Chapter 19: Database Connectivity (e.g., JDBC)

Introduction

In the realm of enterprise-level and data-driven applications, **database connectivity** plays a crucial role. Most applications today rely heavily on databases to store, retrieve, and manipulate data. Java, being a versatile programming language, provides a powerful API called **JDBC (Java Database Connectivity)** for establishing a seamless connection between Java applications and relational databases such as MySQL, PostgreSQL, Oracle, and others.

This chapter explores the architecture, components, and usage of JDBC. It covers various operations such as connecting to a database, executing queries, retrieving results, using prepared statements, and handling exceptions. The knowledge from this chapter is vital for building full-stack applications and integrating backend logic with persistent storage.

19.1 Overview of JDBC

What is JDBC?

JDBC (Java Database Connectivity) is an API in Java that allows applications to interact with a variety of databases using a standard set of interfaces and classes.

Key Features:

- Platform-independent
- Supports multiple RDBMS through drivers
- Enables execution of SQL queries from Java code
- Handles transactions and batch processing

19.2 JDBC Architecture

1. **Two-Tier Architecture**

- Java application communicates directly with the database using JDBC drivers.

2. **Three-Tier Architecture**

- Java application communicates with a middle-tier (like a servlet or application server), which in turn communicates with the database.

19.3 JDBC Drivers

Types of JDBC Drivers:

Type	Name	Description
1	JDBC-ODBC Bridge Driver	Uses ODBC driver (deprecated)
2	Native-API Driver	Converts JDBC calls into DB-specific API calls
3	Network Protocol Driver	Uses middleware server for database access
4	Thin Driver (Pure Java)	Directly converts JDBC calls into network protocol

Type 4 drivers are widely used in modern applications due to their efficiency and platform-independence.

19.4 Basic Steps in JDBC Programming

1. **Import the JDBC package**
2. **Load and register the driver**
3. **Establish a connection**
4. **Create a statement**
5. **Execute SQL queries**
6. **Process the results**
7. **Close the connection**

19.5 Connecting to a Database: Example

Let's connect to a **MySQL** database.

```
import java.sql.*;

public class DBConnect {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/college";
        String username = "root";
        String password = "admin";

        try {
            // Load Driver (optional from JDBC 4.0 onward)
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish Connection
            Connection con = DriverManager.getConnection(url, username,
password);
            System.out.println("Connected successfully!");
```

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

19.6 Executing SQL Statements

1. Statement Interface

Used to execute static SQL statements.

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

2. PreparedStatement Interface

Used for executing **parameterized queries** – safer and faster.

```
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM students WHERE
id = ?");
pstmt.setInt(1, 101);
ResultSet rs = pstmt.executeQuery();
```

3. CallableStatement Interface

Used for calling **stored procedures**.

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

19.7 ResultSet Interface

Used to process the results retrieved from the database.

Common Methods:

- `next()`
- `getInt(columnIndex/name)`
- `getString(columnIndex/name)`

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

19.8 Inserting, Updating, and Deleting Records

Insert Example:

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO students VALUES
(?, ?, ?)");
pstmt.setInt(1, 103);
pstmt.setString(2, "Aman");
pstmt.setString(3, "B.Tech");
int rows = pstmt.executeUpdate();
System.out.println(rows + " rows inserted.");
```

Update Example:

```
PreparedStatement pstmt = con.prepareStatement("UPDATE students SET name=?
WHERE id=?");
pstmt.setString(1, "Rahul");
pstmt.setInt(2, 101);
pstmt.executeUpdate();
```

Delete Example:

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

19.9 Handling Exceptions and Closing Resources

Always close:

- ResultSet
- Statement/PreparedStatement
- Connection

Try-with-resources is recommended:

```
try (Connection con = DriverManager.getConnection(...);
     PreparedStatement pstmt = con.prepareStatement(...)) {

     // execute query

} catch (SQLException e) {
    e.printStackTrace();
}
```

19.10 JDBC Transaction Management

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

try {
    pstmt1.executeUpdate();
    pstmt2.executeUpdate();
    con.commit(); // Commit transaction
} catch (SQLException e) {
    con.rollback(); // Rollback on error
}
```

19.11 Batch Processing in JDBC

Efficient for bulk updates/inserts:

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO students VALUES
(?, ?, ?)");
for (int i = 0; i < 100; i++) {
    pstmt.setInt(1, i);
    pstmt.setString(2, "Student" + i);
    pstmt.setString(3, "B.Tech");
    pstmt.addBatch();
}
pstmt.executeBatch();
```

19.12 Metadata in JDBC

Use `DatabaseMetaData` and `ResultSetMetaData` to get info about DB and result sets.

```
DatabaseMetaData dbmd = con.getMetaData();
System.out.println("DB Product: " + dbmd.getDatabaseProductName());

ResultSetMetaData rsmd = rs.getMetaData();
System.out.println("Column Count: " + rsmd.getColumnCount());
```

19.13 Best Practices in JDBC

- Use `PreparedStatement` to prevent SQL injection.
- Always close resources.
- Prefer connection pooling (using libraries like HikariCP in production).
- Handle exceptions with proper logging.
- Separate DB logic from business logic (DAO pattern).

Summary

This chapter provided a comprehensive understanding of **database connectivity in Java using JDBC**. From understanding JDBC architecture to performing CRUD operations, managing transactions, and ensuring safe database interaction, you are now equipped to build Java applications with integrated database functionality. JDBC remains a cornerstone for backend development, especially in traditional Java EE and Spring applications.

In the next chapter, we will explore how to integrate JDBC with **GUI and Web Applications** for real-world software systems.