Module 2 – Java– RDBMS & Database Programming with JDBC

1.Introduction to JDBC

1.1 What is JDBC(JavaDatabase Connectivity)?

JDBC is a standard Java API used to connect Java applications with relational databases. It provides classes and interfaces to perform database operations such as inserting, updating, deleting, and retrieving data using SQL. JDBC makes Java programs database-independent by using drivers.

1.2 Importance of JDBC in Java Programming.

JDBC is important because it allows Java applications to store and manage data permanently in databases instead of using files. It supports multiple databases (MySQL, Oracle, PostgreSQL, etc.) without changing application logic. JDBC also ensures secure, reliable, and efficient data access in enterprise-level applications.

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

JDBC Architecture :

1. **DriverManager**
   DriverManager manages JDBC drivers and establishes a connection between the Java application and the database. It selects the appropriate driver based on the database URL.

2. **Driver**
   A JDBC Driver is a software component that converts Java JDBC calls into database-specific calls. Different databases require different drivers, such as MySQL Driver or Oracle Driver.

3. **Connection**
   Connection represents a session between the Java application and the database. It is used to create Statement objects and manage transactions.

4. **Statement**
   Statement is used to send SQL queries to the database. It executes SQL commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.

5. **ResultSet**
   ResultSet stores the data returned by a `SELECT` query. It allows the program to read database records row by row in a structured manner.

2. JDBC Driver Types

2.1 Overview of JDBC Driver Types

Type 1: JDBC–ODBC Bridge Driver

This driver uses the ODBC driver to connect Java applications to the database. JDBC calls are translated into ODBC calls, and then sent to the database. It is easy to use but slower and not recommended for production because it depends on platform-specific ODBC.

Advantages:

- Easy to use and understand
- No need to write native database-specific code
- Useful for learning JDBC concepts

Disadvantages:

- Very slow performance due to multiple translations
- Requires ODBC driver installation on the client
- Platform dependent
- Not secure and not suitable for production
- Deprecated and removed from modern Java versions

Type 2: Native-API Driver

This driver converts JDBC calls into database-specific native calls using client-side libraries. It offers better performance than Type 1 but requires native database libraries to be installed on the client machine, making it platform dependent.

Advantages:

- Better performance than Type 1
- Direct interaction with database native APIs
- Suitable for intranet applications

Disadvantages:

- Platform dependent
- Requires native libraries on each client machine
- Difficult to maintain and deploy
- Not suitable for internet-based applications

Type 3: Network Protocol Driver

This driver sends JDBC calls to a middleware server using a network protocol. The middleware then translates the calls into database-specific calls. It is fully Java-based and supports multiple databases, but requires an additional middleware layer.

Advantages:

- Platform independent (pure Java)
- Can connect to multiple databases through one driver
- Centralized control through middleware
- Better security than Type 1 and Type 2

Disadvantages:

- Requires an additional middleware server
- Network overhead may reduce performance
- More complex architecture
- Higher maintenance cost

Type 4: Thin Driver

This driver directly converts JDBC calls into database-specific protocol calls. It is completely written in Java, provides the best performance, and is widely used in real-world applications (e.g., MySQL, Oracle drivers).

Advantages:

- Best performance among all drivers
- Platform independent (pure Java)
- No additional software required on client side
- Highly secure and reliable
- Easy to deploy and maintain

Disadvantages:

- Database-specific driver required
- Separate driver needed for each database type

2.2 Comparison and Usage of Each Driver Type

Driver Type	Performance	Platform Dependency	Usage
Type 1	Low	High (ODBC dependent)	Learning and testing only
Type 2	Medium	Yes (native libraries required)	Legacy Systems
Type 3	Good	No	Enterprise application with middleware
Type 4	High	No	Recommended for production use

3. Steps for Creating JDBC Connections

3.1 Step-by-Step Process to Establish a JDBC Connection

1. Import the JDBC Packages

To use JDBC classes like `Connection`, `Statement`, and `ResultSet`, import the `java.sql` package.

Ex...import java.sql.*;

2. Register the JDBC Driver

The JDBC driver is registered to make Java aware of the database driver.

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

3. Open a Connection to the Database

Create a connection using `DriverManager` with database URL, username, and password.

Ex... Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/testdb",

"root",

"password"

);

4. Create a Statement

A `Statement` object is used to send SQL queries to the database.

Ex... Statement stmt1 = con.createStatement();

5. Execute SQL Queries

Use `executeQuery()` for `SELECT` statements.

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

6. Process the Result Set

Retrieve data row by row from the `ResultSet`.

Ex... while (rs.next()) {

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

7. Close the Connection

Always close JDBC resources to free memory and avoid connection leaks.

Ex... rs.close();

stmt.close();

con.close();

4. Types of JDBC Statements

4.1 Overview of JDBC Statements

1. Statement

`Statement` is used to execute simple SQL queries without parameters. The SQL query is sent to the database and compiled every time it is executed.

Example:

```
Statement stmt1 = con.createStatement();

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

2. PreparedStatement

`PreparedStatement` is used for SQL queries with parameters. The query is precompiled once and can be executed multiple times with different values, improving performance and security.

Example:

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

ps.setInt(1, 101);

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

3. CallableStatement

`CallableStatement` is used to call stored procedures defined in the database. It allows both input and output parameters.

Example:

```
CallableStatement cs = con.prepareCall(
    "{call getStudentById(?)}"
);
cs.setInt(1, 101);
ResultSet rs = cs.executeQuery();
```

4.2 Difference between Statement,CallableStatement and PreparedStatement

Feature	Statement	Prepared Statement	Callable Statement
Query Type	Simple, static SQL	Parameterized SQL	Stored procedures
Compilation	Compiled every time	Precompiled once	Precompiled
Parameters	Not supported	Supported	Supported (IN/OUT)
Performance	Low	High	High
Security	Vulnerable to SQL Injection	Safe	Safe
Use Case	Simple queries	Repeated queries	Complex DB operations

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

5.1 INSERT – Adding a New Record to the Database

The `INSERT` operation is used to add new records into a database table. In JDBC, it is commonly performed using `PreparedStatement` to pass values safely.

Example:

```
PreparedStatement ps = con.prepareStatement(
    "INSERT INTO student(id, name, age) VALUES (?, ?, ?)"
);
ps.setInt(1, 101);
ps.setString(2, "Amit");
ps.setInt(3, 22);
ps.executeUpdate();
```

5.2 UPDATE – Modifying Existing Records

The `UPDATE` operation modifies existing records in a table based on a condition. It is used when data needs to be changed without deleting the record.

Example:

```
PreparedStatement ps = con.prepareStatement(
    "UPDATE student SET age = ? WHERE id = ?"
);
ps.setInt(1, 23);
ps.setInt(2, 101);
ps.executeUpdate();
```

5.3 SELECT – Retrieving Records from the Database

The `SELECT` operation retrieves data from the database. The result is stored in a `ResultSet`, which is processed row by row.

Example:

```
PreparedStatement ps = con.prepareStatement(
    "SELECT * FROM student"
);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.println(
        rs.getInt("id") + " " +
        rs.getString("name") + " " +
        rs.getInt("age")
    );
}
```

5.4 DELETE – Removing Records from the Database

The `DELETE` operation removes records from a table based on a condition. It should be used carefully to avoid accidental data loss.

Example:

```
PreparedStatement ps = con.prepareStatement(
    "DELETE FROM student WHERE id = ?"
);
ps.setInt(1, 101);
ps.executeUpdate();
```

6. ResultSet Interface

6.1 What is ResultSet in JDBC?

`ResultSet` is an interface in JDBC that represents the data returned from a `SELECT` SQL query. It stores records in a tabular form and allows the Java program to read data row by row from the database.

6.2 Navigating Through ResultSet

By default, a `ResultSet` points before the first row. JDBC provides navigation methods to move the cursor.

Methods:

next() : Moves to the next row

previous() : Moves to the previous row

first() : Moves to the first row

last() : Moves to the last row

To use `first()`, `last()`, and `previous()`, the `ResultSet` must be **scrollable**.

Creating a Scrollable ResultSet:

```
Statement stmt1 = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
ResultSet rs = stmt.executeQuery("SELECT * FROM student");
```

6.3 Working with ResultSet to Retrieve Data

In JDBC, the `ResultSet` interface is used to retrieve and process data returned by a `SELECT` SQL query. The data is stored in tabular form, and a cursor is used to move through the rows.

Steps to Retrieve Data Using ResultSet

1. **Execute SELECT Query**
   A `SELECT` query is executed using `Statement` or `PreparedStatement`, which returns a `ResultSet` object.

2. **Move the Cursor**
   The cursor initially points before the first row. Use `next()` to move to each row.

3. **Read Column Values**
   Data is retrieved using getter methods like `getInt()`, `getString()`, etc., by column name or index.

Example:

```
PreparedStatement ps = con.prepareStatement(
    "SELECT id, name, age FROM student"
);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println(id + " " + name + " " + age);
}
```

7. Database Metadata

7.1 What is DatabaseMetaData?

`DatabaseMetaData` is an interface in JDBC that provides information about the database, database driver, tables, columns, and supported features. It helps applications understand the structure and capabilities of the connected database.

7.2 Importance of DatabaseMetaData in JDBC

- ➢ It allows developers to get database details without writing SQL queries
- ➢ Helps in building database-independent applications
- ➢ Useful for tools, frameworks, and dynamic applications
- ➢ Provides information about tables, columns, and database limits

7.3 Methods Provided by DatabaseMetaData

- **getDatabaseProductName():**

  Returns the database name (e.g., MySQL, Oracle)

- **getDatabaseProductVersion()**

  Returns database version

- **getDriverName()**

  Returns JDBC driver name

- **getDriverVersion()**

  Returns JDBC driver version

- **getTables()**

  Returns information about tables in the database

- **getColumns()**

  Returns information about columns in a table

- **supportsTransactions()**

  Checks if transactions are supported

Example:

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb", "root", "password"
);
DatabaseMetaData dbmd = con.getMetaData();
System.out.println("Database Name: " + dbmd.getDatabaseProductName());
System.out.println(“Database version: ”+dbmd.getDatabaseProductVersion());
System.out.println("Driver Name: " + dbmd.getDriverName());
```

8. ResultSetMetaData

8.1 What is ResultSetMetaData?

`ResultSetMetaData` is an interface in JDBC that provides information about the structure of the data returned by a SQL query. It gives details about columns such as their count, names, data types, and sizes.

8.2 Importance of ResultSetMetaData

- Helps to analyze query results without knowing table structure in advance
- Useful in dynamic applications and reporting tools
- Allows developers to write generic code for processing query results
- Widely used in frameworks like ORM tools and database utilities

8.3 Methods in ResultSetMetaData

- **getColumnCount()**

  Returns total number of columns

- **getColumnName(int index)**

  Returns column name

- **getColumnType(int index)**

  Returns SQL data type of column

- **getColumnTypeName(int index)**

  Returns database-specific type

- **isNullable(int index)**

  Checks whether column allows NULL

- **getColumnDisplaySize(int index)**

  Returns column display width

Example:

```
PreparedStatement ps = con.prepareStatement(
   "SELECT * FROM student"
);
ResultSet rs = ps.executeQuery();
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 Name: " + rsmd.getColumnName(i) +
      ", Column Type: " + rsmd.getColumnTypeName(i)
   );
}
```

9.Callable Statement with In and out parameters

9.1 What is a CallableStatement?

`CallableStatement` is a JDBC interface used to execute **stored procedures** defined in the database. It allows Java applications to call database procedures and handle **IN, OUT, and INOUT parameters**.

9.2 How to Call Stored Procedures Using CallableStatement

Syntax:

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

Steps:

Create a `CallableStatement` using `prepareCall()`

Set input parameters using `setXXX()` methods

Register output parameters using `registerOutParameter()`

Execute the stored procedure

9.3 Working with IN and OUT Parameters

Example Stored Procedure (MySQL):

```
CREATE PROCEDURE getStudentName(
    IN sid INT,
    OUT sname VARCHAR(50)
)
```

```
BEGIN
   SELECT name INTO sname FROM student WHERE id = sid;
END;
```

CallableStatement Example in Java:

```
CallableStatement cs = con.prepareCall(
   "{call getStudentName(?, ?)}"
);

// IN parameter
cs.setInt(1, 101);

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

// Execute procedure
cs.execute();

// Get OUT parameter value
String name = cs.getString(2);
System.out.println("Student Name: " + name);
```