MODULE II — JDBC (Java Database Connectivity)

1. What is JDBC?

JDBC (Java Database Connectivity) is a Java API (Application Programming Interface) that defines how a Java application can connect and interact with a database. It provides a standard set of classes and interfaces to perform database operations like inserting, updating, deleting, and selecting data.

Essentially, JDBC acts as a universal bridge between a Java application and a wide range of relational databases (like MySQL, Oracle, PostgreSQL, etc.), allowing developers to write database code once that works across different database systems.

In simple terms: JDBC is the standard way a Java program "talks" to a database to send and receive data.

---

2. Why Do We Need JDBC?

Before JDBC, the primary standard for database connectivity was **ODBC (Open Database Connectivity)**. However, ODBC was built for languages like C/C++ and presented several problems for Java:

- **Platform-Dependent:** ODBC required native code libraries, meaning the application was tied to a specific operating system.
- **Complex Setup:** It required manual configuration of an ODBC driver on the client machine.
- **Inefficient for Java:** The bridge from Java to native code added performance overhead.

To overcome these issues, Java introduced JDBC as a pure Java solution that is:

- **Portable:** Being 100% Java, it works on any operating system with a Java Virtual Machine (JVM).
- **Secure:** It avoids the risks associated with executing native code.
- **Simple:** It uses familiar Java syntax and a straightforward object model.

---

3. JDBC Architecture

The JDBC architecture consists of four main components that work together to connect an application to a database.

1. **JDBC API:** Provides the interfaces and classes for developers to use, such as Connection, Statement, and ResultSet. The developer writes code against this API.

2. **JDBC Driver Manager:** This is the backbone of the JDBC architecture. It manages a list of database drivers and matches a connection request from the application with the correct driver. A key responsibility is loading the driver and establishing a connection using a connection URL.

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

3. **JDBC Driver:** A software component that translates the standard JDBC calls from the application into the specific database protocol that the target database (e.g., MySQL, Oracle) understands. Each database has its own specific driver.
4. **Database:** The actual database system that stores and manages the data.

---

4. Types of JDBC Drivers

There are four main types of JDBC drivers.

Type	Name	Description	Example
Type 1	JDBC-ODBC Bridge Driver	Converts JDBC calls into ODBC calls and uses an ODBC driver to connect to the database. This driver is slow, platform-dependent, and now deprecated.	Deprecated
Type 2	Native API Driver	Converts JDBC calls into the database's native API calls. It requires vendor-specific native libraries on the client machine.	Oracle OCI
Type 3	Network Protocol Driver	Uses a middleware server that acts as a proxy between the client and the database. This driver is platform-independent but adds a layer of network latency.	IDS Server
Type 4	Thin Driver	A pure Java driver that converts JDBC calls directly into the database's network protocol. It is the most efficient, portable, and commonly used driver.	MySQL Connector/J

---

5. JDBC Packages

JDBC functionality is primarily split across two packages:

- **java.sql:** Contains the core API for basic JDBC functionality, including interfaces like Connection, Statement, PreparedStatement, and ResultSet.
- **javax.sql:** Provides an extension to the core API for more advanced, enterprise-level features. A key class is DataSource, which is often used for connection pooling (reusing database connections to improve performance).

---

6. Steps in the JDBC Process

Connecting a Java application to a database involves a standard sequence of steps.

Step 1: Import the JDBC package

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

Step 2: Load and Register the JDBC Driver

This step dynamically loads the driver class into memory so the DriverManager can use it.

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

Step 3: Establish a Connection

Use the DriverManager to create a Connection object, which represents a session with the database. The connection URL contains the protocol, database location, and database name.

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

Step 4: Create a Statement Object

The Statement object is used to send your SQL queries to the database.

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

Step 5: Execute the SQL Query

Run the query and retrieve the results into a ResultSet object, which holds the data in a table-like structure.

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

Step 6: Process the Results

Iterate through the ResultSet to access the data from each row.

```
while (rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
```

Step 7: Close the Connection

Always close the connection and other resources to release database locks and memory.

```
con.close();
```

---

7. JDBC Core Objects

Object	Description
DriverManager	Manages a list of registered drivers and is used to establish a connection to the database.
Connection	An interface that represents a single session between the Java application and the database.
Statement	Used for executing static (simple) SQL queries that do not have parameters.
PreparedStatement	Used for executing precompiled SQL queries with parameters. It's more efficient and secure (prevents SQL injection) than Statement.
CallableStatement	Used to execute stored procedures and functions in the database.
ResultSet	A table of data representing the results of a database query. It maintains a cursor pointing to the current row of data.

8. Example: Displaying a Student Table

```
import java.sql.*;

class JdbcExample {
    public static void main(String args[]) {
        try {
            // Step 1: Load Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Create Connection
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/test", "root", "pass");

            // Step 3: Create Statement
            Statement stmt = con.createStatement();

            // Step 4: Execute Query
            ResultSet rs = stmt.executeQuery("SELECT * FROM student");

            // Step 5: Process Result
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2));
            }

            // Step 6: Close Connection
            con.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

MODULE III — JAVA SERVLETS (Up to Lifecycle)

1. What is a Servlet?

A **Servlet** is a Java program that runs on a web server and is used to create dynamic web content. It extends the capabilities of a server by handling client requests and generating custom responses.

Servlets run inside a **Web Container** (also called a Servlet Container), which is a component of a web server (like Apache Tomcat) that provides the runtime environment for servlets, manages their lifecycle, and maps URLs to specific servlets.

In simple words: When you submit a form or click a link in your browser, a servlet on the server can receive that request, process it (e.g., query a database using JDBC), and send back a dynamic response (like an HTML page or JSON data).

---

2. Features of Servlets

- **Platform Independent:** Written in Java, servlets can run on any platform with a JVM.
- **Efficient and Fast:** Servlets are loaded into memory once and handle requests using lightweight threads, making them much faster than technologies like CGI which create a new process for every request.
- **Secure:** They run within the secure environment of the JVM.
- **Robust and Scalable:** The web container manages the servlet's lifecycle, memory, and multithreading, making applications robust.
- **Integrated:** Servlets can easily use other Java APIs like JDBC to connect to databases.
- **Session Management:** They have built-in support for managing user sessions using cookies or HttpSession objects.

---

3. Advantages of Servlets Over CGI

Servlet	CGI (Common Gateway Interface)
Uses threads (faster and more memory efficient)	Creates a new OS process for each request (slow and resource-intensive)
Platform independent	Platform dependent
Secure and robust (managed by JVM)	Less secure
Easy to maintain	Harder to maintain

---

4. Servlet Architecture

The basic architecture follows a request-response model:

1. **Client (Browser):** Sends an HTTP request to the server.

2. **Web Server / Web Container:** Receives the request and forwards it to the appropriate servlet based on the URL.
3. **Servlet:** The servlet's methods are invoked to process the request. It can read form data, interact with a database, or perform other business logic.
4. **Response:** The servlet generates a response (e.g., an HTML page) and sends it back to the client via the web server.

---

5. The Servlet Lifecycle

The lifecycle of a servlet is managed entirely by the web container and consists of three main methods: **init()**, **service()**, and **destroy()**.

Loading and Instantiation

The container loads the servlet class and creates a single instance of it. This happens either on server startup or when the first request for the servlet arrives.

Initialization (init() method)

This method is called only once in the servlet's entire lifetime, immediately after it is instantiated. It's used for one-time setup tasks, like loading configuration or establishing a database connection pool.

```
public void init() throws ServletException {
    // Initialization code
}
```

Request Handling (service() method)

For every client request, the container calls the service() method. This method is the heart of the servlet and is responsible for processing the request and generating a response. The service() method automatically dispatches the request to other methods like doGet() or doPost() based on the HTTP request type.

```
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException {
    // Handle client request
}
```

Destruction (destroy() method)

This method is called only once just before the servlet instance is removed from service (e.g., when the server is shutting down). It is used to release any resources the servlet has acquired, like closing database connections.

```
public void destroy() {
    // Cleanup code
```

```
}
```

After the destroy() method completes, the servlet object is eligible for garbage collection by the JVM.

Lifecycle Flow:

Client Request → Load Servlet (if not loaded) → init() (once) → service() (for each request) → destroy() (once)

---

6. Example: Servlet Lifecycle Program

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LifecycleDemo extends HttpServlet {

    public void init() {
        // This will be printed to the server console once
        // when the servlet is loaded.
        System.out.println("Servlet Initialized");
    }

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        // This is executed for every request to the servlet.
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<h2>Hello, Servlet Lifecycle!</h2>");
    }

    public void destroy() {
        // This will be printed to the server console once
        // when the server is shutting down.
        System.out.println("Servlet Destroyed");
    }
}
```

---

7. Summary of Servlet Lifecycle Methods

Method	When Called	Purpose
init()	Once, when the servlet is first loaded.	One-time initialization and setup.
service()	For every client request.	To process user requests and generate responses.

Method	When Called	Purpose
destroy()	Once, before the servlet is removed from service.	To clean up and release resources.