- In modern Java versions, drivers are automatically loaded.
- Explicit registration can be done using `Class.forName()`.

Example:

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

---

3. Open a Connection to the Database

- A connection is established using `DriverManager.getConnection()`.
- Requires database URL, username, and password.

Example:

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

---

4. Create a Statement

- A `Statement` object is created to send SQL commands to the database.
- Can be:
  - `Statement`
  - `PreparedStatement`
  - `CallableStatement`

Example:

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

---

5. Execute SQL Queries

- SQL commands are executed using statement methods:
    - `executeQuery()` for SELECT
    - `executeUpdate()` for INSERT, UPDATE, DELETE
    - `execute()` for general SQL

Example:

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

---

6. Process the Result Set

- The `ResultSet` object stores the data returned by the query.
- Use `next()` to iterate through records.

Example:

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

---

7. Close the Connection

- All JDBC resources should be closed to avoid memory leaks.

- Close in reverse order: `ResultSet`, `Statement`, then `Connection`.

Example:

```
rs.close();

stmt.close();

con.close();
```

Complete Example Program

```
import java.sql.*;


public class JDBCExample {

    public static void main(String[] args) {

        try {

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


            Connection con = DriverManager.getConnection(

                "jdbc:mysql://localhost:3306/college", "root",
"password");


            Statement stmt = con.createStatement();

            ResultSet rs = stmt.executeQuery("SELECT * FROM student");
            while (rs.next()) {

                System.out.println(rs.getInt(1) + " " +
rs.getString(2));

            }


            rs.close();

            stmt.close();

            con.close();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}
```

Summary

1. Import JDBC packages
2. Register the JDBC driver
3. Open database connection
4. Create a statement
5. Execute SQL query
6. Process results
7. Close the connection

Overview of JDBC Statements

In JDBC, **Statement objects** are used to send SQL commands from a Java program to a database. JDBC provides **three types of Statement interfaces**, each designed for different use cases.

---

1. Statement

Overview:

- Used to execute **simple SQL queries** without parameters.
- SQL queries are sent directly to the database at runtime.
- Best suited for static SQL statements.

Key Features:

- Easy to use
- Less efficient for repeated execution
- Vulnerable to SQL Injection attacks

Example:

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

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

Usage:

- Suitable for **simple, one-time queries**
- Not recommended when user input is involved

---

2. PreparedStatement

Overview:

- Used for **precompiled SQL queries** with parameters.
- SQL query is compiled once and can be executed multiple times.
- Uses placeholders (`?`) for parameters.

Key Features:

- Faster performance for repeated queries
- Prevents SQL Injection
- Easy parameter handling

Example:

```
PreparedStatement ps =
    con.prepareStatement("INSERT INTO student VALUES (?, ?)");


ps.setInt(1, 101);
ps.setString(2, "Jatin");
ps.executeUpdate();
```

Usage:

- Recommended for **dynamic queries**
- Best choice for most database operations

3. CallableStatement

Overview:

- Used to **call stored procedures** in the database.
- Can handle IN, OUT, and INOUT parameters.
- Useful for complex business logic stored at the database level.

Key Features:

- Improves performance for complex operations
- Supports database-side logic
- Can return multiple result sets

Example:

```
CallableStatement cs =

    con.prepareCall("{call getStudent(?)}");


cs.setInt(1, 101);

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

Usage:

- Used in **enterprise applications**
- Ideal when using stored procedures

Comparison of JDBC Statements

Feature	Statement	PreparedStatement	CallableStatement
Parameters	❌ No	✅ Yes	✅ Yes
Precompiled	❌ No	✅ Yes	✅ Yes
Performance	Low	High	High
SQL Injection Safe	❌ No	✅ Yes	✅ Yes
Stored Procedure Support	❌ No	❌ No	✅ Yes
Reusability	Low	High	High

Conclusion

- Use **Statement** for simple and static SQL queries.
- Use **PreparedStatement** for parameterized and secure database access.
- Use **CallableStatement** for executing stored procedures.

Database Operations in JDBC (CRUD Operations)

In JDBC, database interaction is mainly performed using **CRUD operations**. CRUD stands for **Create, Read, Update, and Delete**, which correspond to SQL commands **INSERT, SELECT, UPDATE, and DELETE**.

1. INSERT – Adding a New Record

Theory:

- The `INSERT` statement is used to add new records into a database table.
- It increases the number of rows in a table.

Syntax:

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

JDBC Example:

```
PreparedStatement ps =

    con.prepareStatement("INSERT INTO student VALUES (?, ?)");


ps.setInt(1, 101);

ps.setString(2, "Jatin");

ps.executeUpdate();
```

Key Point:
`executeUpdate()` returns the number of rows affected.

---

2. UPDATE – Modifying Existing Records

Theory:

- The `UPDATE` statement is used to change existing records in a table.

- Usually combined with a `WHERE` clause to specify rows.

Syntax:

```
UPDATE table_name SET column=value WHERE condition;
```

JDBC Example:

```
PreparedStatement ps =

    con.prepareStatement("UPDATE student SET name=? WHERE id=?");


ps.setString(1, "Rahul");

ps.setInt(2, 101);

ps.executeUpdate();
```

Key Point:
Without a `WHERE` clause, **all records will be updated**.

---

3. SELECT – Retrieving Records

Theory:

- The `SELECT` statement retrieves data from the database.
- Results are stored in a `ResultSet` object.

Syntax:

```
SELECT * FROM table_name;
```

JDBC Example:

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

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


while (rs.next()) {

    System.out.println(rs.getInt("id") + " " + rs.getString("name"));

}
```

Key Point:
`executeQuery()` is used only for `SELECT` statements.

---

4. DELETE – Removing Records

Theory:

- The `DELETE` statement removes records from a table.
- Should be used carefully with a `WHERE` clause.

Syntax:

```
DELETE FROM table_name WHERE condition;
```

JDBC Example:

```
PreparedStatement ps =

    con.prepareStatement("DELETE FROM student WHERE id=?");


ps.setInt(1, 101);

ps.executeUpdate();
```

Key Point:
Without a `WHERE` clause, **all records will be deleted**.

Summary Table

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

ResultSet in JDBC

What is ResultSet in JDBC?

A **ResultSet** is an object in JDBC that stores the data retrieved from a database after executing a **SELECT** query. It represents a table of data where each row corresponds to a database record, and each column corresponds to a field in the table.

- It is obtained by executing `executeQuery()` on a `Statement` or `PreparedStatement`.
- A `ResultSet` maintains a **cursor** that points to the current row.

- Initially, the cursor is positioned **before the first row**.

Example:

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

Navigating Through ResultSet

The cursor movement depends on the type of `ResultSet`. Common navigation methods include:

1. next()

- Moves the cursor to the **next row**.
- Most commonly used method.

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

---

2. first()

- Moves the cursor to the **first row**.
- Works only with **scrollable ResultSet**.

```
rs.first();
```

---

3. last()

- Moves the cursor to the **last row**.
- Useful for counting records.

```
rs.last();
```

---

4. previous()

- Moves the cursor to the **previous row**.
- Requires a scrollable ResultSet.

```
rs.previous();
```

Creating a Scrollable ResultSet

By default, ResultSet is **forward-only**. To enable full navigation:

```
Statement stmt = con.createStatement(

    ResultSet.TYPE_SCROLL_INSENSITIVE,

    ResultSet.CONCUR_READ_ONLY);


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

Working with ResultSet to Retrieve Data

Retrieving Data by Column Name

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

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

Retrieving Data by Column Index

```
int id = rs.getInt(1);

String name = rs.getString(2);
```

Complete Example

```
Statement stmt = con.createStatement(

    ResultSet.TYPE_SCROLL_INSENSITIVE,

    ResultSet.CONCUR_READ_ONLY);


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


// Move to first row

rs.first();

System.out.println(rs.getInt(1) + " " + rs.getString(2));


// Move to last row

rs.last();

System.out.println(rs.getInt(1) + " " + rs.getString(2));
```

```
// Iterate forward

rs.beforeFirst();

while (rs.next()) {

    System.out.println(rs.getInt(1) + " " + rs.getString(2));

}
```

DatabaseMetaData in JDBC

What is DatabaseMetaData?

DatabaseMetaData is an interface in JDBC that provides information about the **database itself**, not the data stored in tables. It describes the database product, version, supported features, schemas, tables, columns, SQL syntax support, and driver capabilities.

- Obtained from a `Connection` object.
- Helps Java applications understand the **structure and capabilities** of the connected database.

Example:

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

Importance of Database Metadata in JDBC

Database metadata is important because:

1. **Database Independence**
   Allows applications to adapt dynamically to different database products.
2. **Schema Exploration**
   Enables retrieval of table names, column names, primary keys, and indexes.
3. **Feature Detection**
   Helps check support for features like transactions, batch updates, and stored procedures.
4. **Tool Development**
   Useful for building database tools, ORM frameworks, and admin utilities.
5. **Runtime Analysis**
   Allows programs to inspect database details at runtime without hardcoding values.

Methods Provided by DatabaseMetaData

1. getDatabaseProductName()

- Returns the name of the database product.

```
String dbName = dbmd.getDatabaseProductName();

System.out.println("Database Name: " + dbName);
```

---

2. getDatabaseProductVersion()

- Returns the database version.

```
System.out.println(dbmd.getDatabaseProductVersion());
```

---

3. getDriverName()

- Returns the JDBC driver name.

```
System.out.println(dbmd.getDriverName());
```

---

4. getDriverVersion()

- Returns the JDBC driver version.

```
System.out.println(dbmd.getDriverVersion());
```

---

5. getTables()

- Retrieves information about tables in the database.
- Returns a `ResultSet`.

```
ResultSet rs = dbmd.getTables(null, null, "%", new String[]{"TABLE"});
while (rs.next()) {
    System.out.println(rs.getString("TABLE_NAME"));
}
```

---

6. getColumns()

- Retrieves information about columns of a table.

```
ResultSet rs = dbmd.getColumns(null, null, "student", "%");

while (rs.next()) {

    System.out.println(

        rs.getString("COLUMN_NAME") + " " +

        rs.getString("TYPE_NAME"));

}
```

---

7. supportsTransactions()

- Checks whether the database supports transactions.

```
boolean support = dbmd.supportsTransactions();

System.out.println("Supports Transactions: " + support);
```

---

8. supportsBatchUpdates()

- Checks batch update support.

```
System.out.println(dbmd.supportsBatchUpdates());
```

ResultSetMetaData in JDBC

What is ResultSetMetaData?

ResultSetMetaData is an interface in JDBC that provides information about the **structure of data returned by a SQL query**.
It describes the **columns** in a `ResultSet`, such as:

- Number of columns
- Column names
- Data types
- Column size
- Whether a column allows NULL values

It is obtained from a `ResultSet` object.

Example:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

Importance of ResultSet Metadata in JDBC

ResultSet metadata is important because it allows programs to **analyze query results dynamically**, without knowing the table structure in advance.

Key Benefits:

1. **Dynamic Data Handling**

- Applications can process query results without hardcoding column names or counts.

2. **Generic Report Generation**

- Useful for creating reports, tables, and export tools (CSV, Excel).

3. **Database Tool Development**

- Used in database browsers, admin tools, and ORM frameworks.

4. **Improved Flexibility**

- Same code can handle results from different queries and tables.

5. **Runtime Column Analysis**

- Helps validate column properties such as data type and nullability.

Common Methods of ResultSetMetaData

1. getColumnCount()

- Returns the total number of columns.

```
int count = rsmd.getColumnCount();
```

---

2. getColumnName(int column)

- Returns the column name.

```
System.out.println(rsmd.getColumnName(1));
```

---

3. getColumnTypeName(int column)

- Returns the database-specific data type.

```
System.out.println(rsmd.getColumnTypeName(1));
```

---

4. getColumnLabel(int column)

- Returns the column alias (if used).

```
System.out.println(rsmd.getColumnLabel(1));
```

5. isNullable(int column)

- Checks whether the column allows NULL values.

```
System.out.println(rsmd.isNullable(1));
```

Example: Using ResultSetMetaData

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

int cols = rsmd.getColumnCount();
for (int i = 1; i <= cols; i++) {
    System.out.println(
        rsmd.getColumnName(i) + " - " +
        rsmd.getColumnTypeName(i));
}
```

Difference Between DatabaseMetaData and ResultSetMetaData

Feature	DatabaseMetaData	ResultSetMetaData
Describes	Database & driver	Query result structure
Obtained From	Connection	ResultSet
Scope	Entire database	Single query result
Use Case	Schema & capability analysis	Dynamic result processing

SQL Queries and Their Implementation in Java Using JDBC

Below are the required **SQL queries** along with their **JDBC implementations**.
Assume a table:

```
student(id INT PRIMARY KEY, name VARCHAR(50), marks INT)
```

---

1. Inserting a Record into a Table

SQL Query

```
INSERT INTO student (id, name, marks) VALUES (101, 'Jatin', 85);
```

JDBC Implementation

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

ps.setInt(1, 101);
ps.setString(2, "Jatin");
ps.setInt(3, 85);

ps.executeUpdate();
System.out.println("Record inserted successfully");
```

---

2. Updating Specific Fields of a Record

SQL Query

```
UPDATE student SET marks = 90 WHERE id = 101;
```

JDBC Implementation

```
PreparedStatement ps =
    con.prepareStatement("UPDATE student SET marks=? WHERE id=?");

ps.setInt(1, 90);
ps.setInt(2, 101);

ps.executeUpdate();
System.out.println("Record updated successfully");
```

---

3. Selecting Records Based on Certain Conditions

SQL Query

```
SELECT * FROM student WHERE marks > 80;
```

JDBC Implementation

```
PreparedStatement ps =
    con.prepareStatement("SELECT * FROM student WHERE marks > ?");

ps.setInt(1, 80);
ResultSet rs = ps.executeQuery();

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

---

4. Deleting Specific Records

SQL Query

```
DELETE FROM student WHERE id = 101;
```

JDBC Implementation

```
PreparedStatement ps =
    con.prepareStatement("DELETE FROM student WHERE id=?");

ps.setInt(1, 101);
ps.executeUpdate();

System.out.println("Record deleted successfully");
```

Complete JDBC Example (CRUD Operations)

```
import java.sql.*;

public class CRUDExample {
    public static void main(String[] args) {
        try {
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/college", "root",
"password");

            // INSERT
            PreparedStatement ps1 =
                con.prepareStatement("INSERT INTO student VALUES (?,
?, ?)");
            ps1.setInt(1, 102);
            ps1.setString(2, "Rahul");
            ps1.setInt(3, 88);
            ps1.executeUpdate();

            // SELECT
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM student");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " +
                                   rs.getString(2) + " " +
                                   rs.getInt(3));
            }

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

Summary Table

Operation	**SQL Command**	**JDBC Method**
Insert	INSERT	executeUpdate()
Update	UPDATE	executeUpdate()
Select	SELECT	executeQuery()
Delete	DELETE	executeUpdate()
