**JDBC (Java Database Connectivity)**

**[**

**-- Create the database**

**CREATE DATABASE jdbc\_ecommerce;**

**-- Use the database**

**USE jdbc\_ecommerce;**

**-- Create categories table**

**CREATE TABLE categories (**

**category\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**category\_name VARCHAR(100) NOT NULL,**

**description TEXT**

**);**

**-- Create products table**

**CREATE TABLE products (**

**product\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**product\_name VARCHAR(100) NOT NULL,**

**price DECIMAL(10, 2) NOT NULL,**

**stock INT NOT NULL,**

**description TEXT**

**);**

**-- Create products\_categories table (Many-to-Many relationship)**

**CREATE TABLE products\_categories (**

**product\_id INT,**

**category\_id INT,**

**PRIMARY KEY (product\_id, category\_id),**

**FOREIGN KEY (product\_id) REFERENCES products(product\_id),**

**FOREIGN KEY (category\_id) REFERENCES categories(category\_id)**

**);**

**-- Create customers table**

**CREATE TABLE customers (**

**customer\_id INT AUTO\_INCREMENT PRIMARY KEY,**

**first\_name VARCHAR(50) NOT NULL,**

**last\_name VARCHAR(50) NOT NULL,**

**email VARCHAR(100) UNIQUE NOT NULL,**

**phone VARCHAR(15),**

**address TEXT**

**);**

**-- Insert data into categories**

**INSERT INTO categories (category\_name, description) VALUES**

**('Electronics', 'Devices and gadgets'),**

**('Books', 'Wide range of books'),**

**('Clothing', 'Men and Women clothing'),**

**('Home Appliances', 'Appliances for home use'),**

**('Toys', 'Toys for kids of all ages');**

**-- Insert data into products**

**INSERT INTO products (product\_name, price, stock, description) VALUES**

**('Smartphone', 699.99, 50, 'Latest model with high-end specs'),**

**('Laptop', 1299.99, 30, 'Powerful laptop for professionals'),**

**('T-shirt', 19.99, 200, 'Comfortable cotton t-shirt'),**

**('Refrigerator', 499.99, 20, 'Energy efficient fridge'),**

**('Novel', 14.99, 100, 'Bestselling novel'),**

**('Action Figure', 29.99, 150, 'Popular superhero figure'),**

**('Microwave Oven', 199.99, 40, 'Compact microwave oven'),**

**('Headphones', 89.99, 80, 'Noise-cancelling headphones'),**

**('Jeans', 49.99, 120, 'Stylish denim jeans'),**

**('Board Game', 34.99, 90, 'Fun for the whole family');**

**-- Insert data into products\_categories**

**INSERT INTO products\_categories (product\_id, category\_id) VALUES**

**(1, 1), -- Smartphone in Electronics**

**(2, 1), -- Laptop in Electronics**

**(3, 3), -- T-shirt in Clothing**

**(4, 4), -- Refrigerator in Home Appliances**

**(5, 2), -- Novel in Books**

**(6, 5), -- Action Figure in Toys**

**(7, 4), -- Microwave Oven in Home Appliances**

**(8, 1), -- Headphones in Electronics**

**(9, 3), -- Jeans in Clothing**

**(10, 5); -- Board Game in Toys**

**-- Insert data into customers**

**INSERT INTO customers (first\_name, last\_name, email, phone, address) VALUES**

**('John', 'Doe', 'john.doe@example.com', '1234567890', '123 Maple Street'),**

**('Jane', 'Smith', 'jane.smith@example.com', '0987654321', '456 Oak Avenue'),**

**('Alice', 'Johnson', 'alice.j@example.com', '5551234567', '789 Pine Road'),**

**('Bob', 'Brown', 'bob.brown@example.com', '6667891234', '321 Cedar Lane'),**

**('Carol', 'Davis', 'carol.davis@example.com', '7776543210', '654 Birch Boulevard'),**

**('Eve', 'Wilson', 'eve.wilson@example.com', '8883334444', '987 Spruce Street'),**

**('Frank', 'Miller', 'frank.m@example.com', '9991112222', '159 Elm Drive'),**

**('Grace', 'Taylor', 'grace.taylor@example.com', '4445556666', '753 Willow Way'),**

**('Henry', 'Anderson', 'henry.a@example.com', '2223334444', '852 Cherry Crescent'),**

**('Ivy', 'Thomas', 'ivy.thomas@example.com', '1119998888', '369 Poplar Place');**

**]**

**JDBC (Java Database Connectivity)** is an **API** (Application Programming Interface) in Java that enables applications to interact with databases.

It allows Java programs to execute SQL statements, retrieve and update data in databases like MySQL, PostgreSQL, Oracle, and others.

JDBC is part of the **Java Standard Edition (Java SE)**, and it provides a standard way to connect to different databases using common methods and interfaces.

**Key Components of JDBC:**

1. **JDBC Drivers**:
   1. Drivers are specific to each database (e.g., MySQL, Oracle).
   2. These are libraries that implement the JDBC API and handle communication between Java applications and the database.
2. **JDBC API**: A set of interfaces and classes that provide methods for:
   1. Establishing a connection to the database (**DriverManager**, **Connection**).
   2. Executing SQL queries (**Statement**, **PreparedStatement**).
   3. Processing results returned from queries (**ResultSet**).
   4. Handling transactions and batch updates.
3. **JDBC Process**:
   1. **Loading the driver**: The appropriate JDBC driver needs to be loaded.
   2. **Establishing a connection**: Use the **DriverManager** to connect to a database.
   3. **Creating a statement**: SQL queries are executed using **Statement** or **PreparedStatement** objects.
   4. **Executing the query**: Statements are executed to interact with the database (queries, inserts, updates).
   5. **Handling results**: Query results are retrieved using **ResultSet**.
   6. **Closing connections**: Once operations are done, the **connection**, **statement**, and **result** **set** are closed.

**Example of JDBC in Java:**

import java.sql.\*;

public class **JDBCExample01** {

public static void main(String[] args) {

// JDBC URL, username, password

String jdbcURL = "jdbc:mysql://localhost:3306/jdbc\_ecommerce";

String username = "root";

String password = "mysql";

try {

// Step 1: Load MySQL JDBC Driver

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

// Step 2: Establish the connection

Connection connection = DriverManager.getConnection(jdbcURL, username, password);

// Step 3: Create a statement

Statement statement = connection.createStatement();

// Step 4: Execute a query

String query = "SELECT \* FROM products";

ResultSet resultSet = statement.executeQuery(query);

// Step 5: Process the results

while (resultSet.next()) {

System.out.println("Product ID: " + resultSet.getInt("id"));

System.out.println("Product Name: " + resultSet.getString("name"));

System.out.println("Price: " + resultSet.getDouble("price"));

System.out.println("---------");

}

// Step 6: Close resources

resultSet.close();

statement.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

**Advantages of JDBC:**

* **Database Independence**: JDBC provides a uniform interface, allowing interaction with different databases without changing much code.
* **Simplicity**: It simplifies database access through well-defined methods.
* **Transaction Management**: JDBC supports transaction control to ensure data integrity.

In modern development, JDBC often serves as the foundation for more advanced libraries like Hibernate (ORM) or JPA for easier database interaction.

**Explain the importance of Class.forName() in your code**

**Class.forName() in JDBC connections:** Understanding its historical context and importance can provide valuable insights into Java's database handling mechanisms.

Here's a detailed look at the significance of Class.forName() in JDBC:

**Historical Importance of Class.forName()**

1. **Explicit Driver Loading**:
   * In earlier versions of Java (prior to JDBC 4.0), Class.forName("com.mysql.cj.jdbc.Driver") was essential to explicitly load the JDBC driver class.
   * This step was necessary because the driver had to be registered with the DriverManager to be available for establishing a database connection.
2. **Driver Registration**:
   * By calling Class.forName(), the JDBC driver is dynamically loaded and registered with the DriverManager.
   * This allows the DriverManager to be aware of the available drivers and to create connections accordingly.
3. **Backward Compatibility**:
   * Including Class.forName() in your code ensures that it remains compatible with older versions of Java and JDBC.
   * If your application needs to run in environments where the JDBC 4.0 features are not available, explicitly loading the driver becomes necessary.

**Situations Where Class.forName() is Still Relevant**

1. **Custom Drivers**:
   * If you're using a custom JDBC driver that does not have the service provider configuration (META-INF/services/java.sql.Driver), you must use Class.forName() to register the driver manually.
2. **Debugging and Testing**:
   * In scenarios where you are troubleshooting driver-related issues or connection problems, explicitly loading the driver can help identify if the driver is properly included in your classpath.
3. **Code Readability**:
   * Including Class.forName() can improve code clarity by making it explicit which driver is being used. This can be helpful for other developers reading your code or for maintaining legacy systems.

**Example of Usage**

Here’s a simple example that shows how Class.forName() can be used in conjunction with JDBC:

**try {**

**// Load the JDBC driver**

**Class.forName("com.mysql.cj.jdbc.Driver"); // Necessary for older JDBC versions**

**// Establish the connection**

**Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);**

**// Perform database operations...**

**} catch (ClassNotFoundException e) {**

**System.out.println("JDBC Driver not found.");**

**e.printStackTrace();**

**} catch (SQLException e) {**

**e.printStackTrace();**

**}**

**Summary**

While Class.forName() is not mandatory in JDBC 4.0 and later, it has played an essential role in the evolution of JDBC by ensuring that drivers are explicitly loaded and registered with the DriverManager.

Its usage can still be relevant in certain scenarios, particularly for backward compatibility, custom drivers, or enhancing code readability.

Thus, while modern JDBC code may often omit it, understanding its significance helps in grasping how Java manages database connections and drivers.

**Interfaces and classes in JDBC**

In **JDBC (Java Database Connectivity)**, there are several key interfaces and classes that form the core of the API.

These interfaces and classes are used to interact with databases, execute SQL queries, manage connections, and handle results.

Here's a breakdown of the main **interfaces** and **classes** in JDBC:

**Key JDBC Interfaces:**

1. **Driver**
   * The interface that every JDBC driver must implement to connect to a database.
   * The driver is automatically loaded when the application is run.
2. **DriverManager**
   * A **class**, not an interface, that manages a list of database drivers.
   * It is used to establish a connection to the database.
3. **Connection**
   * Represents a connection to the database.
   * It allows the creation of statements and the management of transactions.
   * Methods include createStatement(), prepareStatement(), close(), and transaction control methods like commit() and rollback().
4. **Statement**
   * Represents an SQL statement to be executed against the database.
   * It is used to execute static SQL queries.
   * Methods include executeQuery(), executeUpdate(), and execute().
5. **PreparedStatement**
   * A sub-interface of Statement used for executing precompiled SQL queries with parameterized inputs.
   * Methods include setInt(), setString(), executeQuery(), and executeUpdate().
6. **CallableStatement**
   * A sub-interface of PreparedStatement, used to call stored procedures in the database.
   * Methods include registerOutParameter() for handling output parameters in stored procedures.
7. **ResultSet**
   * Represents the result set of a query. It provides methods to navigate through and retrieve data from the rows in the result set.
   * Methods include next(), getString(), getInt(), and getDouble() for retrieving data.
8. **ResultSetMetaData**
   * Provides information about the structure of a ResultSet, like the number of columns and their types.
   * Methods include getColumnCount() and getColumnName().
9. **DatabaseMetaData**
   * Provides metadata about the database, such as its capabilities, tables, and supported features.
   * Methods include getTables(), getDriverName(), and supportsTransactions().
10. **RowSet**
    * A more advanced interface, which extends ResultSet, representing a set of rows from a database. It is easier to use and more flexible than ResultSet.
11. **SQLWarning**
    * A subclass of SQLException that represents database warnings, typically related to database access.
12. **BatchUpdateException**
    * Represents an exception that provides information about database access errors while attempting to execute a batch of SQL statements.

**Key JDBC Classes:**

1. **DriverManager**
   * A class that provides static methods to manage a list of database drivers and establish a connection to the database.
   * Example method: getConnection() to establish a connection with the database.
2. **Types**
   * A class that defines constants for SQL types. These constants are used in ResultSet and PreparedStatement to specify data types (e.g., Types.INTEGER, Types.VARCHAR).
3. **SQLException**
   * A class that handles database access errors. All JDBC exceptions are instances of SQLException.
   * Methods include getMessage(), getErrorCode(), and getSQLState() to retrieve detailed information about database errors.
4. **SQLTimeoutException**
   * A subclass of SQLException that indicates that a timeout has occurred on a database operation.
5. **SQLDataException**
   * A subclass of SQLException that signals an issue related to invalid data or formatting of data during database operations.
6. **SQLIntegrityConstraintViolationException**
   * A subclass of SQLException that signals an integrity constraint (like primary key or foreign key) violation.
7. **SQLSyntaxErrorException**
   * A subclass of SQLException that indicates that there is a syntax error in the SQL query.

**Example JDBC Usage:**

Here's how some of these interfaces and classes are used in a simple JDBC program:

**package** com.jdbc.demo;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**public** **class** Jdbc004 {

**public** **static** **void** main(String[] args) {

**try**(Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/ashish\_jdbc\_ecommerce",

"root", "mysql");

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("select "

+ " products.product\_name,"

+ " categories.category\_name,"

+ " products.stock"

+ " from products join products\_categories"

+ " on products.product\_id = products\_categories.product\_id"

+ " join categories"

+ " on products\_categories.category\_id = categories.category\_id");){

System.***out***.println("Products and Categories");

**while**(rs.next()) {

System.***out***.println("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

System.***out***.println("\n Product Name: " + rs.getString("product\_name"));

System.***out***.println("\n Category Name: " + rs.getString("category\_name"));

System.***out***.println("\n Stock: " + rs.getString("stock"));

}

}

**catch**(SQLException e) {

System.***out***.println(e.getMessage());

}

}

}

**We haven’t used Class.forName() in above code. How it is working?**

In Java JDBC (Java Database Connectivity), the line Class.forName("com.mysql.cj.jdbc.Driver") was traditionally used to explicitly load the JDBC driver class.

This was necessary in earlier versions of JDBC because it would register the driver with the DriverManager, allowing your application to establish a connection to the database.

However, starting from JDBC 4.0 (Java 6), the JDBC driver can be automatically loaded without needing to call Class.forName() explicitly.

Here’s how it works and why your code can run without it:

**How Automatic Driver Loading Works**

1. **Driver Discovery**: When you include a JDBC driver in your project (e.g., the MySQL Connector/J), it typically contains a META-INF/services/java.sql.Driver file. This file lists the driver classes available. For MySQL, this is usually com.mysql.cj.jdbc.Driver.
2. **Automatic Registration**: When you call DriverManager.getConnection(URL, USER, PASSWORD), the JDBC API scans the classpath for JDBC drivers that are registered in the aforementioned file. If it finds one, it loads the driver class automatically and registers it with the DriverManager.
3. **Connection Creation**: After the driver is registered, the DriverManager can create a connection to the database without the need for Class.forName().

**Implications**

* **Simplifies Code**: You no longer need to write the boilerplate code to load the driver explicitly, which simplifies your code and reduces potential errors.
* **Driver Dependencies**: Ensure that the JDBC driver you are using (e.g., MySQL Connector/J) is included in your project’s classpath. If it is, your code will work without Class.forName().

**Difference between Statement and PreparedStatement**

**PreparedStatement** and **Statement** are two interfaces in Java's JDBC (Java Database Connectivity) API that are used to execute SQL queries against a database.

Here’s a breakdown of what each is and how they differ:

**What is PreparedStatement?**

* **Definition**: PreparedStatement is a subinterface of Statement. It represents a precompiled SQL statement that can be executed multiple times with or without parameters.
* **Usage**: It is typically used for executing parameterized queries, where you can set values dynamically before execution.
* **Performance**: Since the SQL statement is precompiled, it can be executed multiple times with improved performance, especially for repeated execution.
* **Security**: It helps prevent SQL injection attacks because it treats input as data, not executable code.

**Example**:

String sql = "INSERT INTO users (username, password) VALUES (?, ?)";

PreparedStatement pstmt = connection.prepareStatement(sql);

pstmt.setString(1, "john\_doe");

pstmt.setString(2, "secure\_password");

pstmt.executeUpdate();

**What is Statement?**

* **Definition**: Statement is the simplest interface for executing SQL statements directly against a database.
* **Usage**: It is used for executing static SQL queries that do not require parameters.
* **Performance**: Each time a Statement is executed, the SQL statement is compiled, which may lead to performance overhead if executed multiple times.
* **Security**: It is more vulnerable to SQL injection attacks if user inputs are directly concatenated into the SQL query.

**Example**:

String sql = "INSERT INTO users (username, password) VALUES ('john\_doe', 'secure\_password')";

Statement stmt = connection.createStatement();

stmt.executeUpdate(sql);

**Key Differences Between PreparedStatement and Statement**

| **Feature** | **PreparedStatement** | **Statement** |
| --- | --- | --- |
| **SQL Execution** | Precompiled; can execute multiple times with parameters. | Compiles every time it is executed. |
| **Parameter Handling** | Supports parameterized queries (placeholders). | Does not support parameters; requires direct SQL strings. |
| **Security** | Less vulnerable to SQL injection; input treated as data. | More vulnerable to SQL injection. |
| **Performance** | Generally faster for repeated executions due to precompilation. | Slower for repeated executions due to recompilation. |
| **Flexibility** | Can handle complex SQL statements easily with parameters. | Limited to static SQL queries. |

**When to Use Which?**

* Use PreparedStatement when:
  + You need to execute the same SQL statement multiple times with different parameters.
  + Security against SQL injection is a concern.
  + Performance is a priority.
* Use Statement when:
  + You are executing a simple SQL statement once, and it does not require parameters.
  + You are confident that the input is safe and will not lead to SQL injection.

In most cases, especially when user input is involved, it is recommended to use PreparedStatement for both security and performance reasons.

**Example 03:**

Here's a complete example using JDBC to perform **CRUD** (Create, Read, Update, Delete) operations on a **MySQL** database.

The example works with a customers table and demonstrates how to connect to the database, execute SQL queries, and manage transactions using **PreparedStatement**.

**Table Structure**

For this example, we assume a customers table with the following structure:

**CREATE TABLE customers (**

**id INT PRIMARY KEY AUTO\_INCREMENT,**

**name VARCHAR(100),**

**email VARCHAR(100),**

**phone VARCHAR(15)**

**);**

**JDBC CRUD Example**

package com.jdbc.demo;

import java.sql.\*;

public class JDBCCRUDExample {

// Database connection details

private static final String URL = "jdbc:mysql://localhost:3306/jdbc\_ecommerce"; // Change mydb to your database name

private static final String USER = "root"; // Change username if necessary

private static final String PASSWORD = "mysql"; // Change password if necessary

public static void main(String[] args) {

JDBCCRUDExample example = new JDBCCRUDExample();

// Create new customers

example.createCustomer("John Doe", "john.doe@example.com", "1234567890");

example.createCustomer("Jane Smith", "jane.smith@example.com", "0987654321");

// Read all customers

System.out.println("Customers:");

example.readCustomers();

// Update a customer's details

example.updateCustomer(1, "John Updated", "john.updated@example.com", "1122334455");

// Read all customers again to see the updated info

System.out.println("Updated Customers:");

example.readCustomers();

// Delete a customer

example.deleteCustomer(2);

// Read all customers again to see the remaining customers

System.out.println("Customers after deletion:");

example.readCustomers();

}

// Method to create a new customer

public void createCustomer(String name, String email, String phone) {

String sql = "INSERT INTO customers (name, email, phone) VALUES (?, ?, ?)";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, name);

pstmt.setString(2, email);

pstmt.setString(3, phone);

int rowsAffected = pstmt.executeUpdate();

if (rowsAffected > 0) {

System.out.println("Customer created: " + name);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

// Method to read all customers

public void readCustomers() {

String sql = "SELECT \* FROM customers";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql);

ResultSet rs = pstmt.executeQuery()) {

while (rs.next()) {

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

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

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

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

System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email + ", Phone: " + phone);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

// Method to update an existing customer

public void updateCustomer(int id, String name, String email, String phone) {

String sql = "UPDATE customers SET name = ?, email = ?, phone = ? WHERE id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, name);

pstmt.setString(2, email);

pstmt.setString(3, phone);

pstmt.setInt(4, id);

int rowsAffected = pstmt.executeUpdate();

if (rowsAffected > 0) {

System.out.println("Customer updated: " + name);

} else {

System.out.println("No customer found with ID: " + id);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

// Method to delete a customer by ID

public void deleteCustomer(int id) {

String sql = "DELETE FROM customers WHERE id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, id);

int rowsAffected = pstmt.executeUpdate();

if (rowsAffected > 0) {

System.out.println("Customer deleted with ID: " + id);

} else {

System.out.println("No customer found with ID: " + id);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Explanation:**

* **Connection**: A connection to the database is established using DriverManager.getConnection(URL, USER, PASSWORD). This connection is closed automatically thanks to the try-with-resources statement.
* **PreparedStatement**: Prepared statements are used to prevent SQL injection and to efficiently execute SQL queries with dynamic values.
  + createCustomer: Adds a new customer to the customers table.
  + readCustomers: Retrieves all customers and displays them.
  + updateCustomer: Updates the information of an existing customer using their id.
  + deleteCustomer: Deletes a customer based on their id.
* **Transactions**: Each SQL query is executed as a transaction. If successful, changes are committed to the database.

**How it Works:**

1. **Create**: The createCustomer() method inserts a new customer record.
2. **Read**: The readCustomers() method fetches and displays all customer records.
3. **Update**: The updateCustomer() method modifies a customer's information based on the provided id.
4. **Delete**: The deleteCustomer() method removes a customer record from the table based on the provided id.

This example shows how to use **JDBC** with MySQL to perform basic CRUD operations efficiently and securely.

**Example 04:**

Here’s how you can refactor the code using a **CustomerDAO** class to handle the CRUD operations for a Customer object. This demonstrates a more organized and scalable approach following the **Data Access Object (DAO)** pattern.

**Step 1: Create the Customer Class**

This class will represent the Customer object.

package com.jdbc.demo;

public class Customer {

private int id;

private String name;

private String email;

private String phone;

public Customer(int id, String name, String email, String phone) {

this.id = id;

this.name = name;

this.email = email;

this.phone = phone;

}

public Customer(String name, String email, String phone) {

this.name = name;

this.email = email;

this.phone = phone;

}

// Getters and Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

public String getPhone() { return phone; }

public void setPhone(String phone) { this.phone = phone; }

}

**Step 2: Create the CustomerDAO Class**

This class will manage all the database operations (CRUD).

package com.jdbc.demo;

import java.sql.\*;

import java.util.ArrayList;

import java.util.List;

public class CustomerDAO {

private static final String URL = "jdbc:mysql://localhost:3306/jdbc\_ecommerce";

private static final String USER = "root";

private static final String PASSWORD = "mysql";

// Method to create a new customer

public void createCustomer(Customer customer) {

String sql = "INSERT INTO customers (name, email, phone) VALUES (?, ?, ?)";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, customer.getName());

pstmt.setString(2, customer.getEmail());

pstmt.setString(3, customer.getPhone());

pstmt.executeUpdate();

System.out.println("Customer created: " + customer.getName());

} catch (SQLException e) {

e.printStackTrace();

}

}

// Method to retrieve all customers

public List<Customer> readCustomers() {

List<Customer> customers = new ArrayList<>();

String sql = "SELECT \* FROM customers";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql);

ResultSet rs = pstmt.executeQuery()) {

while (rs.next()) {

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

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

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

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

customers.add(new Customer(id, name, email, phone));

}

} catch (SQLException e) {

e.printStackTrace();

}

return customers;

}

// Method to update a customer's details

public void updateCustomer(Customer customer) {

String sql = "UPDATE customers SET name = ?, email = ?, phone = ? WHERE id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, customer.getName());

pstmt.setString(2, customer.getEmail());

pstmt.setString(3, customer.getPhone());

pstmt.setInt(4, customer.getId());

int rowsAffected = pstmt.executeUpdate();

if (rowsAffected > 0) {

System.out.println("Customer updated: " + customer.getName());

} else {

System.out.println("No customer found with ID: " + customer.getId());

}

} catch (SQLException e) {

e.printStackTrace();

}

}

// Method to delete a customer

public void deleteCustomer(int id) {

String sql = "DELETE FROM customers WHERE id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, id);

int rowsAffected = pstmt.executeUpdate();

if (rowsAffected > 0) {

System.out.println("Customer deleted with ID: " + id);

} else {

System.out.println("No customer found with ID: " + id);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Step 3: Create the Main Class**

This class will interact with the CustomerDAO to perform CRUD operations.

package com.jdbc.demo;

import java.util.List;

public class JDBCCRUDWithDAO {

public static void main(String[] args) {

CustomerDAO customerDAO = new CustomerDAO();

// Create new customers

Customer customer1 = new Customer("John Doe", "john.doe@example.com", "1234567890");

Customer customer2 = new Customer("Jane Smith", "jane.smith@example.com", "0987654321");

customerDAO.createCustomer(customer1);

customerDAO.createCustomer(customer2);

// Read and display all customers

System.out.println("Customers:");

List<Customer> customers = customerDAO.readCustomers();

customers.forEach(c -> System.out.println("ID: " + c.getId() + ", Name: " + c.getName() + ", Email: " + c.getEmail() + ", Phone: " + c.getPhone()));

// Update a customer

Customer updatedCustomer = new Customer(1, "John Updated", "john.updated@example.com", "1122334455");

customerDAO.updateCustomer(updatedCustomer);

// Read and display updated customers

System.out.println("Updated Customers:");

customers = customerDAO.readCustomers();

customers.forEach(c -> System.out.println("ID: " + c.getId() + ", Name: " + c.getName() + ", Email: " + c.getEmail() + ", Phone: " + c.getPhone()));

// Delete a customer

customerDAO.deleteCustomer(2);

// Read and display customers after deletion

System.out.println("Customers after deletion:");

customers = customerDAO.readCustomers();

customers.forEach(c -> System.out.println("ID: " + c.getId() + ", Name: " + c.getName() + ", Email: " + c.getEmail() + ", Phone: " + c.getPhone()));

}

}

**Key Points:**

1. **Customer Class**: Represents the customer object with attributes like id, name, email, and phone.
2. **CustomerDAO Class**:
   * **Create**: Inserts a new customer record.
   * **Read**: Fetches all customer records and returns them as a list.
   * **Update**: Updates an existing customer by ID.
   * **Delete**: Deletes a customer by ID.
3. **Main Class**: Demonstrates the use of the DAO by performing all CRUD operations.

This DAO pattern helps to separate the business logic from the database operations, making the code cleaner, modular, and easier to maintain.

**Example 05**

Let’s use different tables, like **Authors** and **Books**, where each book is related to an author.

**Step 1: SQL for Table Creation**

We will create two related tables: authors and books, where each book is written by an author, establishing a foreign key relationship.

CREATE TABLE authors (

author\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL

);

CREATE TABLE books (

book\_id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(150) NOT NULL,

published\_year INT,

price DECIMAL(10, 2) NOT NULL,

author\_id INT,

FOREIGN KEY (author\_id) REFERENCES authors(author\_id)

);

**Step 2: Create the Author and Book Classes**

**Author.java**

package com.jdbc.demo;

public class Author {

private int authorId;

private String name;

private String email;

public Author(int authorId, String name, String email) {

this.authorId = authorId;

this.name = name;

this.email = email;

}

public int getAuthorId() {

return authorId;

}

public String getName() {

return name;

}

public String getEmail() {

return email;

}

public void setName(String name) {

this.name = name;

}

public void setEmail(String email) {

this.email = email;

}

}

**Book.java**

package com.jdbc.demo;

public class Book {

private int bookId;

private String title;

private int publishedYear;

private double price;

private int authorId;

public Book(int bookId, String title, int publishedYear, double price, int authorId) {

this.bookId = bookId;

this.title = title;

this.publishedYear = publishedYear;

this.price = price;

this.authorId = authorId;

}

public int getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public int getPublishedYear() {

return publishedYear;

}

public double getPrice() {

return price;

}

public int getAuthorId() {

return authorId;

}

public void setTitle(String title) {

this.title = title;

}

public void setPublishedYear(int publishedYear) {

this.publishedYear = publishedYear;

}

public void setPrice(double price) {

this.price = price;

}

public void setAuthorId(int authorId) {

this.authorId = authorId;

}

}

**Step 3: DAO Class for Author and Book**

**AuthorDAO.java**

package com.jdbc.demo;

import java.sql.\*;

public class AuthorDAO {

private static final String URL = "jdbc:mysql://localhost:3306/jdbc\_library";

private static final String USER = "root";

private static final String PASSWORD = "mysql";

// Create a new author

public void createAuthor(Author author) throws SQLException {

String sql = "INSERT INTO authors (name, email) VALUES (?, ?)";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, author.getName());

pstmt.setString(2, author.getEmail());

pstmt.executeUpdate();

System.out.println("Author added: " + author.getName());

}

}

// Read all authors

public void readAuthors() throws SQLException {

String sql = "SELECT \* FROM authors";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql);

ResultSet rs = pstmt.executeQuery()) {

while (rs.next()) {

int authorId = rs.getInt("author\_id");

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

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

System.out.println("Author ID: " + authorId + ", Name: " + name + ", Email: " + email);

}

}

}

// Update an existing author

public void updateAuthor(int authorId, String name, String email) throws SQLException {

String sql = "UPDATE authors SET name = ?, email = ? WHERE author\_id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, name);

pstmt.setString(2, email);

pstmt.setInt(3, authorId);

int rowsAffected = pstmt.executeUpdate();

System.out.println(rowsAffected > 0 ? "Author updated" : "No author found with ID: " + authorId);

}

}

// Delete an author

public void deleteAuthor(int authorId) throws SQLException {

String sql = "DELETE FROM authors WHERE author\_id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, authorId);

int rowsAffected = pstmt.executeUpdate();

System.out.println(rowsAffected > 0 ? "Author deleted" : "No author found with ID: " + authorId);

}

}

}

**BookDAO.java**

package com.jdbc.demo;

import java.sql.\*;

public class BookDAO {

private static final String URL = "jdbc:mysql://localhost:3306/jdbc\_library";

private static final String USER = "root";

private static final String PASSWORD = "mysql";

// Create a new book

public void createBook(Book book) throws SQLException {

String sql = "INSERT INTO books (title, published\_year, price, author\_id) VALUES (?, ?, ?, ?)";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, book.getTitle());

pstmt.setInt(2, book.getPublishedYear());

pstmt.setDouble(3, book.getPrice());

pstmt.setInt(4, book.getAuthorId());

pstmt.executeUpdate();

System.out.println("Book added: " + book.getTitle());

}

}

// Read all books

public void readBooks() throws SQLException {

String sql = "SELECT books.book\_id, books.title, books.published\_year, books.price, authors.name AS author\_name " +

"FROM books INNER JOIN authors ON books.author\_id = authors.author\_id";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql);

ResultSet rs = pstmt.executeQuery()) {

while (rs.next()) {

int bookId = rs.getInt("book\_id");

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

int year = rs.getInt("published\_year");

double price = rs.getDouble("price");

String authorName = rs.getString("author\_name");

System.out.println("Book ID: " + bookId + ", Title: " + title + ", Year: " + year +

", Price: " + price + ", Author: " + authorName);

}

}

}

// Update a book

public void updateBook(int bookId, String title, int publishedYear, double price) throws SQLException {

String sql = "UPDATE books SET title = ?, published\_year = ?, price = ? WHERE book\_id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, title);

pstmt.setInt(2, publishedYear);

pstmt.setDouble(3, price);

pstmt.setInt(4, bookId);

int rowsAffected = pstmt.executeUpdate();

System.out.println(rowsAffected > 0 ? "Book updated" : "No book found with ID: " + bookId);

}

}

// Delete a book

public void deleteBook(int bookId) throws SQLException {

String sql = "DELETE FROM books WHERE book\_id = ?";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setInt(1, bookId);

int rowsAffected = pstmt.executeUpdate();

System.out.println(rowsAffected > 0 ? "Book deleted" : "No book found with ID: " + bookId);

}

}

}

**Step 4: Main Program**

package com.jdbc.demo;

public class MainApp {

public static void main(String[] args) throws Exception {

AuthorDAO authorDAO = new AuthorDAO();

BookDAO bookDAO = new BookDAO();

// Create Authors

authorDAO.createAuthor(new Author(0, "J.K. Rowling", "jk.rowling@example.com"));

authorDAO.createAuthor(new Author(0, "George R.R. Martin", "george.martin@example.com"));

// Create Books

bookDAO.createBook(new Book(0, "Harry Potter and the Philosopher's Stone", 1997, 39.99, 1));

bookDAO.createBook(new Book(0, "A Game of Thrones", 1996, 49.99, 2));

// Read Authors and Books

System.out.println("\nAuthors:");

authorDAO.readAuthors();

System.out.println("\nBooks:");

bookDAO.readBooks();

// Update Author and Book

authorDAO.updateAuthor(1, "J.K. Rowling", "rowling.new@example.com");

bookDAO.updateBook(1, "Harry Potter and the Sorcerer's Stone", 1997, 39.99);

// Delete Author and Book

authorDAO.deleteAuthor(2);

bookDAO.deleteBook(2);

}

}

**Summary**

This example demonstrates how to perform CRUD operations on two related tables (authors and books) with the AuthorDAO and BookDAO classes interacting with a MySQL database. The books table contains a foreign key relationship with authors.

**Example 06:**

Here is an example of using a **stored procedure** with **JDBC** to interact with a MySQL database. The scenario demonstrates creating a stored procedure for inserting, reading, and updating customer data.

**SQL for Creating Stored Procedure:**

DELIMITER $$

CREATE PROCEDURE create\_customer(

IN p\_name VARCHAR(100),

IN p\_email VARCHAR(100),

IN p\_phone VARCHAR(15)

)

BEGIN

INSERT INTO customers (name, email, phone)

VALUES (p\_name, p\_email, p\_phone);

END $$

CREATE PROCEDURE get\_customer(

IN p\_customer\_id INT

)

BEGIN

SELECT \* FROM customers WHERE id = p\_customer\_id;

END $$

CREATE PROCEDURE update\_customer(

IN p\_customer\_id INT,

IN p\_name VARCHAR(100),

IN p\_email VARCHAR(100),

IN p\_phone VARCHAR(15)

)

BEGIN

UPDATE customers

SET name = p\_name, email = p\_email, phone = p\_phone

WHERE id = p\_customer\_id;

END $$

DELIMITER ;

In this example, we created three stored procedures:

1. **create\_customer**: Inserts a new customer into the customers table.
2. **get\_customer**: Retrieves a customer by their id.
3. **update\_customer**: Updates a customer's details.

**JDBC Code to Call the Stored Procedures:**

package com.jdbc.demo;

import java.sql.\*;

public class StoredProcedureDemo {

private static final String URL = "jdbc:mysql://localhost:3306/jdbc\_ecommerce";

private static final String USER = "root";

private static final String PASSWORD = "mysql";

public static void main(String[] args) {

StoredProcedureDemo demo = new StoredProcedureDemo();

// Call create\_customer procedure

demo.createCustomer("Alice Green", "alice.green@example.com", "1234567890");

// Call get\_customer procedure

demo.getCustomer(1);

// Call update\_customer procedure

demo.updateCustomer(1, "Alice Brown", "alice.brown@example.com", "0987654321");

// Get the updated customer

demo.getCustomer(1);

}

// Call stored procedure to create a customer

public void createCustomer(String name, String email, String phone) {

String sql = "{CALL create\_customer(?, ?, ?)}";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

CallableStatement stmt = conn.prepareCall(sql)) {

stmt.setString(1, name);

stmt.setString(2, email);

stmt.setString(3, phone);

stmt.execute();

System.out.println("Customer created: " + name);

} catch (SQLException e) {

e.printStackTrace();

}

}

// Call stored procedure to retrieve a customer by ID

public void getCustomer(int customerId) {

String sql = "{CALL get\_customer(?)}";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

CallableStatement stmt = conn.prepareCall(sql)) {

stmt.setInt(1, customerId);

ResultSet rs = stmt.executeQuery();

while (rs.next()) {

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

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

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

System.out.println("ID: " + customerId + ", Name: " + name + ", Email: " + email + ", Phone: " + phone);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

// Call stored procedure to update a customer

public void updateCustomer(int customerId, String name, String email, String phone) {

String sql = "{CALL update\_customer(?, ?, ?, ?)}";

try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);

CallableStatement stmt = conn.prepareCall(sql)) {

stmt.setInt(1, customerId);

stmt.setString(2, name);

stmt.setString(3, email);

stmt.setString(4, phone);

stmt.execute();

System.out.println("Customer updated: " + name);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Explanation of the JDBC Code:**

1. **CallableStatement**: In JDBC, CallableStatement is used to call stored procedures in the database. We use {CALL procedure\_name(?, ?, ?)} syntax to call the stored procedure.
2. **Parameters**: We pass the parameters using stmt.setString() or stmt.setInt() methods to bind values for the procedure parameters.
3. **ResultSet**: For reading customer data, we use ResultSet to handle the result from the stored procedure get\_customer.

**Output Example:**

Customer created: Alice Green

ID: 1, Name: Alice Green, Email: alice.green@example.com, Phone: 1234567890

Customer updated: Alice Brown

ID: 1, Name: Alice Brown, Email: alice.brown@example.com, Phone: 0987654321

This demonstrates how to create, retrieve, and update data using stored procedures with JDBC.

The stored procedures encapsulate SQL logic within the database, improving code modularity and efficiency.

**Project: Library Management System**

**Description:**  
Create a Library Management System that allows librarians to manage books, members, and book loans. This project will involve CRUD operations with JDBC and MySQL, handling multiple tables, and ensuring relational integrity.

**Core Features:**

1. **Book Management:**
   * Add, update, delete, and search books.
   * Fields: book\_id, title, author, genre, publication\_year, availability.
2. **Member Management:**
   * Register new members, update member info, delete accounts, and search for members.
   * Fields: member\_id, name, email, phone\_number, membership\_date.
3. **Loan Management:**
   * Issue books to members and manage return dates.
   * Fields: loan\_id, book\_id, member\_id, issue\_date, due\_date, return\_date.
4. **Penalty Tracking:**
   * Calculate fines for overdue books.
5. **Reports:**
   * Generate reports for issued books, late returns, and member activity.

**Database Schema (MySQL):**

* **books** table
* **members** table
* **loans** table
* **penalties** table

**JDBC Features:**

* Use **Prepared Statements** to prevent SQL injection.
* Implement **search and sort** functionalities for books and members.
* Include **error handling** for database connection failures and invalid data.

**1. Database Creation**

CREATE DATABASE LibraryDB;

USE LibraryDB;

**2. Table Creation**

**Books Table**

CREATE TABLE books (

book\_id INT PRIMARY KEY AUTO\_INCREMENT,

title VARCHAR(255) NOT NULL,

author VARCHAR(255) NOT NULL,

genre VARCHAR(100),

publication\_year INT,

availability BOOLEAN DEFAULT TRUE

);

**Members Table**

CREATE TABLE members (

member\_id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(255) NOT NULL,

email VARCHAR(255) UNIQUE NOT NULL,

phone\_number VARCHAR(15),

membership\_date DATE

);

**Loans Table (with Foreign Keys to books and members)**

CREATE TABLE loans (

loan\_id INT PRIMARY KEY AUTO\_INCREMENT,

book\_id INT,

member\_id INT,

issue\_date DATE,

due\_date DATE,

return\_date DATE,

FOREIGN KEY (book\_id) REFERENCES books(book\_id),

FOREIGN KEY (member\_id) REFERENCES members(member\_id)

);

**Penalties Table (related to loans)**

CREATE TABLE penalties (

penalty\_id INT PRIMARY KEY AUTO\_INCREMENT,

loan\_id INT,

penalty\_amount DECIMAL(6, 2),

paid BOOLEAN DEFAULT FALSE,

FOREIGN KEY (loan\_id) REFERENCES loans(loan\_id)

);

**3. Inserting Sample Data**

**Books**

INSERT INTO books (title, author, genre, publication\_year, availability) VALUES

('The Great Gatsby', 'F. Scott Fitzgerald', 'Classic', 1925, TRUE),

('To Kill a Mockingbird', 'Harper Lee', 'Fiction', 1960, TRUE),

('1984', 'George Orwell', 'Dystopian', 1949, TRUE),

('Pride and Prejudice', 'Jane Austen', 'Romance', 1813, TRUE),

('The Hobbit', 'J.R.R. Tolkien', 'Fantasy', 1937, TRUE),

('Moby Dick', 'Herman Melville', 'Adventure', 1851, TRUE),

('The Catcher in the Rye', 'J.D. Salinger', 'Fiction', 1951, TRUE),

('The Alchemist', 'Paulo Coelho', 'Philosophical', 1988, TRUE),

('War and Peace', 'Leo Tolstoy', 'Historical', 1869, TRUE),

('The Lord of the Rings', 'J.R.R. Tolkien', 'Fantasy', 1954, TRUE);

**Members**

INSERT INTO members (name, email, phone\_number, membership\_date) VALUES

('Alice Johnson', 'alice@example.com', '1234567890', '2022-01-15'),

('Bob Smith', 'bob@example.com', '2345678901', '2021-11-22'),

('Charlie Brown', 'charlie@example.com', '3456789012', '2023-02-10'),

('Diana Prince', 'diana@example.com', '4567890123', '2020-08-05'),

('Ethan Hunt', 'ethan@example.com', '5678901234', '2019-09-18'),

('Fiona Gallagher', 'fiona@example.com', '6789012345', '2022-06-30'),

('George Martin', 'george@example.com', '7890123456', '2021-03-12'),

('Hannah Baker', 'hannah@example.com', '8901234567', '2020-12-01'),

('Ian Fleming', 'ian@example.com', '9012345678', '2023-04-25'),

('Jane Doe', 'jane@example.com', '0123456789', '2021-10-09');

**Loans**

INSERT INTO loans (book\_id, member\_id, issue\_date, due\_date, return\_date) VALUES

(1, 1, '2023-01-10', '2023-01-20', '2023-01-18'),

(2, 2, '2023-02-15', '2023-02-25', '2023-02-28'),

(3, 3, '2023-03-01', '2023-03-11', '2023-03-12'),

(4, 4, '2023-04-05', '2023-04-15', NULL),

(5, 5, '2023-05-10', '2023-05-20', '2023-05-22'),

(6, 6, '2023-06-12', '2023-06-22', '2023-06-25'),

(7, 7, '2023-07-18', '2023-07-28', NULL),

(8, 8, '2023-08-20', '2023-08-30', '2023-09-01'),

(9, 9, '2023-09-25', '2023-10-05', NULL),

(10, 10, '2023-10-15', '2023-10-25', '2023-10-29');

**Penalties**

INSERT INTO penalties (loan\_id, penalty\_amount, paid) VALUES

(2, 5.00, TRUE),

(3, 3.50, FALSE),

(5, 2.00, TRUE),

(6, 4.75, FALSE),

(8, 1.50, TRUE),

(9, 6.00, FALSE),

(4, 7.25, FALSE),

(7, 3.00, TRUE),

(1, 0.00, TRUE),

(10, 5.50, FALSE);

**Relationships Recap:**

* **loans.book\_id** → References **books.book\_id**
* **loans.member\_id** → References **members.member\_id**
* **penalties.loan\_id** → References **loans.loan\_id**

This ensures proper referential integrity between tables.

Java application using **JDBC** to perform **CRUD operations** for the books, members, loans, and penalties tables.

This app will include basic operations such as adding, reading, updating, and deleting records.

**Project Structure**

LibraryManagementSystem/

├── src/

│ ├── LibraryApp.java

│ ├── DBConnection.java

│ ├── BookDAO.java

│ ├── MemberDAO.java

│ ├── LoanDAO.java

│ └── PenaltyDAO.java

└── lib/

└── mysql-connector-java.jar

**1. DBConnection.java (Handles MySQL Connection)**

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DBConnection {

private static final String URL = "jdbc:mysql://localhost:3306/LibraryDB";

private static final String USER = "root";

private static final String PASSWORD = "your\_password";

public static Connection getConnection() throws SQLException {

return DriverManager.getConnection(URL, USER, PASSWORD);

}

}

**2. BookDAO.java (CRUD for Books)**

import java.sql.\*;

import java.util.Scanner;

public class BookDAO {

public void addBook() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Title: ");

String title = sc.nextLine();

System.out.print("Enter Author: ");

String author = sc.nextLine();

System.out.print("Enter Genre: ");

String genre = sc.nextLine();

System.out.print("Enter Publication Year: ");

int year = sc.nextInt();

String sql = "INSERT INTO books (title, author, genre, publication\_year, availability) VALUES (?, ?, ?, ?, TRUE)";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, title);

ps.setString(2, author);

ps.setString(3, genre);

ps.setInt(4, year);

ps.executeUpdate();

System.out.println("Book added successfully!");

} catch (SQLException e) {

e.printStackTrace();

}

}

public void viewBooks() {

try (Connection conn = DBConnection.getConnection();

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM books")) {

System.out.println("Books in Library:");

while (rs.next()) {

System.out.println(rs.getInt("book\_id") + " - " + rs.getString("title") +

" by " + rs.getString("author") + " (" + rs.getInt("publication\_year") + ")");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public void updateBook() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Book ID to Update: ");

int bookId = sc.nextInt();

sc.nextLine(); // Clear buffer

System.out.print("Enter New Title: ");

String newTitle = sc.nextLine();

String sql = "UPDATE books SET title = ? WHERE book\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, newTitle);

ps.setInt(2, bookId);

int rows = ps.executeUpdate();

System.out.println(rows > 0 ? "Book updated successfully!" : "Book not found.");

} catch (SQLException e) {

e.printStackTrace();

}

}

public void deleteBook() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Book ID to Delete: ");

int bookId = sc.nextInt();

String sql = "DELETE FROM books WHERE book\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, bookId);

int rows = ps.executeUpdate();

System.out.println(rows > 0 ? "Book deleted successfully!" : "Book not found.");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**3. MemberDAO.java (Similar CRUD for Members)**

import java.sql.\*;

import java.util.Scanner;

public class MemberDAO {

public void addMember() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Name: ");

String name = sc.nextLine();

System.out.print("Enter Email: ");

String email = sc.nextLine();

System.out.print("Enter Phone Number: ");

String phone = sc.nextLine();

String sql = "INSERT INTO members (name, email, phone\_number, membership\_date) VALUES (?, ?, ?, CURDATE())";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, name);

ps.setString(2, email);

ps.setString(3, phone);

ps.executeUpdate();

System.out.println("Member added successfully!");

} catch (SQLException e) {

e.printStackTrace();

}

}

public void viewMembers() {

try (Connection conn = DBConnection.getConnection();

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM members")) {

System.out.println("Library Members:");

while (rs.next()) {

System.out.println(rs.getInt("member\_id") + " - " + rs.getString("name") +

" (" + rs.getString("email") + ")");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public void updateMember() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Member ID to Update: ");

int memberId = sc.nextInt();

sc.nextLine(); // Clear buffer

System.out.print("Enter New Email: ");

String newEmail = sc.nextLine();

String sql = "UPDATE members SET email = ? WHERE member\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setString(1, newEmail);

ps.setInt(2, memberId);

int rows = ps.executeUpdate();

System.out.println(rows > 0 ? "Member updated successfully!" : "Member not found.");

} catch (SQLException e) {

e.printStackTrace();

}

}

public void deleteMember() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Member ID to Delete: ");

int memberId = sc.nextInt();

String sql = "DELETE FROM members WHERE member\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, memberId);

int rows = ps.executeUpdate();

System.out.println(rows > 0 ? "Member deleted successfully!" : "Member not found.");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**4. LibraryApp.java (Main Application)**

import java.util.Scanner;

public class LibraryApp {

public static void main(String[] args) {

BookDAO bookDAO = new BookDAO();

MemberDAO memberDAO = new MemberDAO();

Scanner sc = new Scanner(System.in);

while (true) {

System.out.println("\n===== Library Management System =====");

System.out.println("1. Add Book");

System.out.println("2. View Books");

System.out.println("3. Update Book");

System.out.println("4. Delete Book");

System.out.println("5. Add Member");

System.out.println("6. View Members");

System.out.println("7. Update Member");

System.out.println("8. Delete Member");

System.out.println("9. Exit");

System.out.print("Choose an option: ");

int choice = sc.nextInt();

switch (choice) {

case 1 -> bookDAO.addBook();

case 2 -> bookDAO.viewBooks();

case 3 -> bookDAO.updateBook();

case 4 -> bookDAO.deleteBook();

case 5 -> memberDAO.addMember();

case 6 -> memberDAO.viewMembers();

case 7 -> memberDAO.updateMember();

case 8 -> memberDAO.deleteMember();

case 9 -> {

System.out.println("Exiting... Goodbye!");

System.exit(0);

}

default -> System.out.println("Invalid option. Try again.");

}

}

}

}

**How to Run the Project:**

1. Add the **MySQL Connector JAR** to your classpath.
2. Update the **DBConnection.java** with your MySQL credentials.
3. Compile and run the LibraryApp.java file.

Great! I'll extend the project by adding **Loans** and **Penalties** CRUD operations.

**5. LoanDAO.java (Handles Book Loans)**

import java.sql.\*;

import java.util.Scanner;

public class LoanDAO {

public void addLoan() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Member ID: ");

int memberId = sc.nextInt();

System.out.print("Enter Book ID: ");

int bookId = sc.nextInt();

String checkBook = "SELECT availability FROM books WHERE book\_id = ?";

PreparedStatement checkStmt = conn.prepareStatement(checkBook);

checkStmt.setInt(1, bookId);

ResultSet rs = checkStmt.executeQuery();

if (rs.next() && rs.getBoolean("availability")) {

String sql = "INSERT INTO loans (member\_id, book\_id, loan\_date, due\_date) VALUES (?, ?, CURDATE(), DATE\_ADD(CURDATE(), INTERVAL 14 DAY))";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, memberId);

ps.setInt(2, bookId);

ps.executeUpdate();

String updateBook = "UPDATE books SET availability = FALSE WHERE book\_id = ?";

PreparedStatement updateStmt = conn.prepareStatement(updateBook);

updateStmt.setInt(1, bookId);

updateStmt.executeUpdate();

System.out.println("Book loaned successfully!");

} else {

System.out.println("Book is not available.");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public void viewLoans() {

try (Connection conn = DBConnection.getConnection();

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM loans")) {

System.out.println("Current Book Loans:");

while (rs.next()) {

System.out.println("Loan ID: " + rs.getInt("loan\_id") + " | Member ID: " + rs.getInt("member\_id") +

" | Book ID: " + rs.getInt("book\_id") + " | Due Date: " + rs.getDate("due\_date"));

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public void returnBook() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Loan ID: ");

int loanId = sc.nextInt();

String sql = "SELECT book\_id, due\_date FROM loans WHERE loan\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, loanId);

ResultSet rs = ps.executeQuery();

if (rs.next()) {

int bookId = rs.getInt("book\_id");

Date dueDate = rs.getDate("due\_date");

Date returnDate = new Date(System.currentTimeMillis());

if (returnDate.after(dueDate)) {

long daysLate = (returnDate.getTime() - dueDate.getTime()) / (1000 \* 60 \* 60 \* 24);

double fineAmount = daysLate \* 5.0;

String insertPenalty = "INSERT INTO penalties (loan\_id, amount, status) VALUES (?, ?, 'Pending')";

PreparedStatement penaltyStmt = conn.prepareStatement(insertPenalty);

penaltyStmt.setInt(1, loanId);

penaltyStmt.setDouble(2, fineAmount);

penaltyStmt.executeUpdate();

System.out.println("Book returned late! Penalty of $" + fineAmount + " applied.");

} else {

System.out.println("Book returned on time.");

}

String updateBook = "UPDATE books SET availability = TRUE WHERE book\_id = ?";

PreparedStatement updateStmt = conn.prepareStatement(updateBook);

updateStmt.setInt(1, bookId);

updateStmt.executeUpdate();

String deleteLoan = "DELETE FROM loans WHERE loan\_id = ?";

PreparedStatement deleteStmt = conn.prepareStatement(deleteLoan);

deleteStmt.setInt(1, loanId);

deleteStmt.executeUpdate();

System.out.println("Book returned successfully!");

} else {

System.out.println("Loan not found.");

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**6. PenaltyDAO.java (Manages Penalties)**

import java.sql.\*;

import java.util.Scanner;

public class PenaltyDAO {

public void viewPenalties() {

try (Connection conn = DBConnection.getConnection();

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM penalties")) {

System.out.println("Penalties:");

while (rs.next()) {

System.out.println("Penalty ID: " + rs.getInt("penalty\_id") + " | Loan ID: " + rs.getInt("loan\_id") +

" | Amount: $" + rs.getDouble("amount") + " | Status: " + rs.getString("status"));

}

} catch (SQLException e) {

e.printStackTrace();

}

}

public void payPenalty() {

try (Connection conn = DBConnection.getConnection();

Scanner sc = new Scanner(System.in)) {

System.out.print("Enter Penalty ID: ");

int penaltyId = sc.nextInt();

String sql = "UPDATE penalties SET status = 'Paid' WHERE penalty\_id = ?";

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, penaltyId);

int rows = ps.executeUpdate();

System.out.println(rows > 0 ? "Penalty paid successfully!" : "Penalty not found.");

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**7. Update LibraryApp.java (Add Loan & Penalty Operations)**

import java.util.Scanner;

public class LibraryApp {

public static void main(String[] args) {

BookDAO bookDAO = new BookDAO();

MemberDAO memberDAO = new MemberDAO();

LoanDAO loanDAO = new LoanDAO();

PenaltyDAO penaltyDAO = new PenaltyDAO();

Scanner sc = new Scanner(System.in);

while (true) {

System.out.println("\n===== Library Management System =====");

System.out.println("1. Add Book");

System.out.println("2. View Books");

System.out.println("3. Update Book");

System.out.println("4. Delete Book");

System.out.println("5. Add Member");

System.out.println("6. View Members");

System.out.println("7. Update Member");

System.out.println("8. Delete Member");

System.out.println("9. Loan Book");

System.out.println("10. View Loans");

System.out.println("11. Return Book");

System.out.println("12. View Penalties");

System.out.println("13. Pay Penalty");

System.out.println("14. Exit");

System.out.print("Choose an option: ");

int choice = sc.nextInt();

switch (choice) {

case 1 -> bookDAO.addBook();

case 2 -> bookDAO.viewBooks();

case 3 -> bookDAO.updateBook();

case 4 -> bookDAO.deleteBook();

case 5 -> memberDAO.addMember();

case 6 -> memberDAO.viewMembers();

case 7 -> memberDAO.updateMember();

case 8 -> memberDAO.deleteMember();

case 9 -> loanDAO.addLoan();

case 10 -> loanDAO.viewLoans();

case 11 -> loanDAO.returnBook();

case 12 -> penaltyDAO.viewPenalties();

case 13 -> penaltyDAO.payPenalty();

case 14 -> {

System.out.println("Exiting... Goodbye!");

System.exit(0);

}

default -> System.out.println("Invalid option. Try again.");

}

}

}

}