Java – Backend Assignment 2024

Module 1 – Overview of IT Industry

What is a Program?

***LAB EXERCISE**: Write a simple "Hello World" program in two different programming languages of your choice. Compare the structure and syntax.*

***THEORY EXERCISE**: Explain in your own words what a program is and how it functions.*

What is Programming?

***THEORY EXERCISE**: What are the key steps involved in the programming process?*

Types of Programming Languages

***THEORY EXERCISE**: What are the main differences between high-level and low-level programming languages?*

World Wide Web & How Internet Works

***LAB EXERCISE**: Research and create a diagram of how data is transmitted from a client to a server over the internet.*

***THEORY EXERCISE**: Describe the roles of the client and server in web communication.*

Network Layers on Client and Server

***LAB EXERCISE**: Design a simple HTTP client-server communication in any language.*

***THEORY EXERCISE**: Explain the function of the TCP/IP model and its layers.*

Client and Servers

***THEORY EXERCISE**: Explain Client Server Communication*

Types of Internet Connections

***LAB EXERCISE**: Research different types of internet connections (e.g., broadband, fiber, satellite) and list their pros and cons.*

***THEORY EXERCISE**: How does broadband differ from fiber-optic internet?*

Protocols

***LAB EXERCISE**: Simulate HTTP and FTP requests using command line tools (e.g., curl).*

***THEORY EXERCISE**: What are the differences between HTTP and HTTPS protocols?*

Application Security

***LAB EXERCISE**: Identify and explain three common application security vulnerabilities. Suggest possible solutions.*

***THEORY EXERCISE**: What is the role of encryption in securing applications?*

Software Applications and Its Types

***LAB EXERCISE**: Identify and classify 5 applications you use daily as either system software or application software.*

***THEORY EXERCISE**: What is the difference between system software and application software?*

Software Architecture

***LAB EXERCISE**: Design a basic three-tier software architecture diagram for a web application.*

***THEORY EXERCISE**: What is the significance of modularity in software architecture?*

Layers in Software Architecture

***LAB EXERCISE**: Create a case study on the functionality of the presentation, business logic, and data access layers of a given software system.*

***THEORY EXERCISE**: Why are layers important in software architecture?*

Software Environments

***LAB EXERCISE**: Explore different types of software environments (development, testing, production). Set up a basic environment in a virtual machine.*

***THEORY EXERCISE**: Explain the importance of a development environment in software production.*

Source Code

***LAB EXERCISE**: Write and upload your first source code file to Github.*

***THEORY EXERCISE**: What is the difference between source code and machine code?*

Github and Introductions

***LAB EXERCISE**: Create a Github repository and document how to commit and push code changes.*

***THEORY EXERCISE**: Why is version control important in software development?*

Student Account in Github

***LAB EXERCISE**: Create a student account on Github and collaborate on a small project with a classmate.*

***THEORY EXERCISE**: What are the benefits of using Github for students?*

Types of Software

***LAB EXERCISE**: Create a list of software you use regularly and classify them into the following categories: system, application, and utility software.*

***THEORY EXERCISE**: What are the differences between open-source and proprietary software?*

GIT and GITHUB Training

***LAB EXERCISE**: Follow a GIT tutorial to practice cloning, branching, and merging repositories.*

***THEORY EXERCISE**: How does GIT improve collaboration in a software development team?*

Application Software

***LAB EXERCISE**: Write a report on the various types of application software and how they improve productivity.*

***THEORY EXERCISE**: What is the role of application software in businesses?*

Software Development Process

***LAB EXERCISE**: Create a flowchart representing the Software Development Life Cycle (SDLC).*

***THEORY EXERCISE**: What are the main stages of the software development process?*

Software Requirement

***LAB EXERCISE**: Write a requirement specification for a simple library management system.*

***THEORY EXERCISE**: Why is the requirement analysis phase critical in software development?*

Software Analysis

***LAB EXERCISE**: Perform a functional analysis for an online shopping system.*

***THEORY EXERCISE**: What is the role of software analysis in the development process?*

System Design

***LAB EXERCISE**: Design a basic system architecture for a food delivery app.*

***THEORY EXERCISE**: What are the key elements of system design?*

Software Testing

***LAB EXERCISE**: Develop test cases for a simple calculator program.*

***THEORY EXERCISE**: Why is software testing important?*

Maintenance

***LAB EXERCISE**: Document a real-world case where a software application required critical maintenance.*

***THEORY EXERCISE**: What types of software maintenance are there?*

Development

***THEORY EXERCISE**: What are the key differences between web and desktop applications?*

27. Web Application

***THEORY EXERCISE**: What are the advantages of using web applications over desktop applications?*

28. Designing

***THEORY EXERCISE**: What role does UI/UX design play in application development?*

29. Mobile Application

***THEORY EXERCISE**: What are the differences between native and hybrid mobile apps?*

30. DFD (Data Flow Diagram)

***LAB EXERCISE**: Create a DFD for a hospital management system.*

***THEORY EXERCISE**: What is the significance of DFDs in system analysis?*

31. Desktop Application

***LAB EXERCISE**: Build a simple desktop calculator application using a GUI library.*

***THEORY EXERCISE**: What are the pros and cons of desktop applications compared to web applications?*

32. Flow Chart

***LAB EXERCISE**: Draw a flowchart representing the logic of a basic online registration system.*

***THEORY EXERCISE**: How do flowcharts help in programming and system design?*

Module 2 – Introduction to Programming

Overview of C Programming

- **THEORY EXERCISE**:
  - Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.
- **LAB EXERCISE**:
  - Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.

2. Setting Up Environment

- **THEORY EXERCISE**:
  - Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.
- **LAB EXERCISE**:
  - Install a C compiler on your system and configure the IDE. Write your first program to print "Hello, World!" and run it.

3. Basic Structure of a C Program

- **THEORY EXERCISE**:
  - Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.
- **LAB EXERCISE**:
  - Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

4. Operators in C

- **THEORY EXERCISE**:
  - Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.
- **LAB EXERCISE**:
  - Write a C program that accepts two integers from the user and performs arithmetic, relational, and logical operations on them. Display the results.

5. Control Flow Statements in C

- **THEORY EXERCISE**:

- Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

- **LAB EXERCISE**:
  - Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the month name based on the user's input (1 for January, 2 for February, etc.).

6. Looping in C

- **THEORY EXERCISE**:
  - Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.
- **LAB EXERCISE**:
  - Write a C program to print numbers from 1 to 10 using all three types of loops (while, for, do-while).

7. Loop Control Statements

- **THEORY EXERCISE**:
  - Explain the use of `break`, `continue`, and `goto` statements in C. Provide examples of each.
- **LAB EXERCISE**:
  - Write a C program that uses the `break` statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 using the `continue` statement.

8. Functions in C

- **THEORY EXERCISE**:
  - What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.
- **LAB EXERCISE**:
  - Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.

9. Arrays in C

- **THEORY EXERCISE**:
  - Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.
- **LAB EXERCISE**:
  - Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

10. Pointers in C

- **THEORY EXERCISE**:
  - Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?
- **LAB EXERCISE**:
  - Write a C program to demonstrate pointer usage. Use a pointer to modify the value of a variable and print the result.

11. Strings in C

- **THEORY EXERCISE**:
  - Explain string handling functions like `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, and `strchr()`. Provide examples of when these functions are useful.
- **LAB EXERCISE**:
  - Write a C program that takes two strings from the user and concatenates them using `strcat()`. Display the concatenated string and its length using `strlen()`.

12. Structures in C

- **THEORY EXERCISE**:
  - Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.
- **LAB EXERCISE**:
  - Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3 students and print them.

13. File Handling in C

- **THEORY EXERCISE**:
  - Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.
- **LAB EXERCISE**:
  - Write a C program to create a file, write a string into it, close the file, then open the file again to read and display its contents.

EXTRA LAB EXERCISES FOR IMPROVING PROGRAMMING LOGIC

1. Operators

LAB EXERCISE 1: Simple Calculator

- Write a C program that acts as a simple calculator. The program should take two numbers and an operator as input from the user and perform the respective operation (addition, subtraction, multiplication, division, or modulus) using operators.

- **Challenge**: Extend the program to handle invalid operator inputs.

LAB EXERCISE 2: Check Number Properties

- Write a C program that takes an integer from the user and checks the following using different operators:
    - Whether the number is even or odd.
    - Whether the number is positive, negative, or zero.
    - Whether the number is a multiple of both 3 and 5.

---

2. Control Statements

LAB EXERCISE 1: Grade Calculator

- Write a C program that takes the marks of a student as input and displays the corresponding grade based on the following conditions:
    - Marks > 90: Grade A
    - Marks > 75 and <= 90: Grade B
    - Marks > 50 and <= 75: Grade C
    - Marks <= 50: Grade D
- Use *if-else* or *switch* statements for the decision-making process.

LAB EXERCISE 2: Number Comparison

- Write a C program that takes three numbers from the user and determines:
    - The largest number.
    - The smallest number.
- **Challenge**: Solve the problem using both *if-else* and *switch-case* statements.

---

3. Loops

LAB EXERCISE 1: Prime Number Check

- Write a C program that checks whether a given number is a prime number or not using a *for* loop.
- **Challenge**: Modify the program to print all prime numbers between 1 and a given number.

LAB EXERCISE 2: Multiplication Table

- Write a C program that takes an integer input from the user and prints its multiplication table using a *for* loop.
- **Challenge**: Allow the user to input the range of the multiplication table (e.g., from 1 to N).

LAB EXERCISE 3: Sum of Digits

- Write a C program that takes an integer from the user and calculates the sum of its digits using a *while* loop.
- **Challenge**: Extend the program to reverse the digits of the number.

---

4. Arrays

LAB EXERCISE 1: Maximum and Minimum in Array

- Write a C program that accepts 10 integers from the user and stores them in an array. The program should then find and print the maximum and minimum values in the array.
- **Challenge**: Extend the program to sort the array in ascending order.

LAB EXERCISE 2: Matrix Addition

- Write a C program that accepts two 2x2 matrices from the user and adds them. Display the resultant matrix.
- **Challenge**: Extend the program to work with 3x3 matrices and matrix multiplication.

LAB EXERCISE 3: Sum of Array Elements

- Write a C program that takes N numbers from the user and stores them in an array. The program should then calculate and display the sum of all array elements.
- **Challenge**: Modify the program to also find the average of the numbers.

---

5. Functions

LAB EXERCISE 1: Fibonacci Sequence

- Write a C program that generates the Fibonacci sequence up to N terms using a recursive function.
- **Challenge**: Modify the program to calculate the Nth Fibonacci number using both iterative and recursive methods. Compare their efficiency.

LAB EXERCISE 2: Factorial Calculation

- Write a C program that calculates the factorial of a given number using a function.
- **Challenge**: Implement both an iterative and a recursive version of the factorial function and compare their performance for large numbers.

LAB EXERCISE 3: Palindrome Check

- Write a C program that takes a number as input and checks whether it is a palindrome using a function.
- **Challenge**: Modify the program to check if a given string is a palindrome.

---

6. Strings

LAB EXERCISE 1: String Reversal

- Write a C program that takes a string as input and reverses it using a function.
- **Challenge**: Write the program without using built-in string handling functions.

LAB EXERCISE 2: Count Vowels and Consonants

- Write a C program that takes a string from the user and counts the number of vowels and consonants in the string.
- **Challenge**: Extend the program to also count digits and special characters.

LAB EXERCISE 3: Word Count

- Write a C program that counts the number of words in a sentence entered by the user.
- **Challenge**: Modify the program to find the longest word in the sentence.

Extra Logic Building Challenges

Lab Challenge 1: Armstrong Number

- Write a C program that checks whether a given number is an Armstrong number or not (e.g., 153 = 1^3 + 5^3 + 3^3).
- **Challenge**: Write a program to find all Armstrong numbers between 1 and 1000.

Lab Challenge 2: Pascal's Triangle

- Write a C program that generates Pascal's Triangle up to N rows using loops.
- **Challenge**: Implement the same program using a recursive function.

Lab Challenge 3: Number Guessing Game

- Write a C program that implements a simple number guessing game. The program should generate a random number between 1 and 100, and the user should guess the number within a limited number of attempts.

- **Challenge**: Provide hints to the user if the guessed number is too high or too low.

Module #3 Introduction to OOPS Programming

1. Introduction to C++

LAB EXERCISES:

1. *First C++ Program: Hello World*
   - Write a simple C++ program to display "Hello, World!".
   - *Objective*: Understand the basic structure of a C++ program, including **#include**, **main()**, and **cout**.
2. *Basic Input/Output*
   - Write a C++ program that accepts user input for their name and age and then displays a personalized greeting.
   - *Objective*: Practice input/output operations using **cin** and **cout**.
3. *POP vs. OOP Comparison Program*
   - Write two small programs: one using Procedural Programming (POP) to calculate the area of a rectangle, and another using Object-Oriented Programming (OOP) with a class and object for the same task.
   - *Objective*: Highlight the difference between POP and OOP approaches.
4. *Setting Up Development Environment*
   - Write a program that asks for two numbers and displays their sum. Ensure this is done after setting up the IDE (like Dev C++ or CodeBlocks).
   - *Objective*: Help students understand how to install, configure, and run programs in an IDE.

THEORY EXERCISE:

1. *What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?*
2. *List and explain the main advantages of OOP over POP.*
3. *Explain the steps involved in setting up a C++ development environment.*
4. *What are the main input/output operations in C++? Provide examples.*

---

2. Variables, Data Types, and Operators

LAB EXERCISES:

1. *Variables and Constants*
   - Write a C++ program that demonstrates the use of variables and constants. Create variables of different data types and perform operations on them.
   - *Objective*: Understand the difference between variables and constants.
2. *Type Conversion*
   - Write a C++ program that performs both implicit and explicit type conversions and prints the results.

- *Objective*: Practice type casting in C++.

3. *Operator Demonstration*
   - Write a C++ program that demonstrates arithmetic, relational, logical, and bitwise operators. Perform operations using each type of operator and display the results.
   - *Objective*: Reinforce understanding of different types of operators in C++.

THEORY EXERCISE:

1. *What are the different data types available in C++? Explain with examples.*
2. *Explain the difference between implicit and explicit type conversion in C++.*
3. *What are the different types of operators in C++? Provide examples of each.*
4. *Explain the purpose and use of constants and literals in C++.*

---

3. Control Flow Statements

LAB EXERCISES:

1. *Grade Calculator*
   - Write a C++ program that takes a student's marks as input and calculates the grade based on if-else conditions.
   - *Objective*: Practice conditional statements (**if-else**).
2. *Number Guessing Game*
   - Write a C++ program that asks the user to guess a number between 1 and 100. The program should provide hints if the guess is too high or too low. Use loops to allow the user multiple attempts.
   - *Objective*: Understand **while** loops and conditional logic.
3. *Multiplication Table*
   - Write a C++ program to display the multiplication table of a given number using a **for** loop.
   - *Objective*: Practice using loops.
4. *Nested Control Structures*
   - Write a program that prints a right-angled triangle using stars (*) with a nested loop.
   - *Objective*: Learn nested control structures.

THEORY EXERCISE:

1. *What are conditional statements in C++? Explain the* if-else *and* switch *statements.*
2. *What is the difference between* for*,* while*, and* do-while *loops in C++?*
3. *How are* break *and* continue *statements used in loops? Provide examples.*
4. *Explain nested control structures with an example.*

---

4. Functions and Scope

LAB EXERCISES:

1. *Simple Calculator Using Functions*
   - Write a C++ program that defines functions for basic arithmetic operations (add, subtract, multiply, divide). The main function should call these based on user input.
   - *Objective*: Practice defining and using functions in C++.
2. *Factorial Calculation Using Recursion*
   - Write a C++ program that calculates the factorial of a number using recursion.
   - *Objective*: Understand recursion in functions.
3. *Variable Scope*
   - Write a program that demonstrates the difference between local and global variables in C++. Use functions to show scope.
   - *Objective*: Reinforce the concept of variable scope.

THEORY EXERCISE:

1. *What is a function in C++? Explain the concept of function declaration, definition, and calling.*
2. *What is the scope of variables in C++? Differentiate between local and global scope.*
3. *Explain recursion in C++ with an example.*
4. *What are function prototypes in C++? Why are they used?*

---

5. Arrays and Strings

LAB EXERCISES:

1. *Array Sum and Average*
   - Write a C++ program that accepts an array of integers, calculates the sum and average, and displays the results.
   - *Objective*: Understand basic array manipulation.
2. *Matrix Addition*
   - Write a C++ program to perform matrix addition on two 2x2 matrices.
   - *Objective*: Practice multi-dimensional arrays.
3. *String Palindrome Check*
   - Write a C++ program to check if a given string is a palindrome (reads the same forwards and backwards).
   - *Objective*: Practice string operations.

THEORY EXERCISE:

1. *What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.*
2. *Explain string handling in C++ with examples.*
3. *How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.*

4. *Explain string operations and functions in C++.*

---

6. Introduction to Object-Oriented Programming

LAB EXERCISES:

1. *Class for a Simple Calculator*
   - Write a C++ program that defines a class **Calculator** with functions for addition, subtraction, multiplication, and division. Create objects to use these functions.
   - *Objective*: Introduce basic class structure.
2. *Class for Bank Account*
   - Create a class **BankAccount** with data members like **balance** and member functions like **deposit** and **withdraw**. Implement encapsulation by keeping the data members private.
   - *Objective*: Understand encapsulation in classes.
3. *Inheritance Example*
   - Write a program that implements inheritance using a base class **Person** and derived classes **Student** and **Teacher**. Demonstrate reusability through inheritance.
   - *Objective*: Learn the concept of inheritance.

THEORY EXERCISE:

1. ***Explain the key concepts of Object-Oriented Programming (OOP).***
2. ***What are classes and objects in C++? Provide an example.***
3. ***What is inheritance in C++? Explain with an example.***
4. ***What is encapsulation in C++? How is it achieved in classes?***

Module 4 – Introduction to DBMS

Introduction to SQL

Theory Questions:

1. What is SQL, and why is it essential in database management?
2. Explain the difference between DBMS and RDBMS.
3. Describe the role of SQL in managing relational databases.
4. What are the key features of SQL?

LAB EXERCISES:

- **Lab 1**: Create a new database named `school_db` and a table called `students` with the following columns: `student_id`, `student_name`, `age`, `class`, and `address`.
- **Lab 2**: Insert five records into the `students` table and retrieve all records using the `SELECT` statement.

---

2. SQL Syntax

Theory Questions:

1. What are the basic components of SQL syntax?
2. Write the general structure of an SQL `SELECT` statement.
3. Explain the role of clauses in SQL statements.

LAB EXERCISES:

- **Lab 1**: Write SQL queries to retrieve specific columns (`student_name` and `age`) from the `students` table.
- **Lab 2**: Write SQL queries to retrieve all students whose age is greater than 10.

---

3. SQL Constraints

Theory Questions:

1. What are constraints in SQL? List and explain the different types of constraints.
2. How do `PRIMARY KEY` and `FOREIGN KEY` constraints differ?
3. What is the role of `NOT NULL` and `UNIQUE` constraints?

LAB EXERCISES:

- **Lab 1**: Create a table `teachers` with the following columns: `teacher_id` (Primary Key), `teacher_name` (NOT NULL), `subject` (NOT NULL), and `email` (UNIQUE).
- **Lab 2**: Implement a `FOREIGN KEY` constraint to relate the `teacher_id` from the `teachers` table with the `students` table.

---

4. Main SQL Commands and Sub-commands (DDL)

Theory Questions:

1. Define the SQL Data Definition Language (DDL).
2. Explain the `CREATE` command and its syntax.
3. What is the purpose of specifying data types and constraints during table creation?

LAB EXERCISES:

- **Lab 1**: Create a table `courses` with columns: `course_id`, `course_name`, and `course_credits`. Set the `course_id` as the primary key.
- **Lab 2**: Use the `CREATE` command to create a database `university_db`.

---

5. ALTER Command

Theory Questions:

1. What is the use of the `ALTER` command in SQL?
2. How can you add, modify, and drop columns from a table using `ALTER`?

LAB EXERCISES:

- **Lab 1**: Modify the `courses` table by adding a column `course_duration` using the `ALTER` command.
- **Lab 2**: Drop the `course_credits` column from the `courses` table.

---

6. DROP Command

Theory Questions:

1. What is the function of the `DROP` command in SQL?
2. What are the implications of dropping a table from a database?

LAB EXERCISES:

- **Lab 1**: Drop the `teachers` table from the `school_db` database.
- **Lab 2**: Drop the `students` table from the `school_db` database and verify that the table has been removed.

---

7. Data Manipulation Language (DML)

Theory Questions:

1. Define the `INSERT`, `UPDATE`, and `DELETE` commands in SQL.
2. What is the importance of the `WHERE` clause in `UPDATE` and `DELETE` operations?

LAB EXERCISES:

- **Lab 1**: Insert three records into the `courses` table using the `INSERT` command.
- **Lab 2**: Update the course duration of a specific course using the `UPDATE` command.
- **Lab 3**: Delete a course with a specific `course_id` from the `courses` table using the `DELETE` command.

---

8. Data Query Language (DQL)

Theory Questions:

1. What is the `SELECT` statement, and how is it used to query data?
2. Explain the use of the `ORDER BY` and `WHERE` clauses in SQL queries.

LAB EXERCISES:

- **Lab 1**: Retrieve all courses from the `courses` table using the `SELECT` statement.
- **Lab 2**: Sort the courses based on `course_duration` in descending order using `ORDER BY`.
- **Lab 3**: Limit the results of the `SELECT` query to show only the top two courses using `LIMIT`.

---

9. Data Control Language (DCL)

Theory Questions:

1. What is the purpose of `GRANT` and `REVOKE` in SQL?
2. How do you manage privileges using these commands?

LAB EXERCISES:

- **Lab 1**: Create two new users `user1` and `user2` and grant `user1` permission to `SELECT` from the `courses` table.
- **Lab 2**: Revoke the `INSERT` permission from `user1` and give it to `user2`.

---

10. Transaction Control Language (TCL)

Theory Questions:

1. What is the purpose of the `COMMIT` and `ROLLBACK` commands in SQL?
2. Explain how transactions are managed in SQL databases.

LAB EXERCISES:

- **Lab 1**: Insert a few rows into the `courses` table and use `COMMIT` to save the changes.
- **Lab 2**: Insert additional rows, then use `ROLLBACK` to undo the last insert operation.
- **Lab 3**: Create a `SAVEPOINT` before updating the `courses` table, and use it to roll back specific changes.

---

11. SQL Joins

Theory Questions:

1. Explain the concept of `JOIN` in SQL. What is the difference between `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL OUTER JOIN`?
2. How are joins used to combine data from multiple tables?

LAB EXERCISES:

- **Lab 1**: Create two tables: `departments` and `employees`. Perform an `INNER JOIN` to display employees along with their respective departments.
- **Lab 2**: Use a `LEFT JOIN` to show all departments, even those without employees.

---

12. SQL Group By

Theory Questions:

1. What is the `GROUP BY` clause in SQL? How is it used with aggregate functions?
2. Explain the difference between `GROUP BY` and `ORDER BY`.

LAB EXERCISES:

- **Lab 1**: Group employees by department and count the number of employees in each department using `GROUP BY`.
- **Lab 2**: Use the `AVG` aggregate function to find the average salary of employees in each department.

---

13. SQL Stored Procedure

Theory Questions:

1. What is a stored procedure in SQL, and how does it differ from a standard SQL query?
2. Explain the advantages of using stored procedures.

LAB EXERCISES:

- **Lab 1**: Write a stored procedure to retrieve all employees from the `employees` table based on department.
- **Lab 2**: Write a stored procedure that accepts `course_id` as input and returns the course details.

---

14. SQL View

Theory Questions:

1. What is a view in SQL, and how is it different from a table?
2. Explain the advantages of using views in SQL databases.

LAB EXERCISES:

- **Lab 1**: Create a view to show all employees along with their department names.
- **Lab 2**: Modify the view to exclude employees whose salaries are below $50,000.

---

15. SQL Triggers

Theory Questions:

1. What is a trigger in SQL? Describe its types and when they are used.
2. Explain the difference between `INSERT`, `UPDATE`, and `DELETE` triggers.

LAB EXERCISES:

- **Lab 1**: Create a trigger to automatically log changes to the `employees` table when a new employee is added.
- **Lab 2**: Create a trigger to update the `last_modified` timestamp whenever an employee record is updated.

---

16. Introduction to PL/SQL

Theory Questions:

1. What is PL/SQL, and how does it extend SQL's capabilities?
2. List and explain the benefits of using PL/SQL.

LAB EXERCISES:

- **Lab 1**: Write a PL/SQL block to print the total number of employees from the `employees` table.
- **Lab 2**: Create a PL/SQL block that calculates the total sales from an `orders` table.

---

17. PL/SQL Control Structures

Theory Questions:

1. What are control structures in PL/SQL? Explain the `IF-THEN` and `LOOP` control structures.
2. How do control structures in PL/SQL help in writing complex queries?

LAB EXERCISES:

- *Lab 1*: Write a PL/SQL block using an `IF-THEN` condition to check the department of an employee.
- **Lab 2**: Use a `FOR LOOP` to iterate through employee records and display their names.

---

18. SQL Cursors

Theory Questions:

1. What is a cursor in PL/SQL? Explain the difference between implicit and explicit cursors.
2. When would you use an explicit cursor over an implicit one?

LAB EXERCISES:

- **Lab 1**: Write a PL/SQL block using an explicit cursor to retrieve and display employee details.
- **Lab 2**: Create a cursor to retrieve all courses and display them one by one.

---

19. Rollback and Commit Savepoint

Theory Questions:

1. Explain the concept of `SAVEPOINT` in transaction management. How do `ROLLBACK` and `COMMIT` interact with savepoints?
2. When is it useful to use savepoints in a database transaction?

LAB EXERCISES:

- **Lab 1**: Perform a transaction where you create a savepoint, insert records, then rollback to the savepoint.
- **Lab 2**: Commit part of a transaction after using a savepoint and then rollback the remaining changes.

EXTRA LAB PRACTISE FOR DATABASE CONCEPTS

1. Introduction to SQL

LAB EXERCISES:

- **Lab 3**: Create a database called `library_db` and a table `books` with columns: `book_id`, `title`, `author`, `publisher`, `year_of_publication`, and `price`. Insert five records into the table.
- **Lab 4**: Create a table `members` in `library_db` with columns: `member_id`, `member_name`, `date_of_membership`, and `email`. Insert five records into this table.

---

2. SQL Syntax

LAB EXERCISES:

- **Lab 3**: Retrieve all `members` who joined the library before 2022. Use appropriate SQL syntax with `WHERE` and `ORDER BY`.
- **Lab 4**: Write SQL queries to display the titles of books published by a specific author. Sort the results by `year_of_publication` in descending order.

---

3. SQL Constraints

LAB EXERCISES:

- **Lab 3**: Add a `CHECK` constraint to ensure that the `price` of books in the `books` table is greater than 0.
- **Lab 4**: Modify the `members` table to add a `UNIQUE` constraint on the `email` column, ensuring that each member has a unique email address.

---

4. Main SQL Commands and Sub-commands (DDL)

LAB EXERCISES:

- **Lab 3**: Create a table `authors` with the following columns: `author_id`, `first_name`, `last_name`, and `country`. Set `author_id` as the primary key.
- **Lab 4**: Create a table `publishers` with columns: `publisher_id`, `publisher_name`, `contact_number`, and `address`. Set `publisher_id` as the primary key and `contact_number` as unique.

---

5. ALTER Command

LAB EXERCISES:

- **Lab 3**: Add a new column `genre` to the `books` table. Update the `genre` for all existing records.
- **Lab 4**: Modify the `members` table to increase the length of the `email` column to 100 characters.

---

6. DROP Command

LAB EXERCISES:

- **Lab 3**: Drop the `publishers` table from the database after verifying its structure.
- **Lab 4**: Create a backup of the `members` table and then drop the original `members` table.

---

7. Data Manipulation Language (DML)

LAB EXERCISES:

- **Lab 4**: Insert three new authors into the `authors` table, then update the last name of one of the authors.
- **Lab 5**: Delete a book from the `books` table where the `price` is higher than $100.

---

8. UPDATE Command

LAB EXERCISES:

- **Lab 3**: Update the `year_of_publication` of a book with a specific `book_id`.
- **Lab 4**: Increase the `price` of all books published before 2015 by 10%.

---

9. DELETE Command

LAB EXERCISES:

- **Lab 3**: Remove all members who joined before 2020 from the `members` table.
- **Lab 4**: Delete all books that have a `NULL` value in the `author` column.

---

10. Data Query Language (DQL)

LAB EXERCISES:

- **Lab 4**: Write a query to retrieve all `books` with `price` between $50 and $100.
- **Lab 5**: Retrieve the list of `books` sorted by `author` in ascending order and limit the results to the top 3 entries.

---

11. Data Control Language (DCL)

LAB EXERCISES:

- **Lab 3**: Grant `SELECT` permission to a user named `librarian` on the `books` table.
- **Lab 4**: Grant `INSERT` and `UPDATE` permissions to the user `admin` on the `members` table.

---

12. REVOKE Command

LAB EXERCISES:

- **Lab 3**: Revoke the `INSERT` privilege from the user `librarian` on the `books` table.
- **Lab 4**: Revoke all permissions from user `admin` on the `members` table.

---

13. Transaction Control Language (TCL)

LAB EXERCISES:

- **Lab 3**: Use `COMMIT` after inserting multiple records into the `books` table, then make another insertion and perform a `ROLLBACK`.
- **Lab 4**: Set a `SAVEPOINT` before making updates to the `members` table, perform some updates, and then roll back to the `SAVEPOINT`.

---

14. SQL Joins

LAB EXERCISES:

- **Lab 3**: Perform an `INNER JOIN` between `books` and `authors` tables to display the `title` of books and their respective authors' names.
- **Lab 4**: Use a `FULL OUTER JOIN` to retrieve all records from the `books` and `authors` tables, including those with no matching entries in the other table.

---

15. SQL Group By

LAB EXERCISES:

- **Lab 3**: Group `books` by `genre` and display the total number of books in each genre.
- **Lab 4**: Group `members` by the year they joined and find the number of members who joined each year.

---

16. SQL Stored Procedure

LAB EXERCISES:

- **Lab 3**: Write a stored procedure to retrieve all `books` by a particular `author`.

- **Lab 4**: Write a stored procedure that takes `book_id` as an argument and returns the `price` of the book.

---

17. SQL View

LAB EXERCISES:

- **Lab 3**: Create a view to show only the `title`, `author`, and `price` of books from the `books` table.
- **Lab 4**: Create a view to display `members` who joined before 2020.

---

18. SQL Trigger

LAB EXERCISES:

- **Lab 3**: Create a trigger to automatically update the `last_modified` timestamp of the `books` table whenever a record is updated.
- **Lab 4**: Create a trigger that inserts a log entry into a `log_changes` table whenever a `DELETE` operation is performed on the `books` table.

---

19. Introduction to PL/SQL

LAB EXERCISES:

- **Lab 3**: Write a PL/SQL block to insert a new `book` into the `books` table and display a confirmation message.
- **Lab 4**: Write a PL/SQL block to display the total number of books in the `books` table.

---

20. PL/SQL Syntax

LAB EXERCISES:

- **Lab 3**: Write a PL/SQL block to declare variables for `book_id` and `price`, assign values, and display the results.
- **Lab 4**: Write a PL/SQL block using `constants` and perform arithmetic operations on book prices.

---

21. PL/SQL Control Structures

LAB EXERCISES:

- **Lab 3**: Write a PL/SQL block using `IF-THEN-ELSE` to check if a book's price is above $100 and print a message accordingly.
- **Lab 4**: Use a `FOR LOOP` in PL/SQL to display the details of all books one by one.

---

22. SQL Cursors

LAB EXERCISES:

- **Lab 3**: Write a PL/SQL block using an explicit cursor to fetch and display all records from the `members` table.
- **Lab 4**: Create a cursor to retrieve books by a particular author and display their titles.

---

23. Rollback and Commit Savepoint

LAB EXERCISES:

- **Lab 3**: Perform a transaction that includes inserting a new `member`, setting a `SAVEPOINT`, and rolling back to the savepoint after making updates.
- **Lab 4**: Use `COMMIT` after successfully inserting multiple books into the `books` table, then use `ROLLBACK` to undo a set of changes made after a savepoint.

Module 6 – Core Java

1. Introduction to Java

- **Theory**:
  - History of Java
  - Features of Java (Platform Independent, Object-Oriented, etc.)
  - Understanding JVM, JRE, and JDK
  - Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)
  - Java Program Structure (Packages, Classes, Methods)
- **Lab Exercise**:
  - Install JDK and set up environment variables.
  - Write a simple "Hello World" Java program.
  - Compile and run the program using command-line tools (javac, java).

2. Data Types, Variables, and Operators

- **Theory**:
  - Primitive Data Types in Java (int, float, char, etc.)
  - Variable Declaration and Initialization
  - Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise
  - Type Conversion and Type Casting
- **Lab Exercise**:
  - Write a program to demonstrate the use of different data types.
  - Create a calculator using arithmetic and relational operators.
  - Demonstrate type casting (explicit and implicit).

3. Control Flow Statements

- **Theory**:
  - If-Else Statements
  - Switch Case Statements
  - Loops (For, While, Do-While)
  - Break and Continue Keywords
- **Lab Exercise**:
  - Write a program to find if a number is even or odd using an if-else statement.
  - Implement a simple menu-driven program using a switch-case.
  - Write a program to display the Fibonacci series using a loop.

4. Classes and Objects

- **Theory**:
  - Defining a Class and Object in Java
  - Constructors and Overloading
  - Object Creation, Accessing Members of the Class
  - this Keyword
- **Lab Exercise**:
  - Create a class `Student` with attributes (name, age) and a method to display the details.

    - Create multiple constructors in a class and demonstrate constructor overloading.
    - Implement a simple class with getters and setters for encapsulation.

5. Methods in Java

- **Theory**:
    - Defining Methods
    - Method Parameters and Return Types
    - Method Overloading
    - Static Methods and Variables
- **Lab Exercise**:
    - Write a program to find the maximum of three numbers using a method.
    - Implement method overloading by creating methods for different data types.
    - Create a class with static variables and methods to demonstrate their use.

6. Object-Oriented Programming (OOPs) Concepts

- **Theory**:
    - Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction
    - Inheritance: Single, Multilevel, Hierarchical
    - Method Overriding and Dynamic Method Dispatch
- **Lab Exercise**:
    - Write a program demonstrating single inheritance.
    - Create a class hierarchy and demonstrate multilevel inheritance.
    - Implement method overriding to show polymorphism in action.

7. Constructors and Destructors

- **Theory**:
    - Constructor Types (Default, Parameterized)
    - Copy Constructor (Emulated in Java)
    - Constructor Overloading
    - Object Life Cycle and Garbage Collection
- **Lab Exercise**:
    - Write a program to create and initialize an object using a parameterized constructor.
    - Demonstrate constructor overloading by passing different types of parameters.

8. Arrays and Strings

- **Theory**:
    - One-Dimensional and Multidimensional Arrays
    - String Handling in Java: String Class, StringBuffer, StringBuilder
    - Array of Objects
    - String Methods (length, charAt, substring, etc.)
- **Lab Exercise**:
    - Write a program to perform matrix addition and subtraction using 2D arrays.
    - Create a program to reverse a string and check for palindromes.
    - Implement string comparison using `equals()` and `compareTo()` methods.

9. Inheritance and Polymorphism

- **Theory**:
  - Inheritance Types and Benefits
  - Method Overriding
  - Dynamic Binding (Run-Time Polymorphism)
  - Super Keyword and Method Hiding
- **Lab Exercise**:
  - Write a program that demonstrates inheritance using `extends` keyword.
  - Implement runtime polymorphism by overriding methods in the child class.
  - Use the `super` keyword to call the parent class constructor and methods.

10. Interfaces and Abstract Classes

- **Theory**:
  - Abstract Classes and Methods
  - Interfaces: Multiple Inheritance in Java
  - Implementing Multiple Interfaces
- **Lab Exercise**:
  - Create an abstract class and implement its methods in a subclass.
  - Write a program that implements multiple interfaces in a single class.
  - Implement an interface for a real-world example, such as a payment gateway.

11. Packages and Access Modifiers

- **Theory**:
  - Java Packages: Built-in and User-Defined Packages
  - Access Modifiers: Private, Default, Protected, Public
  - Importing Packages and Classpath
- **Lab Exercise**:
  - Create a user-defined package and import it into another program.
  - Demonstrate the use of different access modifiers within the same package and across different packages.

12. Exception Handling

- **Theory**:
  - Types of Exceptions: Checked and Unchecked
  - try, catch, finally, throw, throws
  - Custom Exception Classes
- **Lab Exercise**:
  - Write a program to demonstrate exception handling using try-catch-finally.
  - Implement multiple catch blocks for different types of exceptions.
  - Create a custom exception class and use it in your program.

13. Multithreading

- **Theory**:

    - Introduction to Threads
    - Creating Threads by Extending Thread Class or Implementing Runnable Interface
    - Thread Life Cycle
    - Synchronization and Inter-thread Communication
- **Lab Exercise**:
    - Write a program to create and run multiple threads using the `Thread` class.
    - Implement thread synchronization using `synchronized` blocks or methods.
    - Use inter-thread communication methods like `wait()`, `notify()`, and `notifyAll()`.

14. File Handling

- **Theory**:
    - Introduction to File I/O in Java (java.io package)
    - FileReader and FileWriter Classes
    - BufferedReader and BufferedWriter
    - Serialization and Deserialization
- **Lab Exercise**:
    - Write a program to read and write content to a file using `FileReader` and `FileWriter`.
    - Implement a program that reads a file line by line using `BufferedReader`.
    - Create a program that demonstrates object serialization and deserialization.

15. Collections Framework

- **Theory**:
    - Introduction to Collections Framework
    - List, Set, Map, and Queue Interfaces
    - ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap
    - Iterators and ListIterators
- **Lab Exercise**:
    - Write a program that demonstrates the use of an `ArrayList` and `LinkedList`.
    - Implement a program using `HashSet` to remove duplicate elements from a list.
    - Create a `HashMap` to store and retrieve key-value pairs.

16. Java Input/Output (I/O)

- **Theory**:
    - Streams in Java (InputStream, OutputStream)
    - Reading and Writing Data Using Streams
    - Handling File I/O Operations
- **Lab Exercise**:
    - Write a program to read input from the console using `Scanner`.
    - Implement a file copy program using `FileInputStream` and `FileOutputStream`.
    - Create a program that reads from one file and writes the content to another file.

Module 7 – Java – RDBMS & Database Programming with JDBC

Introduction to JDBC

- **Theory**:
  - What is JDBC (Java Database Connectivity)?
  - Importance of JDBC in Java Programming
  - JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet
- **Lab Exercise**:
  - Write a simple Java program to connect to a MySQL database using JDBC.
  - Demonstrate the process of loading a JDBC driver and establishing a connection.

2. JDBC Driver Types

- **Theory**:
  - Overview of JDBC Driver Types:
    - Type 1: JDBC-ODBC Bridge Driver
    - Type 2: Native-API Driver
    - Type 3: Network Protocol Driver
    - Type 4: Thin Driver
  - Comparison and Usage of Each Driver Type
- **Lab Exercise**:
  - Identify which driver your Java program uses to connect to MySQL.
  - Research and explain the best JDBC driver for your database and Java environment.

3. Steps for Creating JDBC Connections

- **Theory**:
  - Step-by-Step Process to Establish a JDBC Connection:
    1. Import the JDBC packages
    2. Register the JDBC driver
    3. Open a connection to the database
    4. Create a statement
    5. Execute SQL queries
    6. Process the result set
    7. Close the connection
- **Lab Exercise**:
  - Write a Java program to establish a connection to a database and print a confirmation message upon successful connection.

4. Types of JDBC Statements

- **Theory**:
  - Overview of JDBC Statements:
    - `Statement`: Executes simple SQL queries without parameters.
    - `PreparedStatement`: Precompiled SQL statements for queries with parameters.
    - `CallableStatement`: Used to call stored procedures.

    - Differences between `Statement`, `PreparedStatement`, and `CallableStatement`
- **Lab Exercise**:
    - Create a program that inserts, updates, selects, and deletes data using `Statement`.
    - Modify the program to use `PreparedStatement` for parameterized queries.

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

- **Theory**:
    - Insert: Adding a new record to the database.
    - Update: Modifying existing records.
    - Select: Retrieving records from the database.
    - Delete: Removing records from the database.
- **Lab Exercise**:
    - Write a Java program that performs the following CRUD operations:
        - Insert a new record.
        - Update an existing record.
        - Select and display records.
        - Delete a record from the database.

6. ResultSet Interface

- **Theory**:
    - What is `ResultSet` in JDBC?
    - Navigating through `ResultSet` (first, last, next, previous)
    - Working with `ResultSet` to retrieve data from SQL queries
- **Lab Exercise**:
    - Write a program that executes a SELECT query and processes the `ResultSet` to display records from the database.
    - Demonstrate how to navigate through the `ResultSet` using methods like `next()`, `previous()`, etc.

7. Database Metadata

- **Theory**:
    - What is `DatabaseMetaData`?
    - Importance of Database Metadata in JDBC
    - Methods provided by `DatabaseMetaData` (getDatabaseProductName, getTables, etc.)
- **Lab Exercise**:
    - Write a program that retrieves and displays metadata information about your database using `DatabaseMetaData`.
    - Display database name, version, list of tables, and supported SQL features.

8. ResultSet Metadata

- **Theory**:
    - What is `ResultSetMetaData`?
    - Importance of ResultSet Metadata in analyzing the structure of query results

    - Methods in `ResultSetMetaData` (getColumnCount, getColumnName, getColumnType)
- **Lab Exercise**:
    - Write a program that retrieves and displays column names, types, and count of a `ResultSet` using `ResultSetMetaData`.
    - Use a SELECT query to display this metadata for a specific table.

9. Practical SQL Query Examples

- **Lab Exercise**:
    - Write SQL queries for:
        - Inserting a record into a table.
        - Updating specific fields of a record.
        - Selecting records based on certain conditions.
        - Deleting specific records.
    - Implement these queries in Java using JDBC.

10. Practical Example 1: Swing GUI for CRUD Operations

- **Theory**:
    - Introduction to Java Swing for GUI development
    - How to integrate Swing components with JDBC for CRUD operations
- **Lab Exercise**:
    - Create a simple Swing GUI with input fields for `id`, `fname`, `lname`, and `email`.
    - Implement CRUD operations (Insert, Update, Select, Delete) using JDBC and MySQL.
    - On button clicks, the program should interact with the database and perform the appropriate operation (insert, update, display records, or delete records).

11. Practical Example 2: Callable Statement with IN and OUT Parameters

- **Theory**:
    - What is a `CallableStatement`?
    - How to call stored procedures using `CallableStatement` in JDBC
    - Working with IN and OUT parameters in stored procedures
- **Lab Exercise**:
    - Create a stored procedure in MySQL with IN and OUT parameters (e.g., a procedure that takes an employee ID as input and returns the employee's full name as output).
    - Write a Java program that uses `CallableStatement` to call this stored procedure.
    - Demonstrate how to pass IN parameters and retrieve OUT parameters.

Sample Lab Assignments Summary:

Lab Assignment 1: Simple JDBC Program

1. Write a Java program that connects to a MySQL database and executes a simple query to retrieve all records from a table.

Lab Assignment 2: CRUD Operations using JDBC

1. Write a Java program that performs the following operations on a MySQL database:
    - Insert a new record.
    - Update an existing record.
    - Select and display records.
    - Delete a record.

Lab Assignment 3: Swing GUI with JDBC

1. Create a Swing-based GUI with fields for `id`, `fname`, `lname`, and `email`.
2. Implement buttons for Insert, Update, Select, and Delete.
3. Perform the corresponding JDBC operations for each button click.

Lab Assignment 4: Using CallableStatement

1. Create a stored procedure in MySQL with IN and OUT parameters.
2. Write a Java program that calls the stored procedure using `CallableStatement` and demonstrates how to pass parameters and retrieve results.

Module 8) Web Technologies in Java

HTML Tags: Anchor, Form, Table, Image, List Tags, Paragraph, Break, Label

Theory:

- Introduction to HTML and its structure.
- Explanation of key tags:
  - `<a>`: Anchor tag for hyperlinks.
  - `<form>`: Form tag for user input.
  - `<table>`: Table tag for data representation.
  - `<img>`: Image tag for embedding images.
  - List tags: `<ul>`, `<ol>`, and `<li>`.
  - `<p>`: Paragraph tag.
  - `<br>`: Line break.
  - `<label>`: Label for form inputs.

Lab Exercise:

1. Create a webpage that includes:
   - A navigation menu with anchor tags.
   - A form with input fields, labels, and a submit button.
   - A table that displays user data.
   - Images with appropriate `alt` text.
   - Both ordered and unordered lists.

CSS: Inline CSS, Internal CSS, External CSS

Theory:

- Overview of CSS and its importance in web design.
- Types of CSS:
  - **Inline CSS**: Directly in HTML elements.
  - **Internal CSS**: Inside a `<style>` tag in the head section.
  - **External CSS**: Linked to an external file.

Lab Exercise:

1. Create a webpage where:
   - You apply inline CSS to an element.
   - Use internal CSS for another element.
   - Link an external CSS file to style other elements.

CSS: Margin and Padding

Theory:

- Definition and difference between margin and padding.
- How margins create space outside the element and padding creates space inside.

Lab Exercise:

1. Create a webpage and use CSS to demonstrate:
    - Margin applied to an element.
    - Padding applied to a div.
    - The effect of different margin and padding values on the layout.

CSS: Pseudo-Class

Theory:

- Introduction to CSS pseudo-classes like `:hover`, `:focus`, `:active`, etc.
- Use of pseudo-classes to style elements based on their state.

Lab Exercise:

1. Create a navigation menu and use pseudo-classes to:
    - Change the color of links on hover.
    - Style form inputs when they are focused.

CSS: ID and Class Selectors

Theory:

- Difference between `id` and `class` in CSS.
- Usage scenarios for `id` (unique) and `class` (reusable).

Lab Exercise:

1. Create a webpage where:
    - You apply an `id` to an element and style it uniquely.
    - Use `class` to apply the same style to multiple elements.

Introduction to Client-Server Architecture

Theory:

- Overview of client-server architecture.
- Difference between client-side and server-side processing.
- Roles of a client, server, and communication protocols.

Lab Exercise:

1. Create a diagram explaining client-server communication flow and explain how a request is processed by the server and sent back to the client.

HTTP Protocol Overview with Request and Response Headers

Theory:

- Introduction to the HTTP protocol and its role in web communication.
- Explanation of HTTP request and response headers.

Lab Exercise:

1. Create a Java servlet that:
    - Displays the HTTP request headers.
    - Sends an HTTP response with custom headers.

J2EE Architecture Overview

Theory:

- Introduction to J2EE and its multi-tier architecture.
- Role of web containers, application servers, and database servers.

Lab Exercise:

1. Draw and explain the J2EE architecture, labeling the layers like the presentation layer, business logic layer, and data layer.

Web Component Development in Java (CGI Programming)

Theory:

- Introduction to CGI (Common Gateway Interface).
- Process, advantages, and disadvantages of CGI programming.

Lab Exercise:

1. Write a simple CGI script using Java to accept user input from a form and display it on a webpage.

Servlet Programming: Introduction, Advantages, and Disadvantages

Theory:

- Introduction to servlets and how they work.
- Advantages and disadvantages compared to other web technologies.

Lab Exercise:

1. Write a simple Java servlet that accepts parameters from a user and displays a response.
2. Discuss the advantages of using servlets over CGI.

Servlet Versions, Types of Servlets

Theory:

- History of servlet versions.
- Types of servlets: Generic and HTTP servlets.

Lab Exercise:

1. Create a Java servlet program using both `GenericServlet` and `HttpServlet` and compare their implementation.

Difference between HTTP Servlet and Generic Servlet

Theory:

- Detailed comparison between `HttpServlet` and `GenericServlet`.

Lab Exercise:

1. Write a program using `HttpServlet` to handle HTTP-specific requests like `GET` and `POST`.

Servlet Life Cycle

Theory:

- Explanation of the servlet life cycle: `init()`, `service()`, and `destroy()` methods.

Lab Exercise:

1. Write a servlet program and override all life cycle methods to log messages when each method is called.

Creating Servlets and Servlet Entry in web.xml

Theory:

- How to create servlets and configure them using `web.xml`.

Lab Exercise:

1. Create a servlet and configure it in `web.xml` for deployment.

Logical URL and ServletConfig Interface

Theory:

- Explanation of logical URLs and their use in servlets.
- Overview of `ServletConfig` and its methods.

Lab Exercise:

1. Write a servlet that uses `ServletConfig` to fetch initialization parameters.

RequestDispatcher Interface: Forward and Include Methods

Theory:

- Explanation of `RequestDispatcher` and the `forward()` and `include()` methods.

Lab Exercise:

1. Create a login form in JSP, send the data to a servlet, and use `RequestDispatcher` to forward or include a response based on input validity.

ServletContext Interface and Web Application Listener

Theory:

- Introduction to `ServletContext` and its scope.
- How to use web application listeners for lifecycle events.

Lab Exercise:

1. Use `ServletContext` to share data across multiple servlets.
2. Create a web application listener that logs application start and stop events.

Practical Example 1: Fetch Data Using ServletConfig

Lab Exercise:

1. Write a servlet to fetch and display initialization parameters from `web.xml` using `ServletConfig`.

Practical Example 2: Fetch Data Using ServletContext

Lab Exercise:

1. Create multiple servlets that fetch shared data from `web.xml` using `ServletContext`.

Practical Example 3: JSP-Servlet Registration Form with RequestDispatcher

Lab Exercise:

1. Create a registration form in JSP.
2. Send form data to a servlet, process it, and forward the response back to a JSP using `RequestDispatcher`.

Java Filters: Introduction and Filter Life Cycle

Theory:

- What are filters in Java and when are they needed?
- Filter lifecycle and how to configure them in `web.xml`.

Lab Exercise:

1. Implement a filter to perform server-side validation of user input.

Practical Example: Server-Side Validation Using Filters

Lab Exercise:

1. Write a filter that checks whether form input fields are empty. If they are, forward back to the input form; otherwise, proceed with the request.

JSP Basics: JSTL, Custom Tags, Scriplets, and Implicit Objects

Theory:

- Introduction to JSP and its key components: JSTL, custom tags, scriplets, and implicit objects.

Lab Exercise:

1. Create a JSP page that uses JSTL to iterate through a list, display scriplets, and access implicit objects.

Session Management and Cookies

Theory:

- Overview of session management techniques: cookies, hidden form fields, URL rewriting, and sessions.
- How to track user sessions in web applications.

Lab Exercise:

1. Implement a login system in JSP and servlet that uses cookies and session tracking to manage user authentication.

Module 9) Java – Software Design Patter and Project

Software Design Patterns and Project (MVC + DAO)

Theory:

- **Introduction to Software Design Patterns**:
  - Definition and purpose of design patterns.
  - Classification: Creational, Structural, and Behavioral patterns.
  - Examples of popular patterns: Singleton, Factory, Observer, Decorator, etc.
- **Introduction to MVC Pattern**:
  - Model-View-Controller (MVC) architecture explained.
  - Separation of concerns and how MVC helps in structuring applications.
- **Introduction to Data Access Object (DAO)**:
  - Purpose of the DAO pattern in decoupling data access logic from business logic.
  - How DAO works in combination with MVC to interact with databases.

Lab Exercise:

1. **Build a simple web application using MVC + DAO**:
   - **Step 1**: Create a simple CRUD web application for user management (register, login, update profile, delete user).
   - **Step 2**: Implement DAO pattern to handle database interactions (e.g., for MySQL database).
   - **Step 3**: Follow the MVC pattern:
     - Model: Contains business logic and DAO.
     - View: JSP files for the user interface.
     - Controller: Java servlets to handle requests and manage responses.

---

2. Session Management (Session, Cookie, Hidden Form Field, URL Rewriting)

Theory:

- **Session Management Overview**:
  - Why session management is essential in web applications.
  - Difference between client-side and server-side session management.
- **Session**:
  - Definition of a session and its importance in tracking user activity.
  - How to create, retrieve, and destroy sessions using Java servlets.
- **Cookies**:
  - What cookies are and how they store small amounts of data on the client-side.
  - Creating, reading, updating, and deleting cookies in Java servlets.
- **Hidden Form Fields**:
  - Explanation of hidden form fields and their role in passing data between pages.
- **URL Rewriting**:
  - How URL rewriting can be used to track sessions when cookies are disabled.

Lab Exercise:

1. **Session Management in Web Application**:
    - **Step 1**: Create a login page in JSP.
    - **Step 2**: Use a session to track the logged-in user and display a welcome page with their details.
    - **Step 3**: Implement logout functionality that invalidates the session.
2. **Cookie Implementation**:
    - **Step 1**: Store the user's preferences (e.g., theme) in a cookie.
    - **Step 2**: On subsequent visits, read the cookie and apply the stored preferences to the web page.
3. **Hidden Form Fields**:
    - **Step 1**: Create a multi-step form for user registration.
    - **Step 2**: Pass data between forms using hidden fields without using sessions.
4. **URL Rewriting**:
    - **Step 1**: Implement URL rewriting to maintain the session for a user in case cookies are disabled.

---

3. Project Covering Topics:

3.1. Template Integration

Theory:

- What is template integration in web applications.
- Importance of using pre-built templates for faster UI development.

Lab Exercise:

1. **Integrate a Template in Your Web Application**:
    - Download a free HTML/CSS template from a website (e.g., Bootstrap template).
    - Integrate the template into your MVC project to enhance the front-end design.

3.2. Image Upload/Download

Theory:

- Steps to upload and download files in Java web applications.
- Explanation of the multipart request and handling file uploads using `MultipartConfig`.

Lab Exercise:

1. **Image Upload/Download Functionality**:

- **Step 1**: Create a JSP form to upload an image file.
- **Step 2**: Write a servlet to handle the file upload and store the image in a designated folder on the server.
- **Step 3**: Implement a servlet to list and download stored images by retrieving the files from the server.

3.3. Mail Integration

Theory:

- How to send emails from a Java web application using JavaMail API.
- Explanation of SMTP and how it's used for sending emails.

Lab Exercise:

1. **Integrate Email Functionality in the Project**:
   - **Step 1**: Create a registration form.
   - **Step 2**: After successful registration, send a confirmation email to the user using the JavaMail API.

3.4. OTP via Mail Integration

Theory:

- Introduction to OTP (One-Time Password) and its importance in enhancing security.
- How to generate and send OTP via email for verification purposes.

Lab Exercise:

1. **OTP Verification**:
   - **Step 1**: Create a registration form with an email field.
   - **Step 2**: Generate an OTP upon form submission and send it to the provided email address.
   - **Step 3**: Create a form to enter the OTP and verify the user's email before allowing account creation.

3.5. Online Payment Integration

Theory:

- Introduction to online payment gateways (e.g., PayPal, Stripe).

- How to integrate payment gateways into web applications.

Lab Exercise:

1. **Payment Gateway Integration**:
    - **Step 1**: Register for a sandbox account with a payment provider (e.g., PayPal Sandbox).
    - **Step 2**: Implement a checkout page for product purchases and integrate it with the payment gateway.

3.6. AJAX

Theory:

- Introduction to AJAX and its role in improving the user experience by enabling asynchronous requests.
- Explanation of how AJAX works in combination with JavaScript and the server.

Lab Exercise:

1. **Implement AJAX in Web Application**:
    - **Step 1**: Create a form for live username validation using AJAX.
    - **Step 2**: When a user enters their username, send an asynchronous request to the server to check if the username is available.
    - **Step 3**: Display the result on the page without refreshing the form.

Module 10) Java – Hibernate Framework

1. Introduction to Hibernate Architecture

Theory:

- **What is Hibernate?**:
  - Definition and purpose of Hibernate as an ORM (Object Relational Mapping) tool.
  - Comparison between Hibernate and JDBC.
  - Why use Hibernate? (Advantages: Database independence, automatic table creation, HQL, etc.)
- **Hibernate Architecture**:
  - Explanation of the Hibernate architecture components:
    - **SessionFactory**: Configuration of Hibernate and creation of sessions.
    - **Session**: The main interface between the Java application and the database.
    - **Transaction**: Handling database transactions in Hibernate.
    - **Query**: Writing HQL (Hibernate Query Language) queries to interact with the database.
    - **Criteria**: Criteria API for building dynamic queries.
  - How Hibernate works internally from loading configuration files to executing queries.

Lab Exercise:

1. **Setting Up Hibernate in a Project**:
   - **Step 1**: Download the required Hibernate dependencies (e.g., Hibernate Core, Hibernate EntityManager, Hibernate Validator, and MySQL Connector).
   - **Step 2**: Create a Hibernate configuration file (`hibernate.cfg.xml`) to set up the connection to a MySQL database.
   - **Step 3**: Write a simple Java application to establish a session with Hibernate and perform a basic operation (e.g., inserting data into a table).

---

2. Hibernate Relationships (One-to-One, One-to-Many, Many-to-One, Many-to-Many)

Theory:

- **Object Relationships in Hibernate**:
  - How Hibernate manages relationships between Java objects and database tables.
  - Overview of the different types of relationships:
    - **One-to-One Relationship**:
      - A single instance of an entity is related to a single instance of another entity.
    - **One-to-Many Relationship**:
      - One entity can have multiple related entities.
    - **Many-to-One Relationship**:
      - Many entities are associated with a single entity.
    - **Many-to-Many Relationship**:

        - Multiple instances of an entity are associated with multiple instances of another entity.
- **Mapping Relationships in Hibernate**:
    - How to map relationships in Hibernate using annotations like `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`.
    - The concept of owning and inverse sides in relationships.
    - Cascade types and how they affect related entities.

Lab Exercise:

1. **One-to-One Relationship**:
    - **Step 1**: Create two entity classes, e.g., `User` and `Profile`, where each user has one profile.
    - **Step 2**: Map the relationship using `@OneToOne` annotation in Hibernate.
    - **Step 3**: Write a program to save and retrieve a user and its profile using Hibernate.
2. **One-to-Many Relationship**:
    - **Step 1**: Create two entity classes, e.g., `Author` and `Book`, where one author can have multiple books.
    - **Step 2**: Map the relationship using `@OneToMany` and `@ManyToOne` annotations.
    - **Step 3**: Write a program to add multiple books for an author and retrieve the author's details along with their books.
3. **Many-to-Many Relationship**:
    - **Step 1**: Create two entity classes, e.g., `Student` and `Course`, where a student can enroll in multiple courses, and a course can have multiple students.
    - **Step 2**: Use the `@ManyToMany` annotation to map the relationship and create a join table.
    - **Step 3**: Write a program to assign multiple courses to students and retrieve student-course details.

---

3. Hibernate CRUD Example

Theory:

- **Understanding CRUD Operations in Hibernate**:
    - **Create (Insert)**: How to use Hibernate to insert records into a database.
    - **Read (Select)**: Fetching data from the database using Hibernate.
    - **Update**: Modifying existing records in the database.
    - **Delete**: Removing records from the database.
- **Writing HQL (Hibernate Query Language)**:
    - Basics of HQL and how it differs from SQL.
    - How to perform CRUD operations using HQL.
    - Introduction to the `Criteria` API for dynamic queries.

Lab Exercise:

1. **Create (Insert) Operation**:

   - **Step 1**: Define a simple entity class, e.g., `Employee`, with fields like `id`, `name`, `department`, and `salary`.
   - **Step 2**: Write a Hibernate program to insert employee records into a database table using `Session.save()` method.
   - **Step 3**: Verify the inserted data by querying the database directly.
2. **Read (Select) Operation**:
   - **Step 1**: Write a Hibernate query to retrieve all employees from the database using `Session.get()` or HQL.
   - **Step 2**: Display the retrieved employee data in the console.
3. **Update Operation**:
   - **Step 1**: Write a Hibernate program to update the salary of an employee.
   - **Step 2**: Use `Session.update()` method to modify an existing record.
   - **Step 3**: Fetch and verify that the employee's salary has been updated in the database.
4. **Delete Operation**:
   - **Step 1**: Write a Hibernate program to delete an employee from the database.
   - **Step 2**: Use `Session.delete()` method to remove a record.
   - **Step 3**: Verify the deletion by querying the database.

Practical Project Example:

Create a simple Employee Management System using Hibernate to perform CRUD operations and manage employee details. The system should support:

- **Inserting** a new employee record.
- **Viewing** all employee records.
- **Updating** employee details (e.g., changing department, salary).
- **Deleting** an employee.

Incorporate Hibernate relationships such as:

- **One-to-One**: Each employee has one profile (e.g., employee details and profile picture).
- **One-to-Many**: One department can have many employees.
- **Many-to-Many**: Employees can work on multiple projects, and projects can have multiple employees assigned.

Module 9) Java – Spring

1. Introduction to Spring Framework

Theory:

- **What is Spring Framework?**
  - Overview of the Spring Framework and its purpose in Java development.
  - Key features of Spring:
    - Inversion of Control (IoC)
    - Dependency Injection (DI)
    - Aspect-Oriented Programming (AOP)
    - Transaction Management
    - Spring's flexibility for creating both web and non-web applications.
- **Spring Architecture**:
  - Overview of the core components of the Spring Framework:
    - **Core Container**: IoC and DI
    - **Spring AOP**: Aspect-Oriented Programming
    - **Spring ORM**: Integrating Spring with ORM frameworks (e.g., Hibernate, JPA)
    - **Spring Web**: Web framework for creating Java web applications.
    - **Spring MVC**: Model-View-Controller framework for building web applications.

Lab Exercise:

1. **Setting up a Spring Project**:
   - **Step 1**: Install and configure Spring dependencies using Maven or Gradle.
   - **Step 2**: Create a basic Spring application.
   - **Step 3**: Configure a simple XML or annotation-based Spring application with one bean and test it by loading the Spring application context.

---

2. BeanFactory and ApplicationContext

Theory:

- **BeanFactory vs. ApplicationContext**:
  - What is **BeanFactory**?:
    - A simple container for managing Spring beans.
    - Pros and cons of using BeanFactory.
  - What is **ApplicationContext**?:
    - A more advanced container that includes features like event propagation, declarative mechanisms, and AOP support.
  - Differences between **BeanFactory** and **ApplicationContext** (e.g., lazy initialization in BeanFactory vs. eager initialization in ApplicationContext).
- **Spring Beans**:
  - Definition of a bean in Spring.

- Scope of beans: Singleton, Prototype, Request, Session.
- Bean lifecycle: Initialization and destruction of beans.

Lab Exercise:

1. **Using BeanFactory and ApplicationContext**:
   - **Step 1**: Create a Spring configuration file (`beans.xml`) to define a few simple beans.
   - **Step 2**: Write Java code to load the beans using **BeanFactory** and display the bean properties.
   - **Step 3**: Modify the code to load the same beans using **ApplicationContext** and discuss the difference.
2. **Bean Scopes**:
   - **Step 1**: Configure beans with different scopes (e.g., Singleton and Prototype) in the `beans.xml` file.
   - **Step 2**: Write Java code to demonstrate the effect of different bean scopes by retrieving beans multiple times and checking if the same instance is returned.

---

3. Container Concepts in Spring

Theory:

- **Spring IoC (Inversion of Control)**:
  - Understanding IoC and how Spring uses it to manage object creation and dependencies.
  - Benefits of IoC in application design (loose coupling, modularity, and testability).
- **Dependency Injection (DI)**:
  - Types of Dependency Injection:
    - Constructor-based Dependency Injection.
    - Setter-based Dependency Injection.
  - Advantages of DI in Spring.

Lab Exercise:

1. **Constructor and Setter Dependency Injection**:
   - **Step 1**: Create a Spring configuration file and define two beans with dependencies.
   - **Step 2**: Demonstrate constructor-based DI by wiring dependencies via the constructor.
   - **Step 3**: Demonstrate setter-based DI by wiring dependencies via setter methods.
   - **Step 4**: Test the configuration by retrieving the beans and checking the injection.
2. **Configuring IoC in XML and Annotations**:
   - **Step 1**: Define beans in XML to implement DI.
   - **Step 2**: Modify the same beans to use annotations (`@Autowired`, `@Qualifier`) for DI.

---

4. Spring Data JPA Template

Theory:

- **What is Spring Data JPA?**:
  - Introduction to Spring Data JPA and how it simplifies interaction with databases.
  - Explanation of JPA (Java Persistence API) and its role in ORM (Object Relational Mapping).
  - Benefits of using Spring Data JPA over manual SQL queries.
- **Spring Data JPA Components**:
  - **Repositories**: How Spring Data JPA auto-generates repository implementations.
  - **Entities**: Mapping Java objects to database tables using JPA annotations.
  - **Query Methods**: Creating custom queries using method naming conventions (e.g., `findById`, `findByName`).

Lab Exercise:

1. **Basic CRUD Operations with Spring Data JPA**:
   - **Step 1**: Set up a Spring Boot project with Spring Data JPA and a MySQL database.
   - **Step 2**: Create an entity class (`Employee`) with fields like `id`, `name`, `department`.
   - **Step 3**: Create a repository interface extending `JpaRepository`.
   - **Step 4**: Write a service class to perform basic CRUD operations (Insert, Update, Delete, Select) on the `Employee` entity.
   - **Step 5**: Test the CRUD operations using a REST controller or unit tests.
2. **Custom Queries Using Spring Data JPA**:
   - **Step 1**: Create a repository interface with custom query methods (e.g., `findByDepartment(String department)`).
   - **Step 2**: Implement the repository and perform database queries based on method names.

---

5. Spring MVC

Theory:

- **What is Spring MVC?**:
  - Overview of the MVC (Model-View-Controller) design pattern.
  - Explanation of the Spring MVC framework and how it simplifies web development.
- **Spring MVC Components**:
  - **Controller**: Handles HTTP requests and returns a response.
  - **Model**: Holds the data to be displayed on the view.
  - **View**: Renders the data from the model in a user-friendly format (e.g., JSP, Thymeleaf).
  - **DispatcherServlet**: Central servlet in Spring MVC that manages the request flow.
- **Request Mapping in Spring MVC**:
  - Using `@RequestMapping`, `@GetMapping`, and `@PostMapping` annotations to map HTTP requests to controller methods.

- Path variables, request parameters, and form handling.

Lab Exercise:

1. **Building a Simple Spring MVC Application**:
   - **Step 1**: Set up a Spring MVC project and configure the `DispatcherServlet` in `web.xml`.
   - **Step 2**: Create a simple controller class with `@RequestMapping` to handle a basic request (e.g., `/welcome`).
   - **Step 3**: Create a view (e.g., JSP or Thymeleaf) to display a welcome message to the user.
   - **Step 4**: Test the application by accessing the controller endpoint and displaying the view.
2. **Handling Forms in Spring MVC**:
   - **Step 1**: Create a Spring MVC form for user registration.
   - **Step 2**: Create a controller method to handle form submission and capture user data.
   - **Step 3**: Validate the form inputs using Spring's form validation (`@Valid`, `BindingResult`).
   - **Step 4**: Display validation errors on the view if inputs are invalid.

Project Example for Spring MVC + Spring Data JPA:

Employee Management System:

- Build a basic web application using **Spring MVC** for handling requests and **Spring Data JPA** for CRUD operations.
- Key Features:
  1. **User Registration**: Create a form for registering a new employee.
  2. **View Employees**: Display all employees from the database on a webpage.
  3. **Update Employee**: Provide an option to update employee details.
  4. **Delete Employee**: Allow the deletion of employee records.
  5. **Search Employees**: Add functionality to search for employees by name or department.

Module 10) Java – Spring Boot

1. Introduction to STS (Spring Tool Suite)

Theory:

- **What is Spring Tool Suite (STS)?**
  - Overview of STS: An Eclipse-based IDE for developing Spring applications.
  - Key features and benefits of using STS, including built-in support for Spring Boot, easy dependency management, and a robust debugging environment.
- **Installation and Setup**:
  - Step-by-step guide on how to download, install, and configure STS for Java/Spring development.
  - Overview of the interface, how to create a Spring Boot project, and the workspace organization.

Lab Exercise:

1. **Setting up STS and Creating a Simple Spring Boot Application**:
   - **Step 1**: Install and configure STS.
   - **Step 2**: Create a new Spring Boot project in STS.
   - **Step 3**: Configure dependencies (Spring Web, Spring Data JPA, etc.) via Maven or Gradle.
   - **Step 4**: Write a simple controller and run the application to display "Hello, Spring!" on the browser.

---

2. Spring MVC (Model-View-Controller)

Theory:

- **Spring MVC Overview**:
  - Introduction to the MVC design pattern and how it is implemented in Spring.
  - Explanation of core components: Controller, Model, and View.
- **Template Integration**:
  - Using templating engines like Thymeleaf or JSP in Spring MVC applications.
  - How template engines help in creating dynamic web pages and separating concerns.
- **CRUD Operations**:
  - Implementing basic Create, Read, Update, and Delete functionality in a Spring MVC application.
  - Flow of data between the view, controller, and model.
- **Form Validation**:
  - Introduction to form validation in Spring MVC using annotations like `@Valid` and `@NotNull`.
  - Validating user input and handling validation errors.
- **Pagination**:
  - Implementing pagination in Spring MVC to handle large datasets.

- Using `Pageable` and `Page` interfaces in Spring Data JPA.

Lab Exercise:

1. **Template Integration**:
   - **Step 1**: Create a Spring MVC project and integrate Thymeleaf (or JSP) as the view layer.
   - **Step 2**: Create a simple template to display dynamic content (e.g., a list of users).
   - **Step 3**: Configure the template to accept data from the Spring controller and display it on the view.
2. **CRUD Operations with Spring MVC**:
   - **Step 1**: Set up a Spring Boot project with Spring MVC and Spring Data JPA.
   - **Step 2**: Create an entity class `Product` with fields `id`, `name`, `price`, and `description`.
   - **Step 3**: Implement the CRUD operations (Create, Read, Update, Delete) in the controller, using a service layer and repository.
   - **Step 4**: Create views for adding, listing, editing, and deleting products.
3. **Form Validation**:
   - **Step 1**: Create a form for user registration.
   - **Step 2**: Add validation to the form fields (e.g., name, email) using `@NotEmpty`, `@Email`, and other validation annotations.
   - **Step 3**: Implement validation handling in the controller and display error messages on the view when validation fails.
4. **Pagination**:
   - **Step 1**: Create a service to fetch data in a paginated format using `Pageable`.
   - **Step 2**: Implement pagination in the controller and view to display large datasets (e.g., a list of products or users) across multiple pages.
   - **Step 3**: Create navigation controls to move between pages.

---

3. Aspect-Oriented Programming (AOP)

Theory:

- **What is AOP (Aspect-Oriented Programming)?**
  - Definition of AOP and its importance in separating cross-cutting concerns (logging, security, transaction management).
  - Key components in AOP:
    - **Aspect**: A module that encapsulates cross-cutting concerns.
    - **Joinpoint**: A point in the program where the aspect is applied.
    - **Advice**: The action taken by an aspect at a particular joinpoint (Before, After, Around).
    - **Pointcut**: An expression to define where advice should be applied.

Lab Exercise:

1. **Logging Aspect in Spring AOP**:
   - **Step 1**: Set up a Spring Boot project with AOP support.

- **Step 2**: Create an `Aspect` for logging method execution times.
- **Step 3**: Implement `@Before`, `@After`, and `@Around` advices to log details before and after method execution in a service class.
- **Step 4**: Test the aspect by calling a method from the service class and checking the logs for method execution details.

---

4. Spring Security

Theory:

- **Introduction to Spring Security**:
  - Overview of Spring Security, its purpose, and how it secures web applications.
  - Key features: Authentication and Authorization, Security Filters, and Form-based login.
- **Role-Based Authentication**:
  - How to define roles (e.g., USER, ADMIN) and restrict access to specific URLs or methods based on user roles.
  - Securing endpoints using `@Secured` or `@PreAuthorize`.
- **OAuth2 Authentication**:
  - Introduction to OAuth2 and how it is used for third-party authentication (Google, Facebook).
  - Explanation of OAuth2 flows: Authorization Code Grant, Implicit Grant, etc.
- **Token-Based Authentication (JWT)**:
  - Introduction to token-based authentication using JSON Web Tokens (JWT).
  - Explanation of the authentication process: token generation, validation, and secure access to protected resources.

Lab Exercise:

1. **Role-Based Authentication**:
   - **Step 1**: Set up a Spring Boot project with Spring Security.
   - **Step 2**: Define roles (USER, ADMIN) and create a simple login form.
   - **Step 3**: Secure specific URLs (e.g., `/admin`, `/user`) and restrict access based on roles.
   - **Step 4**: Test the application by logging in with different users and checking if the correct restrictions are applied.
2. **OAuth2 Integration**:
   - **Step 1**: Set up OAuth2 login with Google or Facebook in a Spring Boot application.
   - **Step 2**: Configure the application to redirect to Google/Facebook for authentication.
   - **Step 3**: Once authenticated, display the user's information (name, email) on the dashboard.
3. **Token-Based Authentication (JWT)**:
   - **Step 1**: Implement JWT-based authentication in a Spring Boot REST API.
   - **Step 2**: Create an endpoint for user login and generate a JWT token upon successful authentication.
   - **Step 3**: Implement a filter to validate the JWT token for each request to protected resources.

- **Step 4**: Test the application by logging in, obtaining a token, and accessing secured endpoints using the token.

Project Example: E-Commerce Web Application Using Spring MVC, AOP, and Security

Key Features:

- **User Registration and Login**: Implement user registration with form validation and Spring Security.
- **Role-based Authorization**: Admin can manage products, and users can view and purchase products.
- **CRUD Operations**: Admin can create, update, delete, and view products.
- **Aspect-Oriented Programming**: Implement logging for product management operations (create, update, delete).
- **Pagination**: Display a paginated list of products for users.
- **OAuth2 Authentication**: Allow users to sign in via Google or Facebook.
- **JWT Authentication**: Implement JWT for securing REST API endpoints for managing products.

Module 11) Java – Spring Webservices

1. Introduction to Web Services

Theory:

- **What are Web Services?**
  - Definition of web services and their importance in enabling communication between different applications over the internet.
  - Types of Web Services:
    - **SOAP (Simple Object Access Protocol)**
    - **REST (Representational State Transfer)**
- **Advantages of Web Services**:
  - Platform and language independence.
  - Integration across diverse systems.
  - Enables microservices architecture.

Lab Exercise:

1. **Create a Simple Web Service**:
   - **Step 1**: Set up a simple RESTful web service using Spring Boot.
   - **Step 2**: Create a REST endpoint `/greeting` that returns a simple greeting message (e.g., “Hello, World!”).
   - **Step 3**: Test the endpoint using Postman or Curl to verify it returns the expected response.

---

2. Basics of REST APIs

Theory:

- **What is REST (Representational State Transfer)?**
  - Overview of REST principles: statelessness, resource-based URLs, use of HTTP methods (GET, POST, PUT, DELETE), and status codes.
  - Key REST concepts:
    - **Resources**: Everything is treated as a resource.
    - **URI**: Uniform Resource Identifiers for identifying resources.
    - **Stateless Communication**: Each request from a client to the server must contain all the information needed to understand and process the request.
- **HTTP Methods**:
  - **GET**: Retrieve data.
  - **POST**: Submit data.
  - **PUT**: Update data.
  - **DELETE**: Remove data.

Lab Exercise:

1. **Create a RESTful API for a Student Resource**:
   - **Step 1**: Set up a Spring Boot project with Spring Web dependency.
   - **Step 2**: Create a `Student` entity with fields `id`, `name`, `email`, and `course`.
   - **Step 3**: Implement REST endpoints for CRUD operations:
     - **GET** `/students`: Retrieve a list of students.
     - **POST** `/students`: Add a new student.
     - **PUT** `/students/{id}`: Update an existing student's details.
     - **DELETE** `/students/{id}`: Delete a student.
   - **Step 4**: Test the endpoints using Postman or any REST client.

---

3. Spring MVC (Model-View-Controller)

Theory:

- **Spring MVC Overview**:
  - Explanation of the MVC design pattern: Model, View, and Controller.
  - How Spring MVC handles incoming web requests and maps them to the correct controller.
- **Controller and View**:
  - Creating a controller to handle user requests.
  - Using a view template engine (e.g., Thymeleaf) to render dynamic data.

Lab Exercise:

1. **Create a Spring MVC Web Application**:
   - **Step 1**: Set up a Spring Boot project with Spring Web and Thymeleaf.

- **Step 2**: Create a simple controller that handles a GET request and returns a view.
- **Step 3**: Create a view template using Thymeleaf to display a list of students passed from the controller.

---

4. Aspect-Oriented Programming (AOP)

Theory:

- **What is AOP (Aspect-Oriented Programming)?**
  - Overview of AOP and how it helps in separating cross-cutting concerns (e.g., logging, security, transaction management).
  - Key AOP terms:
    - **Aspect**: Module encapsulating cross-cutting concerns.
    - **Advice**: The action taken by an aspect (Before, After, or Around).
    - **Joinpoint**: Point in the execution of the program where the aspect is applied.
    - **Pointcut**: Expression that defines where the advice should be applied.

Lab Exercise:

1. **Implement Logging Aspect Using AOP**:
   - **Step 1**: Set up a Spring Boot project with AOP dependency.
   - **Step 2**: Create an Aspect class that logs the method execution time.
   - **Step 3**: Use `@Before` and `@After` annotations to log the execution of specific methods in a service class.
   - **Step 4**: Test the logging aspect by calling methods in the service class and checking the logs.

---

5. Spring REST (CRUD API, Pagination, Fetching from Multiple Tables, Image Upload/Download)

Theory:

- **Spring REST Overview**:
  - Introduction to creating RESTful services in Spring Boot.
  - Use of `@RestController` to create REST APIs.
  - Handling HTTP requests and returning JSON or XML responses.
- **Pagination**:
  - Introduction to pagination in REST APIs to handle large datasets.
  - Use of `Pageable` and `Page` interfaces from Spring Data JPA for pagination support.
- **CRUD Operations**:
  - Create, Read, Update, Delete (CRUD) operations using Spring Data JPA.
- **Fetching Data from Multiple Tables**:
  - Use of JPA relationships (`@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany`) to retrieve related data from multiple tables.

- **Image Upload/Download**:
    - Handling file upload and download in a Spring REST API.

Lab Exercise:

1. **CRUD API with Pagination**:
    - **Step 1**: Set up a Spring Boot project with Spring Data JPA and Spring Web.
    - **Step 2**: Create two entities, `Student` and `Course`, with a **many-to-one** relationship between them.
    - **Step 3**: Implement CRUD operations for the `Student` entity with endpoints for adding, updating, retrieving, and deleting students.
    - **Step 4**: Implement pagination on the GET endpoint to retrieve a paginated list of students using the `Pageable` interface.
    - **Step 5**: Test the API using Postman or any REST client.
2. **Fetching Data from Multiple Tables**:
    - **Step 1**: Extend the above lab by fetching a list of students enrolled in a particular course.
    - **Step 2**: Implement a GET endpoint to fetch students based on the course ID.
    - **Step 3**: Return a list of students enrolled in the course, showing the relationship between the two tables.
3. **Image Upload/Download in REST API**:
    - **Step 1**: Implement an API endpoint that allows users to upload an image file.
    - **Step 2**: Store the uploaded image in the file system or database (e.g., as a `BLOB`).
    - **Step 3**: Create another API endpoint to download and display the image file.
    - **Step 4**: Test the image upload and download functionality using Postman or any REST client.

Project Example: Bookstore Application Using Spring REST, AOP, and Pagination

Features:

- **Book Management**: Implement CRUD operations for books.
- **Author Management**: CRUD operations for authors, with a relationship between books and authors (One-to-Many).
- **Pagination**: Display paginated lists of books on the frontend.
- **AOP Logging**: Implement logging for the CRUD operations on books and authors.
- **Image Upload/Download**: Allow users to upload book cover images and download them.
- **Search Functionality**: Implement a search API to find books by title or author.

Module 14) Java – Micro services with Spring Boot, Spring Cloud

1. Microservices with Spring Boot and Spring Cloud

Theory:

- **What are Microservices?**
  - Definition and characteristics of Microservices architecture.
  - Key principles: Decoupled services, scalability, independent deployment.
- **Advantages of Microservices Over Monolithic Architecture**:
  - Scalability: Independent scaling of services.
  - Fault Isolation: Issues in one service do not affect others.
  - Flexibility: Different technologies can be used in different services.
  - Faster Deployment: Continuous delivery and deployment pipelines are easier.
- **Components of Microservices Architecture**:
  - **API Gateway**: Routes and load balances requests to microservices.
  - **Service Registry (Eureka)**: Keeps track of services and their locations.
  - **Circuit Breaker**: Manages service failures.
  - **Load Balancer**: Distributes requests across services.

Lab Exercise:

1. **Create a Simple Microservice with Spring Boot**:
   - **Step 1**: Set up a Spring Boot application for a simple microservice (e.g., `UserService`).
   - **Step 2**: Implement basic CRUD operations for the `UserService` using RESTful APIs.
   - **Step 3**: Test the APIs locally using Postman or Curl.

---

2. Introduction to Microservice Architecture

Theory:

- **Microservice vs. Monolithic Architecture**:
  - **Monolithic Architecture**: All functionalities reside in one large application.
  - **Microservices**: Applications are split into independent services.
- **Key Characteristics**:
  - **Decentralization**: Each microservice has its own database.
  - **Inter-Service Communication**: Services communicate using lightweight protocols like HTTP or messaging systems like RabbitMQ.

Lab Exercise:

1. **Convert a Monolithic Application into Microservices**:
   - **Step 1**: Take a sample monolithic application (e.g., a shopping app with user management and product management).

- **Step 2**: Split the monolithic app into two microservices: `UserService` and `ProductService`.
- **Step 3**: Set up communication between the services using REST.

---

3. Developing and Deploying a Microservice Application Locally

Theory:

- **Steps to Build a Microservice**:
  - Develop each service independently.
  - Use Spring Boot for microservice development.
  - Package and deploy each service using Docker or directly on localhost.

Lab Exercise:

1. **Deploy Two Microservices Locally**:
   - **Step 1**: Create two microservices (`UserService` and `OrderService`) using Spring Boot.
   - **Step 2**: Set up the services to run on different ports (e.g., `UserService` on port 8081 and `OrderService` on port 8082).
   - **Step 3**: Test communication between the services using REST APIs locally.
2. **Optional**:
   - **Step 4**: Package the services as Docker containers and run them using Docker Compose.

---

4. Introduction to Service Discovery: Eureka Server

Theory:

- **Service Discovery**:
  - In microservices, each service may start and stop dynamically, so a **Service Registry** is essential to keep track of service instances.
- **What is Eureka?**
  - **Eureka** is a Service Registry from Netflix that allows services to register themselves and discover other services.
- **Eureka Server and Eureka Client**:
  - **Eureka Server**: Acts as the registry for services.
  - **Eureka Client**: Registers itself with the Eureka Server and discovers other services.

Lab Exercise:

1. **Set up a Eureka Server**:
   - **Step 1**: Create a Spring Boot application and add the Eureka Server dependency.
   - **Step 2**: Enable Eureka Server in the application using `@EnableEurekaServer`.

    - **Step 3**: Run the Eureka Server and check the Eureka dashboard (default on `http://localhost:8761`).
2. **Register a Service with Eureka**:
    - **Step 1**: Create a simple Spring Boot microservice (`OrderService`) and add the Eureka Client dependency.
    - **Step 2**: Enable Eureka Client in the service using `@EnableEurekaClient`.
    - **Step 3**: Register the service with the Eureka Server and check if it is listed in the Eureka dashboard.

---

5. Client-Side and Server-Side Discovery Patterns

Theory:

- **Client-Side Discovery**:
    - The client is responsible for service discovery by interacting with the Eureka Server and finding the instances of a particular service.
- **Server-Side Discovery**:
    - The client makes a request to an API Gateway or Load Balancer, which then forwards the request to the appropriate service.

Lab Exercise:

1. **Client-Side Service Discovery**:
    - **Step 1**: Create a microservice (`UserService`) and register it with the Eureka Server.
    - **Step 2**: Create another microservice (`OrderService`) that uses a `RestTemplate` to discover `UserService` from Eureka and make a request to its API.
2. **Server-Side Discovery**:
    - **Step 1**: Set up an API Gateway (e.g., Spring Cloud Gateway) that forwards requests to `UserService` and `OrderService`.
    - **Step 2**: Use Eureka for server-side service discovery, where the gateway fetches the available instances from the Eureka Server.

---

6. Load Balancing Configuration

Theory:

- **What is Load Balancing?**
    - Load balancing helps distribute incoming requests across multiple instances of a service to ensure better performance and fault tolerance.
- **Types of Load Balancers**:
    - **Client-Side Load Balancer**: Managed at the client-side (e.g., Ribbon).
    - **Server-Side Load Balancer**: Managed centrally (e.g., API Gateway, Nginx).

Lab Exercise:

1. **Client-Side Load Balancing with Ribbon**:
   - **Step 1**: Create multiple instances of a microservice (e.g., two instances of `UserService` running on different ports).
   - **Step 2**: Enable Ribbon client-side load balancing in another service (`OrderService`).
   - **Step 3**: Use `RestTemplate` to make a call to `UserService` and test if the requests are balanced across both instances.
2. **Server-Side Load Balancing Using Spring Cloud Gateway**:
   - **Step 1**: Set up Spring Cloud Gateway to route requests to multiple instances of `UserService`.
   - **Step 2**: Configure the gateway to load balance the requests between instances.
   - **Step 3**: Test load balancing by sending multiple requests to the gateway and checking the distribution.

Project Example: E-commerce Microservices with Eureka and Load Balancing

Features:

- **Service Registry**: Use Eureka Server to register services (`UserService`, `OrderService`, `ProductService`).
- **API Gateway**: Set up an API Gateway to route traffic to the different services.
- **Load Balancing**: Configure load balancing for services with multiple instances.
- **Database Integration**: Use Spring Data JPA for database interactions in each service (e.g., MySQL or PostgreSQL).
- **Communication**: Use REST APIs for inter-service communication.

Module 19) Debugging Exercises for Problem Solving

1. Simple Arithmetic Calculation (Off-by-One Error)

Description: This program is meant to calculate the sum of the first 10 natural numbers. However, there's an off-by-one error.

```
java
Copy code
public class SumOfNumbers {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i <= 10; i++) {  // Off-by-one error here
            sum += i;
        }
        System.out.println("Sum of first 10 natural numbers is: " + sum);
    }
}
```

Objective:

- Identify the off-by-one error.
- Debug the loop so that it correctly sums the first 10 natural numbers.

Expected Output:
Sum of first 10 natural numbers is: 55

---

2. Array Index Out of Bound

Description: The program is designed to calculate the average of the numbers in an array, but it throws an `ArrayIndexOutOfBoundsException`.

```
java
Copy code
public class ArrayAverage {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int sum = 0;
        for (int i = 0; i <= numbers.length; i++) {  // Off-by-one error
            sum += numbers[i];
        }
        double average = sum / numbers.length;
        System.out.println("Average is: " + average);
    }
}
```

Objective:

- Identify the mistake causing the `ArrayIndexOutOfBoundsException`.
- Fix the error and ensure the program calculates the average correctly.

Expected Output:
Average is: 30.0

---

3. Infinite Loop

Description: The following program should print numbers from 1 to 5, but it runs infinitely due to a logical error in the loop.

```
java
Copy code
public class PrintNumbers {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println(i);
            // Increment is missing here
        }
    }
}
```

Objective:

- Find the cause of the infinite loop.
- Correct the code so it prints numbers from 1 to 5 without running indefinitely.

Expected Output:

```
Copy code
1
2
3
4
5
```

---

4. Null Pointer Exception

Description: The following program should print the length of a string, but it throws a `NullPointerException`.

```
java
Copy code
public class StringLength {
    public static void main(String[] args) {
        String str = null;
        System.out.println("Length of the string is: " + str.length());
    }
}
```

Objective:

- Identify why the program throws a `NullPointerException`.
- Modify the code to avoid the exception and handle the `null` string properly.

Expected Output:
Length of the string is: 0 (or handle it with an appropriate message)

---

5. Incorrect Output Due to Floating-Point Division

Description: The following program tries to calculate the percentage of marks, but the result is incorrect due to integer division.

```
java
Copy code
public class PercentageCalculator {
    public static void main(String[] args) {
        int totalMarks = 450;
        int marksObtained = 375;
        int percentage = (marksObtained / totalMarks) * 100;  // Incorrect 
division
        System.out.println("Percentage: " + percentage + "%");
    }
}
```

Objective:

- Identify the cause of the incorrect output.
- Correct the code to ensure that floating-point division is used for calculating the percentage.

Expected Output:
Percentage: 83.33%

---

6. Logical Error in Prime Number Check

Description: The following program is supposed to check if a number is prime or not, but it incorrectly identifies some composite numbers as prime.

```
java
Copy code
public class PrimeCheck {
    public static void main(String[] args) {
        int number = 15;
        boolean isPrime = true;

        for (int i = 2; i <= number / 2; i++) {
            if (number % i == 0) {
                isPrime = false;
                break;
            }
        }
```

```
        if (isPrime) {
            System.out.println(number + " is a prime number.");
        } else {
            System.out.println(number + " is not a prime number.");
        }
    }
}
```

Objective:

- Identify why the program incorrectly identifies some composite numbers as prime.
- Correct the prime number logic to work for any input.

Expected Output:
15 is not a prime number.

---

7. Wrong Use of Equals for String Comparison

Description: The following program tries to compare two strings for equality, but it gives incorrect results.

```
java
Copy code
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "hello";
        String str2 = new String("hello");

        if (str1 == str2) {
            System.out.println("Strings are equal.");
        } else {
            System.out.println("Strings are not equal.");
        }
    }
}
```

Objective:

- Identify why the comparison gives incorrect results.
- Use the correct method for comparing strings.

Expected Output:
Strings are equal.

---

8. Off-by-One Error in Array Sum

Description: The following program should calculate the sum of elements in an array, but it doesn't add all elements correctly due to an off-by-one error.

```
java
Copy code
public class ArraySum {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int sum = 0;

        for (int i = 0; i < arr.length - 1; i++) {  // Off-by-one error
            sum += arr[i];
        }

        System.out.println("Sum of array elements: " + sum);
    }
}
```

Objective:

- Identify the off-by-one error in the array summation.
- Correct the loop to add all elements of the array.

Expected Output:
Sum of array elements: 15

---

9. Wrong Output for Fibonacci Series

Description: The program should print the first 5 Fibonacci numbers, but it prints incorrect values due to improper handling of the loop variables.

```
java
Copy code
public class Fibonacci {
    public static void main(String[] args) {
        int n1 = 0, n2 = 1, n3;
        System.out.print(n1 + " " + n2);

        for (int i = 2; i <= 5; i++) {
            n3 = n1 + n2;
            System.out.print(" " + n3);
            n1 = n2;  // Incorrect update of n1 and n2
            n2 = n3;
        }
    }
}
```

Objective:

- Identify why the Fibonacci sequence is incorrect.

- Fix the logic to correctly generate the first 5 Fibonacci numbers.

Expected Output:
0 1 1 2 3 5

---

10. Logical Error in Palindrome Check

Description: The following program is supposed to check if a string is a palindrome, but it incorrectly identifies some non-palindromes as palindromes.

```
java
Copy code
public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        String reverse = "";

        for (int i = 0; i <= str.length(); i++) {  // Off-by-one error
            reverse += str.charAt(i);
        }

        if (str.equals(reverse)) {
            System.out.println(str + " is a palindrome.");
        } else {
            System.out.println(str + " is not a palindrome.");
        }
    }
}
```

Objective:

- Identify the off-by-one error in the loop and correct it.
- Ensure the program correctly checks if a string is a palindrome.

Expected Output:
madam is a palindrome.