**Java–RDBMS & Database Programming**

**with JDBC**

Introduction to JDBC

**1. What is JDBC (Java Database Connectivity)?**

JDBC is an **API (Application Programming Interface)** in Java that allows Java programs to interact with relational databases. It enables Java applications to perform operations like connecting to a database, executing SQL queries, and retrieving results.

**Example**: Using JDBC, a Java program can connect to a MySQL database and retrieve employee data using SQL.

**2. Importance of JDBC in Java Programming**

* **Platform Independence**: JDBC provides a standard interface, allowing Java applications to work with different databases (MySQL, Oracle, PostgreSQL, etc.) without changing the application code.
* **Seamless Integration**: JDBC bridges the gap between Java applications and relational databases.
* **Secure and Scalable**: Supports secure transactions and large-scale database operations.
* **Widely Used**: Core technology for enterprise applications, web applications (with JSP/Servlets), and backend services.

**3. JDBC Architecture**

The JDBC architecture follows a layered approach involving the following key components:

1. **DriverManager**
   * Manages a list of database drivers.
   * Establishes a connection between a Java application and a database.
   * Example: Connection con = DriverManager.getConnection(...)
2. **Driver**
   * Interface implemented by database vendors (e.g., MySQL, Oracle).
   * Translates JDBC calls into database-specific calls.
3. **Connection**
   * Represents an active connection to a database.
   * Used to create Statements and manage transactions.
4. **Statement**
   * Used to execute SQL queries.
   * Types:
     + Statement: For static SQL.
     + PreparedStatement: For parameterized SQL.
     + CallableStatement: For calling stored procedures.
5. **ResultSet**
   * Represents the result of a query.
   * Allows iteration over the result rows using next().

JDBC Driver Types

**Type 1: JDBC-ODBC Bridge Driver**

* **Description**: Uses ODBC (Open Database Connectivity) driver to connect to the database.
* **Working**: Java → JDBC API → JDBC-ODBC Bridge → ODBC Driver → Database
* **Advantages**:
  + Easy to use for testing.
* **Disadvantages**:
  + Platform-dependent.
  + Requires ODBC drivers to be installed.
  + Slower and deprecated in Java 8+.
* **Usage**: Legacy systems, not recommended for production.

**Type 2: Native-API Driver**

* **Description**: Converts JDBC calls into database-specific native API calls using C/C++ libraries.
* **Working**: Java → JDBC API → Native API (C/C++) → Database
* **Advantages**:
  + Better performance than Type 1.
* **Disadvantages**:
  + Requires native database libraries to be installed on the client.
  + Platform-dependent.
* **Usage**: Used in client-server applications where performance is critical.

**Type 3: Network Protocol Driver**

* **Description**: JDBC calls are translated into a database-independent network protocol which is then translated to DB-specific calls by a middleware server.
* **Working**: Java → JDBC API → Middleware Server → Database
* **Advantages**:
  + No native library on client.
  + Can connect to multiple databases via a single middleware.
* **Disadvantages**:
  + Requires a dedicated middleware.
  + Complex architecture.
* **Usage**: Web-based enterprise applications with central middleware.

**Type 4: Thin Driver (Pure Java Driver)**

* **Description**: Directly communicates with the database using database-specific protocol; implemented entirely in Java.
* **Working**: Java → JDBC API → Database Protocol → Database
* **Advantages**:
  + Platform-independent.
  + No native libraries required.
  + High performance.
* **Disadvantages**:
  + Database-specific (a different driver is needed for each DBMS).
* **Usage**: Most commonly used in modern applications; ideal for web and enterprise applications.

Steps for Creating JDBC Connections

Step-by-Step Process to Establish a JDBC Connection in Java

**1. Import the JDBC Packages**

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

**Connection** :- Establishes a connection to the database.

**DriverManager :-** Manages database drivers.

**Statement :-** Executes SQL queries.

**ResultSet :-** Stores the query results.

**SQLException :-** Handles database errors.

**2. Register the JDBC Driver**

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

System.out.println("JDBC Driver Registered Successfully!");

**3. Open a Connection to the Database**

conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/java\_2", "root","");

**4. Create a Statement**

Connection conn=UserUtil.createConnection();

**5. Execute SQL Queries**

String sql="Select \* from cart where uid=? and payment\_status=?";

PreparedStatement pst=conn.prepareStatement(sql);

pst.setInt(1, uid);

pst.setBoolean(2, flag);

ResultSet rs=pst.executeQuery();

**6. Process the ResultSet**

while (rs.next()) {

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

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

String position = rs.getString("position");

System.out.println("ID: " + id + ", Name: " + name + ", Position: " + position);

}

**7. Close the Connection**

rs.close();

conn.close();

**Types of JDBC Statements**

**1. Statement (Simple SQL Queries Without Parameters)**

**Definition:**

The Statement interface is used to execute simple SQL queries (e.g., SELECT, INSERT, UPDATE, DELETE).

It does not accept input parameters.

**How It Works:**

A SQL query is written as a string and passed to the Statement object.

The database executes the query and returns a ResultSet (for SELECT queries).

**Syntax:**

PreparedStatement pst=conn.prepareStatement(sql);

pst.setInt(1, uid);

pst.setBoolean(2, flag);

ResultSet rs=pst.executeQuery();

**Methods in Statement Interface:**

Method Description

executeQuery(String sql) Executes a SELECT statement and returns a ResultSet.

executeUpdate(String sql) Executes INSERT, UPDATE, or DELETE and returns the number of affected rows.

execute(String SQL) Used for DDL (Data Definition Language) queries (e.g., CREATE, DROP).

**2. Prepared Statement (Precompiled SQL Statements with Parameters)**

**Definition:**

Prepared Statement is used for executing SQL queries with parameters.

It precompiles the SQL statement, making execution faster and more secure than Statement.

Prevents SQL Injection attacks by using parameterized queries.

**How It Works:**

Instead of directly embedding values in the query, place placeholders (?) where values should go.

Use setInt(), setString(), etc., to safely insert values into placeholders.

**Syntax:**

PreparedStatement pstmt = conn.prepareStatement("SELECT \* FROM employees WHERE id = ?");

pstmt.setInt(1, 101);

ResultSet rs = pstmt.executeQuery();

**Methods in PreparedStatement Interface:**

Method Description

setInt(int parameterIndex, int value) Sets an integer parameter.

setString(int parameterIndex, String value) Sets a string parameter.

setDouble(int parameterIndex, double value) Sets a double parameter.

executeQuery() Executes a SELECT query.

executeUpdate() Executes INSERT, UPDATE, or DELETE queries.

**3. CallableStatement (Used to Call Stored Procedures)**

**Definition:**

CallableStatement is used to call stored procedures in a database.

Stored procedures are precompiled SQL code blocks stored in the database, improving performance.

**How It Works:**

Stored procedures reduce network traffic by executing multiple operations in a single call.

Supports IN (input), OUT (output), and INOUT (both input & output) parameters.

**Syntax:**

CallableStatement cstmt = conn.prepareCall("{CALL getEmployeeById(?)}");

cstmt.setInt(1, 101);

ResultSet rs = cstmt.executeQuery();

**Differences between Statement, PreparedStatement, and CallableStatement**

| **Feature** | **Statement** | **PreparedStatement** | **CallableStatement** |
| --- | --- | --- | --- |
| Query Type | Static SQL | Dynamic SQL with parameters | Stored procedure calls |
| Precompiled | No | Yes | Yes |
| Performance | Slower for repeated use | Faster for repeated queries | High (depends on stored procedure) |
| Security | Vulnerable to injection | Safe from SQL injection | Safe (if procedure is secure) |
| Syntax Complexity | Simple | Medium | Higher (uses SQL procedure syntax) |
| Use Case | Simple direct queries | Frequent, user-input based queries | Reusable business logic in DB |

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

**1. INSERT: Adding a New Record to the Database**

**Definition :**

The INSERT statement is used to add new rows to a table

**Syntax :**

INSERT INTO table\_name (column1, column2, column3) VALUES (value1, value2, value3);

**2. UPDATE: Modifying Existing Records**

**Definition :**

The UPDATE statement modifies existing records in a table.

**Syntax :**

UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;

**3. SELECT: Retrieving Records from the Database**

**Definition :**

The SELECT statement retrieves data from a table.

**Syntax :**

SELECT column1, column2 FROM table\_name WHERE condition;

**4. DELETE: Removing Records from the Database**

**Definition :**

The DELETE statement removes records from a table.

**Syntax :**

DELETE FROM table\_name WHERE condition;

ResultSet Interface

**What is ResultSet in JDBC?**

**Definition:**

ResultSet is an interface in JDBC that stores the result of a SQL query executed using Statement or PreparedStatement.

It acts as a cursor that moves through the retrieved records.

It is used for fetching and processing data from a database.

**Navigating Through ResultSet**

By default, the cursor moves forward-only. However, we can enable different navigation options using ResultSet.TYPE\_SCROLL\_INSENSITIVE or ResultSet.TYPE\_SCROLL\_SENSITIVE.

**Method Description**

next() Moves to the next row (default movement).

previous() Moves to the previous row.

first() Moves to the first row.

last() Moves to the last row.

absolute(int row) Moves to a specific row number.

relative(int row) Moves a specified number of rows forward/backward.

**Working with ResultSet to Retrieve Data**

How to Extract Data from ResultSet:

Use methods like getInt(), getString(), getDouble(), etc., to retrieve column values.

**Database Metadata**

**What is Database MetaData?**

**Definition:**

DatabaseMetaData is an interface in JDBC that provides information about the database.

It helps in retrieving details like database name, version, tables, columns, and supported SQL features.

It is obtained using the Connection object.

**Importance of Database Metadata in JDBC**

Helps understand the database structure without manually inspecting it.

Useful for dynamic applications that work with multiple databases.

Provides information about supported SQL features for compatibility checks.

Helps generate reports on tables, columns, primary keys, etc.

Assists in database migrations by retrieving schema details.

**Methods Provided by Database MetaData**

-Basic Information Methods

Method Description

getDatabaseProductName() Returns the database name (e.g., MySQL, PostgreSQL).

getDatabaseProductVersion() Returns the database version.

getDriverName() Returns the JDBC driver name.

getDriverVersion() Returns the JDBC driver version.

getUserName() Returns the username of the current connection.

**ResultSet Metadata**

**What is ResultSetMetaData?**

**Definition :**

ResultSetMetaData is an interface in JDBC that provides information about the structure of a ResultSet.

It helps in retrieving details such as number of columns, column names, data types, and size.

It is useful for dynamic applications where query results may vary.

**Importance of ResultSetMetaData in Analyzing Query Results**

Helps analyze query results dynamically without prior knowledge of table structure.

Useful in building dynamic reports and generic database tools.

Assists in data validation and transformation based on column data types.

Enables applications to work with multiple databases without schema dependency

**Methods in ResultSetMetaData**

Column Information Methods

**Method Description**

getColumnCount() Returns the number of columns in the ResultSet.

getColumnName(int column) Returns the name of a specific column.

getColumnType(int column) Returns the SQL type of a column.

getColumnTypeName(int column) Returns the name of the SQL type.

getColumnDisplaySize(int column) Returns the column size (in characters).

isNullable(int column) Checks if a column allows NULL values.

isAutoIncrement(int column) Checks if a column is auto-incremented.

**Swing GUI for CRUD Operations**

**Introduction to Java Swing for GUI Development**

* **Java Swing** is a part of Java’s standard library (javax.swing) used for building **graphical user interfaces (GUIs)**.
* Swing provides components like:
  + JFrame (main window)
  + JPanel, JButton, JTextField, JLabel, JTable etc.
* It's event-driven and allows you to design forms similar to desktop applications.

Callable Statement with IN and OUT Parameters

**What is a CallableStatement?**

* CallableStatement is a **JDBC interface** used to **call stored procedures** in a database.
* Stored procedures are SQL routines stored in the database that can be reused for tasks like insert, update, complex logic, etc.
* CallableStatement is part of the java.sql package.

**How to Call Stored Procedures Using CallableStatement**

**Syntax:**

CallableStatement cstmt = connection.prepareCall("{call procedure\_name(?, ?)}");

* Use ? as placeholders for parameters.
* Use execute() or executeQuery()/executeUpdate() depending on what the procedure returns.

**Working with IN and OUT Parameters**

Stored procedures can have:

* IN parameters: for input values
* OUT parameters: for returning values
* INOUT parameters: for both