**Object-Oriented Programming**

**Lab File**

**BTech 4th Semester**

**2024 - 2025**

****

**Submitted To:** Mr.Aryan Gupta

Assistant Professor (SoCS)

**Submitted By:**

Dhruv Shah

B.Tech CSE with specialization in AI/ML and minor in Aerospace Engineering

SAP ID-500118979

Batch – AI/ML 11

ALL CODES ARE AVAILABLE IN THIS GITHUB REPOSITORY

<https://github.com/dhruvshah11/OOPS-LAB-CODES>

**Table of content**

**TABLE OF CONTENTS**

| **Sr. No.** | **Experiment Title** | **Programs Implemented** |
| --- | --- | --- |
| 1 | Introduction to Java | HelloWorld.java, DisplayAddress.java, GradeSheet.java |
|  |  | Key Concepts: Java structure, variables, I/O, JDK-JRE-JVM concepts |
| 2 | Control Structures | CommandLineCalculator.java, DayOfWeek.java, NumberClassifier.java, StudentGrade.java, TriangleArea.java |
|  |  | Key Concepts: if-else, switch, loops, command-line arguments |
| 3 | Arrays and Basic Programs | ArrayCopy.java, Fibonacci.java, PrimeNumbers.java, PatternPrinting.java, SecondLargest.java |
|  |  | Advanced Concepts: 2D arrays, sorting, patterns, series |
| 4 | Classes and Objects | BankAccount.java, Calculator.java, Student.java, Employee.java |
|  |  | Key Concepts: Constructors, static vs instance, access modifiers |
| 5 | Inheritance | EmployeeManager.java, VehicleHierarchy.java, WorkerHierarchy.java, UniversitySystem.java |
|  |  | Advanced OOP: Inheritance types, overriding, abstract classes, super |
| 6 | Package and Access Control | Balance.java, Account.java, FinalDemo.java, LoggerDemo.java |
|  |  | Key Concepts: Packages, access levels, final keyword, logging |
| 7 | Abstract Classes and Interfaces | BankDemo.java, ShapeDemo.java, EmployeeDemo.java, TestPlayer.java |
|  |  | Advanced Concepts: Abstract classes, interfaces, multiple inheritance |
| 8 | Exception Handling | DivisionCalculator.java, FileReadDemo.java, ArrayAccessDemo.java, StudentFileReader.java |
|  |  | Key Concepts: try-catch, throw/throws, custom exceptions |
| 9 | GUI Programming | Calculator.java, LoginForm.java, ToDoList.java, EmployeeManagement.java |
|  |  | Key Concepts: AWT/Swing, layout managers, event handling |
| 10 | Collections | ProductInventory.java, EmployeeArrayList.java, NameDeduplication.java, PrimeNumberChecker.java |
|  |  | Key Concepts: ArrayList, HashSet, Map, utility methods |

### **ACKNOWLEDGMENT**

I would like to express my sincere gratitude to **Mr. Aryan Gupta**, Assistant Professor (SoCS), for his continuous guidance, support, and encouragement throughout the Object-Oriented Programming Lab sessions. His insightful lectures and constructive feedback have helped me gain a solid foundation in Java and object-oriented concepts.

I would also like to thank the **School of Computer Science (SoCS), UPES**, for providing a well-structured and hands-on learning environment that enabled me to explore and implement the practical aspects of OOP.

Finally, I am thankful to my peers, family, and mentors for their motivation, patience, and support during the preparation of this lab file. The successful completion of this lab file is a result of the collective effort, guidance, and teamwork received throughout the semester.

Output

### **INTRODUCTION**

The **Object-Oriented Programming System (OOPS)** Lab is a crucial component of the curriculum for students pursuing Computer Science Engineering with a specialization in Artificial Intelligence and Machine Learning. This lab is designed to provide practical exposure to the core concepts of object-oriented programming using **Java** as the implementation language.

The lab file covers a wide range of topics, including the Java development environment, control structures, arrays, classes and objects, inheritance, polymorphism, exception handling, packages, interfaces, GUI development, and the collection framework. Each experiment is structured to enhance the understanding of theoretical concepts through real-time coding implementations.

This lab aims to:

* Develop problem-solving abilities using OOP principles.
* Strengthen the understanding of encapsulation, inheritance, and polymorphism.
* Explore modular and reusable coding practices.
* Enable the development of user-friendly GUI applications and robust backend logic.

By completing these experiments, I have gained a deeper understanding of writing efficient, maintainable, and scalable object-oriented Java programs. This lab has significantly contributed to my journey as a software developer, preparing me to solve real-world challenges using modern programming paradigms.

# EXPERIMENT – 1

# TITLE: Introduction to Java Environment

1. Explore and understand the role of JDK, JRE and JVM.
2. Install latest available JDK and verify the Java Environment.
3. Understanding JDK, JRE, and JVM
4. JVM (Java Virtual Machine)
5. - The JVM is an abstract machine that provides a runtime environment for Java bytecode to execute.
6. - It is platform-dependent (different for Windows, Linux, Mac).
7. - Main responsibilities:
8. - Loads, verifies, and executes Java bytecode
9. - Memory management and garbage collection
10. - Provides runtime environment for Java applications
11. JRE (Java Runtime Environment)
12. - The JRE is an implementation of the JVM along with supporting libraries.
13. - It contains everything needed to run compiled Java programs but not to develop them.W
14. - Components:
15. - JVM
16. - Core libraries and supporting files
17. - Java application launcher
18. JDK (Java Development Kit)
19. - The JDK is a software development kit used to develop Java applications.
20. - It includes the JRE plus development tools.
21. - Components:
22. - JRE (which includes the JVM)
23. - Compiler (javac)
24. - Debugger (jdb)
25. - Documentation generator (javadoc)
26. - Other development tools
27. Relationship
28. - JDK contains JRE, which contains JVM
29. - For development: Use JDK
30. - For running Java applications: JRE is sufficient
31. - For executing bytecode: JVM is the core component
32. Verification of Java Environment
33. To verify your Java installation, open a command prompt and type:
34. ```
35. java -version
36. javac -version
37. ```
38. If both commands display version information, your Java environment is properly set up.
39. Create a Sample Hello World Program using simple text editor (e.g. Notepad) and show the steps to compile and execute the program using command prompt.

/\*\*

 \* Question: Create Your First Java Program

 \*

 \* Write a Java program that displays "Hello, World!" on the console.

 \* This is the traditional first program for learning any programming language.

 \*

 \* Learning Objectives:

 \* 1. Understanding basic Java program structure

 \* 2. Creating and running a Java class

 \* 3. Using System.out.println() for console output

 \* 4. Proper placement of main method

 \*

 \* Concepts Demonstrated:

 \* - Java class declaration

 \* - Main method syntax

 \* - String literals

 \* - Basic output operations

 \*/

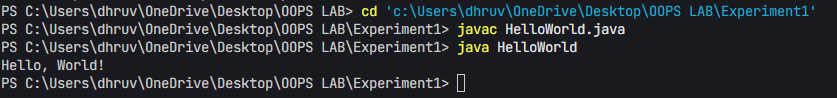
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World!");

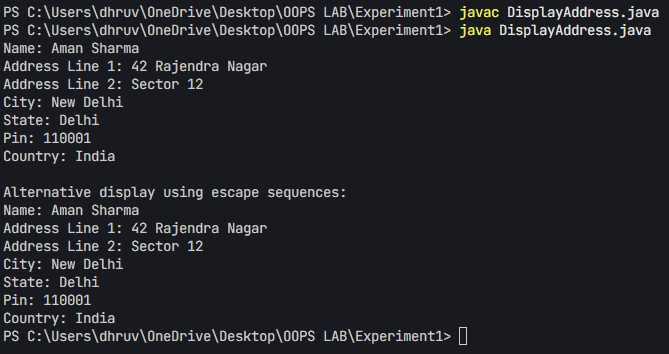
    }

}

Output-

1. Display your name and complete address in different lines.
2. /\*\*
3. \* Question: Multi-line Text Output with Address Information
4. \*
5. \* Write a Java program that displays a person's complete address information
6. \* using different output techniques. The program should demonstrate proper
7. \* use of System.out.println() and escape sequences.
8. \*
9. \* Requirements:
10. \* 1. Display the following address components on separate lines:
11. \*    - Name
12. \*    - Address Line 1
13. \*    - Address Line 2
14. \*    - City
15. \*    - State
16. \*    - Pin Code
17. \*    - Country
18. \* 2. Implement two different output methods:
19. \*    - Using multiple println statements
20. \*    - Using a single println with escape sequences
21. \*
22. \* Learning Objectives:
23. \* - Understanding console output methods
24. \* - Working with String literals
25. \* - Using escape sequences (\n)
26. \* - Formatting text output
27. \* - String concatenation
28. \*/
29. public class DisplayAddress {
30. public static void main(String[] args) {
31. // Display name and address on different lines
32. System.out.println("Name: Aman Sharma");
33. System.out.println("Address Line 1: 42 Rajendra Nagar");
34. System.out.println("Address Line 2: Sector 12");
35. System.out.println("City: New Delhi");
36. System.out.println("State: Delhi");
37. System.out.println("Pin: 110001");
38. System.out.println("Country: India");
40. // Alternative method using a single println with escape sequences
41. System.out.println("\nAlternative display using escape sequences:\n" +
42. "Name: Aman Sharma\n" +
43. "Address Line 1: 42 Rajendra Nagar\n" +
44. "Address Line 2: Sector 12\n" +
45. "City: New Delhi\n" +
46. "State: Delhi\n" +
47. "Pin: 110001\n" +
48. "Country: India");
49. }
50. }

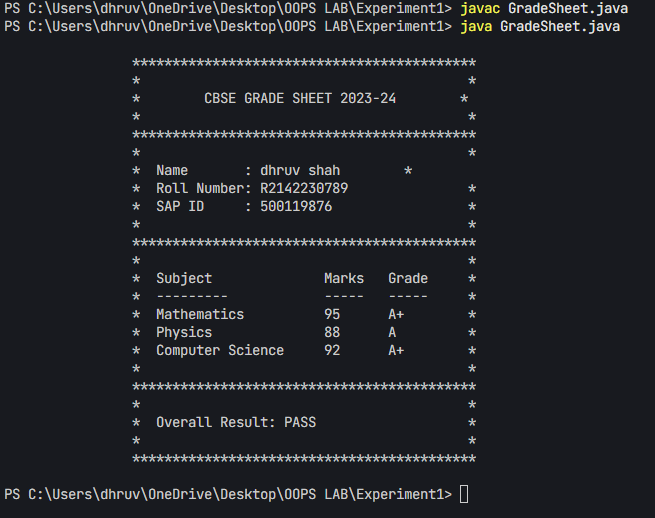
OUTPUT-



Additional Question:

1. Design a visually appealing gradesheet that displays the Name, Roll Number, SAP ID, and Result. Use escape sequences and special characters like '\*' to enhance its presentation. [No need to take any input from User].
2. /\*\*
3. \* Question: Grade Sheet Formatting with Special Characters
4. \*
5. \* Create a Java program that generates a visually appealing grade sheet using
6. \* console output formatting techniques. The program should demonstrate the use
7. \* of escape sequences and special characters for text alignment.
8. \*
9. \* Requirements:
10. \* 1. Display a formatted grade sheet with:
11. \*    - Student details (Name, Roll Number, SAP ID)
12. \*    - Subject-wise marks and grades
13. \*    - Overall result
14. \* 2. Use appropriate formatting elements:
15. \*    - Borders using special characters
16. \*    - Proper alignment using tabs and spaces
17. \*    - Clear section separation
18. \* 3. Implement proper spacing and alignment for readability
19. \*
20. \* Learning Objectives:
21. \* - Working with escape sequences (\t, \n)
22. \* - Text formatting and alignment
23. \* - Using special characters for borders
24. \* - String concatenation
25. \* - Console output formatting
26. \*/
27. public class GradeSheet {
28. public static void main(String[] args) {
29. // Define student information
30. String name = "dhruv shah";
31. String rollNumber = "R2142230789";
32. String sapId = "500119876";
33. String result = "PASS";
35. // Create a visually appealing grade sheet using escape sequences and special characters
36. System.out.println("\n\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
37. System.out.println("\t\t\*                                         \*");
38. System.out.println("\t\t\*        CBSE GRADE SHEET 2023-24        \*");
39. System.out.println("\t\t\*                                         \*");
40. System.out.println("\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
41. System.out.println("\t\t\*                                         \*");
42. System.out.println("\t\t\*  Name       : " + name + "\t  \*");
43. System.out.println("\t\t\*  Roll Number: " + rollNumber + "\t\t  \*");
44. System.out.println("\t\t\*  SAP ID     : " + sapId + "\t\t  \*");
45. System.out.println("\t\t\*                                         \*");
46. System.out.println("\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
47. System.out.println("\t\t\*                                         \*");
48. System.out.println("\t\t\*  Subject\t\tMarks\tGrade\t  \*");
49. System.out.println("\t\t\*  ---------\t\t-----\t-----\t  \*");
50. System.out.println("\t\t\*  Mathematics\t\t95\tA+\t  \*");
51. System.out.println("\t\t\*  Physics\t\t88\tA\t  \*");
52. System.out.println("\t\t\*  Computer Science\t92\tA+\t  \*");
53. System.out.println("\t\t\*                                         \*");
54. System.out.println("\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");
55. System.out.println("\t\t\*                                         \*");
56. System.out.println("\t\t\*  Overall Result: " + result + "\t\t\t  \*");
57. System.out.println("\t\t\*                                         \*");
58. System.out.println("\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");
59. }
60. }

Output



## EXPERIMENT – 2

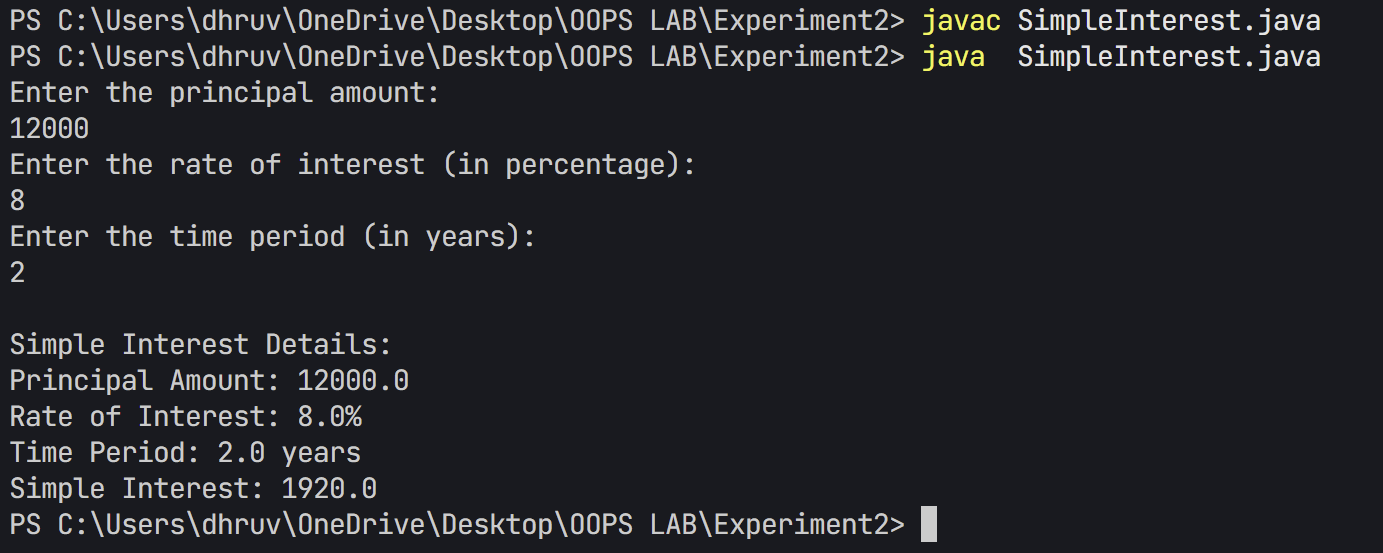
## TITLE: Basic Java Programming

1. Write a program to find area of triangle.
2. /\*\*
3. \* Program to find the area of a triangle using base and height
4. \*
5. \* Problem Statement:
6. \* Write a Java program that calculates the area of a triangle by taking the base and height
7. \* as input from the user. The program should use the formula: Area = (1/2) × base × height
8. \*
9. \* Key Concepts:
10. \* - User input handling using Scanner class
11. \* - Basic arithmetic operations
12. \* - Type conversion and calculations
13. \* - Proper resource management (closing Scanner)
14. \*
15. \* Expected Input:
16. \* - Base length of the triangle (double)
17. \* - Height of the triangle (double)
18. \*
19. \* Expected Output:
20. \* - Area of the triangle (double)
21. \*/
22. import java.util.Scanner;
23. public class TriangleArea {
24. public static void main(String[] args) {
25. Scanner scanner = new Scanner(System.in);
27. // Get input from user
28. System.out.println("Enter the base of the triangle: ");
29. double base = scanner.nextDouble();
31. System.out.println("Enter the height of the triangle: ");
32. double height = scanner.nextDouble();
34. // Calculate area
35. double area = 0.5 \* base \* height;
37. // Display result
38. System.out.println("The area of the triangle is: " + area);
40. // Close scanner
41. scanner.close();
42. }
43. }

2. Write a program to find simple interest.

1. /\*\*
2. \* Question: Simple Interest Calculator with User Input
3. \*
4. \* Create a Java program that calculates Simple Interest based on user input
5. \* for principal amount, rate of interest, and time period. The program should
6. \* demonstrate proper use of Scanner class and floating-point calculations.
7. \*
8. \* Requirements:
9. \* 1. Accept user input for:
10. \*    - Principal amount
11. \*    - Rate of interest (in percentage)
12. \*    - Time period (in years)
13. \* 2. Calculate Simple Interest using the formula: (P \* R \* T) / 100
14. \* 3. Display all input values and the calculated result
15. \* 4. Format output for better readability
16. \*
17. \* Learning Objectives:
18. \* - Working with Scanner class
19. \* - Floating-point calculations
20. \* - Basic input/output operations
21. \* - Variable declaration and initialization
22. \* - Proper code formatting and documentation
23. \*/
24. import java.util.Scanner;
25. public class SimpleInterest {
26. public static void main(String[] args) {
27. Scanner scanner = new Scanner(System.in);
29. // Get input from user
30. System.out.println("Enter the principal amount: ");
31. double principal = scanner.nextDouble();
33. System.out.println("Enter the rate of interest (in percentage): ");
34. double rate = scanner.nextDouble();
36. System.out.println("Enter the time period (in years): ");
37. double time = scanner.nextDouble();
39. // Calculate simple interest
40. double simpleInterest = (principal \* rate \* time) / 100;
42. // Display result
43. System.out.println("\nSimple Interest Details:");
44. System.out.println("Principal Amount: " + principal);
45. System.out.println("Rate of Interest: " + rate + "%");
46. System.out.println("Time Period: " + time + " years");
47. System.out.println("Simple Interest: " + simpleInterest);
49. // Close scanner
50. scanner.close();
51. }
52. }

Output



1. Write a program to implement a command line calculator. (Try for Add sub Mul in same program for 2 digits.)

**(Hint: Integer.parseInt will be used)**

**For e.g.** java calc 20 + 30

**Output** should be Sum of 20 and 30 is 50

java calc 50 \* 30

**Output** should be Product of 50 and 30 is 1500

/\*\*

 \* Question: Command Line Calculator Implementation

 \*

 \* Create a Java program that implements a command-line calculator to perform basic

 \* arithmetic operations. The program should demonstrate proper command-line argument

 \* handling and error management.

 \*

 \* Requirements:

 \* 1. Accept three command-line arguments in order:

 \*    - First number (operand 1)

 \*    - Operator (+, -, \*, /)

 \*    - Second number (operand 2)

 \* 2. Implement the following operations:

 \*    - Addition (+)

 \*    - Subtraction (-)

 \*    - Multiplication (\*)

 \*    - Division (/)

 \* 3. Include error handling for:

 \*    - Invalid number of arguments

 \*    - Invalid number format

 \*    - Division by zero

 \*    - Invalid operator

 \*

 \* Learning Objectives:

 \* - Working with command-line arguments

 \* - Implementing control structures (switch-case)

 \* - Exception handling in Java

 \* - Input validation techniques

 \* - Basic arithmetic operations

 \*/

public class CommandLineCalculator {

    public static void main(String[] args) {

        if (args.length != 3) {

            System.out.println("Usage: java CommandLineCalculator <number1> <operator> <number2>");

            System.out.println("Operators: + - \* /");

            return;

        }

        try {

            double num1 = Double.parseDouble(args[0]);

            String operator = args[1];

            double num2 = Double.parseDouble(args[2]);

            double result = 0;

            switch (operator) {

                case "+":

                    result = num1 + num2;

                    break;

                case "-":

                    result = num1 - num2;

                    break;

                case "\*":

                    result = num1 \* num2;

                    break;

                case "/":

                    if (num2 == 0) {

                        System.out.println("Error: Division by zero!");

                        return;

                    }

                    result = num1 / num2;

                    break;

                default:

                    System.out.println("Error: Invalid operator! Use +, -, \*, or /");

                    return;

            }

            System.out.println(num1 + " " + operator + " " + num2 + " = " + result);

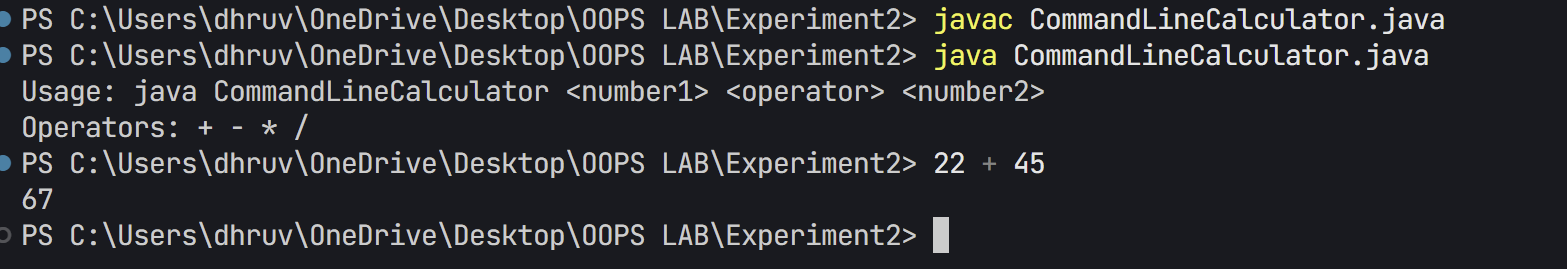
        } catch (NumberFormatException e) {

            System.out.println("Error: Please enter valid numbers!");

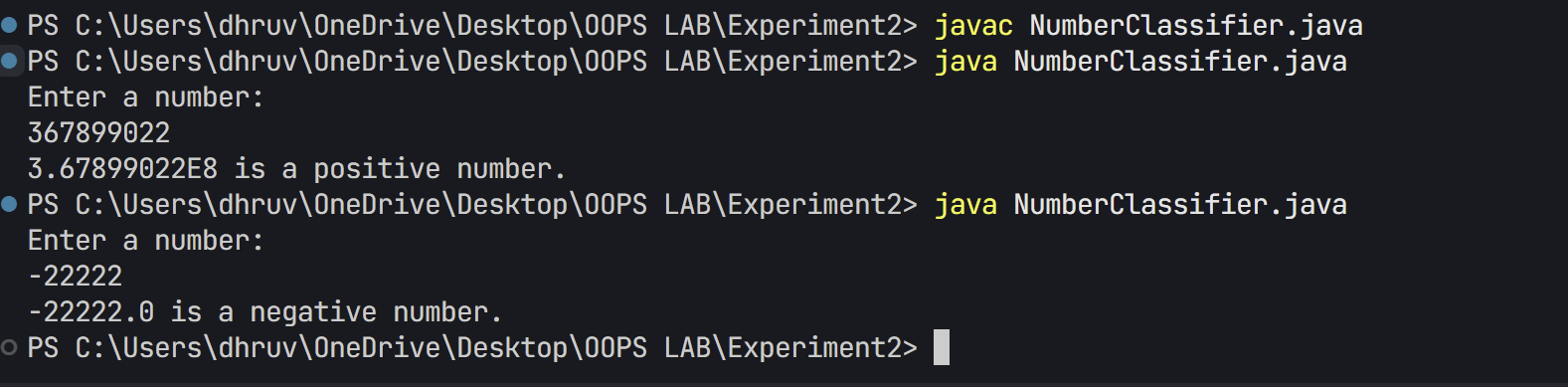
        }

    }

}



1. Write a Java program to check whether a given number is positive, negative, or zero using an if-else statement.
2. /\*\*
3. \* Question: Number Classification Program
4. \*
5. \* Write a Java program that takes a number as input and classifies it as positive,
6. \* negative, or zero. The program should demonstrate the use of conditional statements
7. \* for number classification.
8. \*
9. \* Requirements:
10. \* 1. Accept a number input from the user
11. \* 2. Classify the number as:
12. \*    - Positive (greater than 0)
13. \*    - Negative (less than 0)
14. \*    - Zero (equal to 0)
15. \* 3. Display appropriate message based on classification
16. \*
17. \* Learning Objectives:
18. \* - Implementation of if-else statements
19. \* - Working with Scanner class for input
20. \* - Basic number comparison operations
21. \* - Proper input handling and output formatting
22. \*/
23. import java.util.Scanner;
24. public class NumberClassifier {
25. public static void main(String[] args) {
26. Scanner scanner = new Scanner(System.in);
28. // Get input from user
29. System.out.println("Enter a number: ");
30. double number = scanner.nextDouble();
32. // Check if the number is positive, negative, or zero
33. if (number > 0) {
34. System.out.println(number + " is a positive number.");
35. } else if (number < 0) {
36. System.out.println(number + " is a negative number.");
37. } else {
38. System.out.println("The number is zero.");
39. }
41. // Close scanner
42. scanner.close();
43. }
44. }



5.Create a program that accepts three integers and determines the greatest among them using nested if-else statements.

/\*\*

 \* Question: Finding the Greatest Among Three Numbers

 \*

 \* Write a Java program that finds the greatest number among three integers input

 \* by the user. The program should demonstrate the use of conditional statements

 \* and comparison operators.

 \*

 \* Requirements:

 \* 1. Accept three integer inputs from the user

 \* 2. Compare the numbers using appropriate conditional statements

 \* 3. Display the greatest number among the three

 \*

 \* Learning Objectives:

 \* - Implementation of if statements for comparison

 \* - Working with Scanner class for multiple inputs

 \* - Basic number comparison logic

 \* - Proper variable initialization and updating

 \*/

import java.util.Scanner;

public class GreatestNumber {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Get input from user

        System.out.println("Enter three numbers:");

        System.out.print("First number: ");

        int num1 = scanner.nextInt();

        System.out.print("Second number: ");

        int num2 = scanner.nextInt();

        System.out.print("Third number: ");

        int num3 = scanner.nextInt();

        // Find the greatest number

        int greatest = num1;

        if (num2 > greatest) {

            greatest = num2;

        }

        if (num3 > greatest) {

            greatest = num3;

        }

        // Display result

        System.out.println("\nThe greatest number among " + num1 + ", " + num2 + ", and " + num3 + " is: " + greatest);

        // Close scanner

        scanner.close();

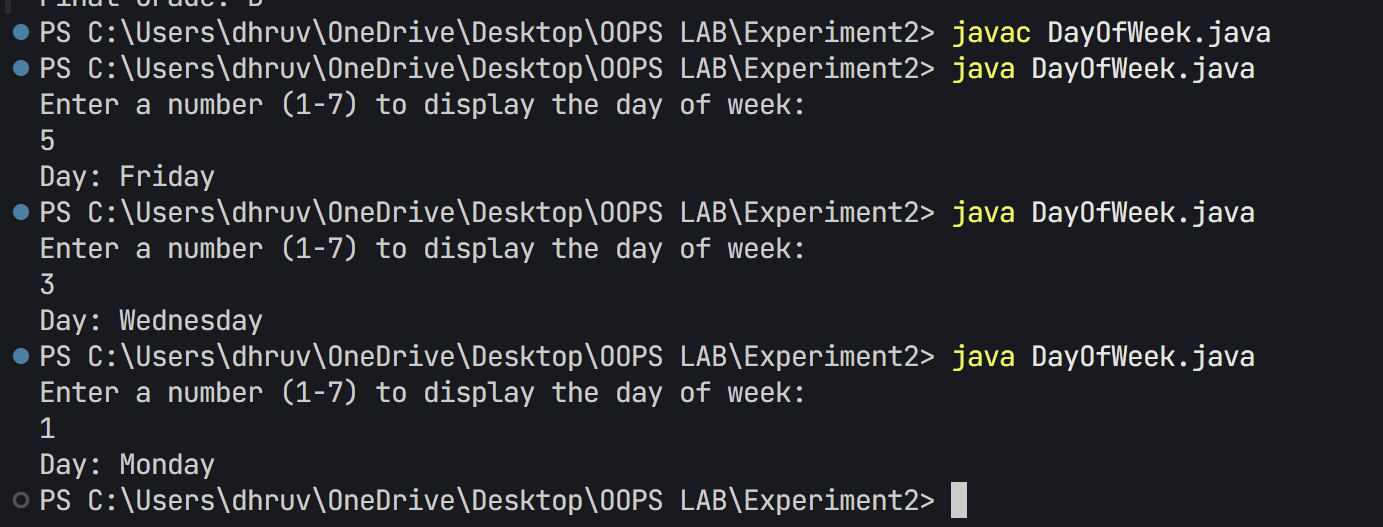
    }

}

6. Create a program that accepts a number (1–7) and displays the corresponding day of the week using a switch statement.

1. /\*\*
2. \* Question: Day of Week Converter using Switch Statement
3. \*
4. \* Write a Java program that accepts a number (1-7) as input and displays
5. \* the corresponding day of the week. The program should demonstrate the
6. \* use of switch statement for multiple condition handling.
7. \*
8. \* Requirements:
9. \* 1. Accept a number input from the user (1-7)
10. \* 2. Use switch statement to convert number to day name
11. \* 3. Display appropriate error message for invalid inputs
12. \* 4. Implement proper input validation
13. \*
14. \* Learning Objectives:
15. \* - Understanding switch statement syntax and usage
16. \* - Working with Scanner class for user input
17. \* - Implementing input validation
18. \* - String variable assignment
19. \* - Basic control flow
20. \*/
21. import java.util.Scanner;
22. public class DayOfWeek {
23. public static void main(String[] args) {
24. Scanner scanner = new Scanner(System.in);
26. // Get input from user
27. System.out.println("Enter a number (1-7) to display the day of week:");
28. int dayNumber = scanner.nextInt();
30. String day;
32. // Determine the day using switch statement
33. switch (dayNumber) {
34. case 1:
35. day = "Monday";
36. break;
37. case 2:
38. day = "Tuesday";
39. break;
40. case 3:
41. day = "Wednesday";
42. break;
43. case 4:
44. day = "Thursday";
45. break;
46. case 5:
47. day = "Friday";
48. break;
49. case 6:
50. day = "Saturday";
51. break;
52. case 7:
53. day = "Sunday";
54. break;
55. default:
56. day = "Invalid day number! Please enter a number between 1 and 7.";
57. }
59. // Display result
60. System.out.println("Day: " + day);
62. // Close scanner
63. scanner.close();
64. }
65. }

**Output**

****

**Additional Question:**

1. Write a program to calculate the final grade of a student based on the marks entered in three subjects. Use the following grading scale:

Average >= 90: Grade A

Average >= 75: Grade B

Average >= 50: Grade C

Otherwise: Grade F

/\*\*

 \* Question: Student Grade Calculator

 \*

 \* Write a Java program that calculates a student's final grade based on marks obtained

 \* in three subjects. The program should validate input marks and assign appropriate

 \* grade based on the average score.

 \*

 \* Requirements:

 \* 1. Accept marks for three subjects (range: 0-100)

 \* 2. Validate that marks are within valid range

 \* 3. Calculate average of three subjects

 \* 4. Assign grade based on the following criteria:

 \*    - A: >= 90

 \*    - B: >= 80

 \*    - C: >= 70

 \*    - D: >= 60

 \*    - F: < 60

 \*

 \* Learning Objectives:

 \* - Input validation using conditional statements

 \* - Working with Scanner class for multiple inputs

 \* - Calculating averages with type conversion

 \* - Implementing multi-level grade classification

 \* - Error handling for invalid inputs

 \*/

import java.util.Scanner;

public class StudentGrade {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Get input from user

        System.out.println("Enter marks for three subjects (0-100):");

        System.out.print("Subject 1: ");

        int subject1 = scanner.nextInt();

        System.out.print("Subject 2: ");

        int subject2 = scanner.nextInt();

        System.out.print("Subject 3: ");

        int subject3 = scanner.nextInt();

        // Validate marks

        if (subject1 < 0 || subject1 > 100 || subject2 < 0 || subject2 > 100 || subject3 < 0 || subject3 > 100) {

            System.out.println("Invalid marks! Marks should be between 0 and 100.");

            scanner.close();

            return;

        }

        // Calculate average

        double average = (subject1 + subject2 + subject3) / 3.0;

        // Determine grade

        String grade;

        if (average >= 90) {

            grade = "A+";

        } else if (average >= 80) {

            grade = "A";

        } else if (average >= 70) {

            grade = "B";

        } else if (average >= 60) {

            grade = "C";

        } else if (average >= 50) {

            grade = "D";

        } else {

            grade = "F";

        }

        // Display results

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

        System.out.println("Subject 1: " + subject1);

        System.out.println("Subject 2: " + subject2);

        System.out.println("Subject 3: " + subject3);

        System.out.printf("Average: %.2f\n", average);

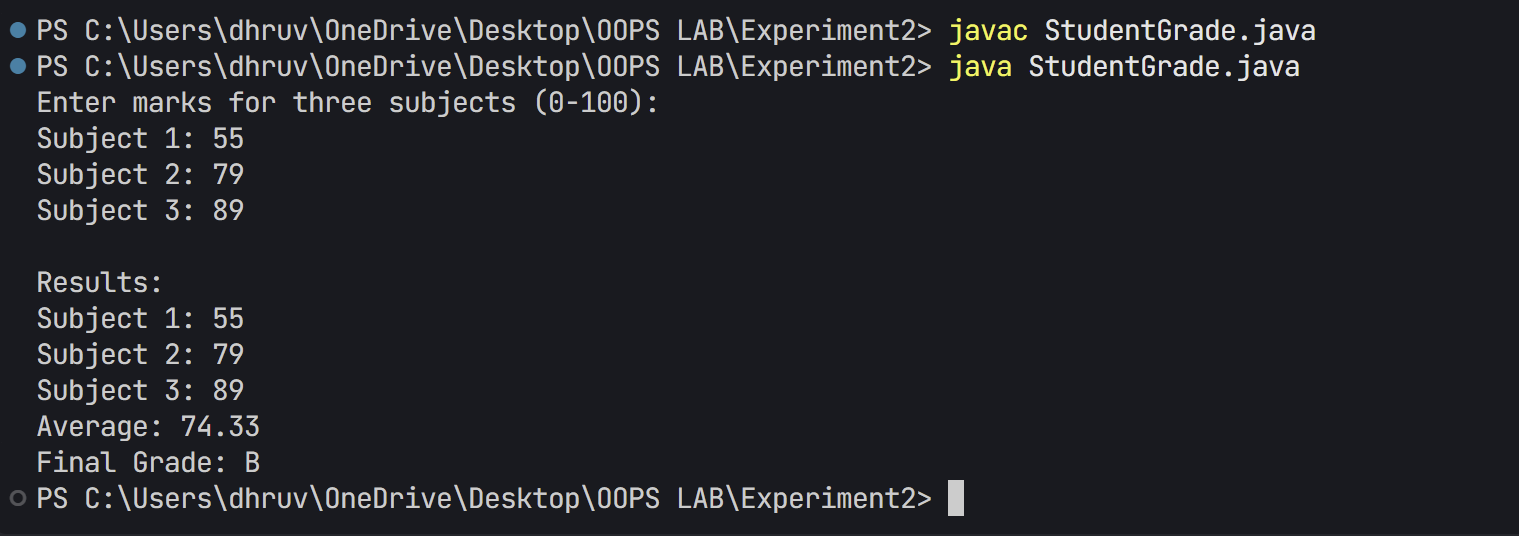
        System.out.println("Final Grade: " + grade);

        // Close scanner

        scanner.close();

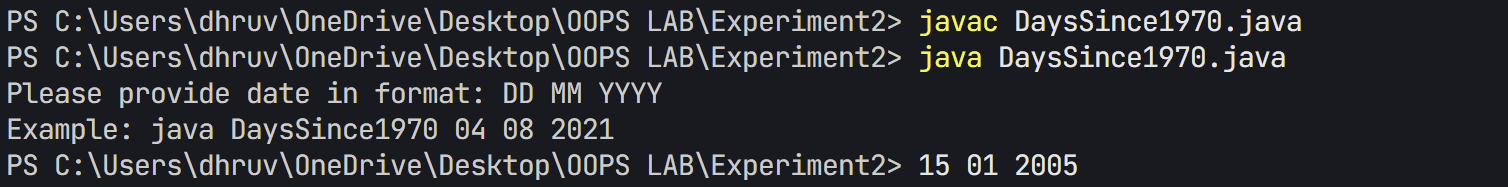
    }

}



1. WAP to Take input as DD MM YYYY(04 08 2021) in command line and calculate number of days since 1 January 1970.
2. /\*\*
3. \* Question: Days Calculator Since Unix Epoch
4. \*
5. \* Write a Java program that calculates the number of days between a given date
6. \* and January 1, 1970 (Unix epoch). The program should demonstrate the use of
7. \* Java's date/time API and command-line argument handling.
8. \*
9. \* Requirements:
10. \* 1. Accept date input as command-line arguments in format: DD MM YYYY
11. \* 2. Validate input format and date validity
12. \* 3. Calculate days between input date and Unix epoch (01/01/1970)
13. \* 4. Handle invalid inputs and edge cases appropriately
14. \*
15. \* Learning Objectives:
16. \* - Working with Java's LocalDate class
17. \* - Command-line argument parsing and validation
18. \* - Date calculations using ChronoUnit
19. \* - Exception handling for invalid inputs
20. \* - Proper date formatting and display
21. \*/
22. import java.time.LocalDate;
23. import java.time.temporal.ChronoUnit;
24. public class DaysSince1970 {
25. public static void main(String[] args) {
26. // Check if correct number of arguments are provided
27. if (args.length != 3) {
28. System.out.println("Please provide date in format: DD MM YYYY");
29. System.out.println("Example: java DaysSince1970 04 08 2021");
30. return;
31. }
32. try {
33. // Parse command line arguments
34. int day = Integer.parseInt(args[0]);
35. int month = Integer.parseInt(args[1]);
36. int year = Integer.parseInt(args[2]);
37. // Create LocalDate objects
38. LocalDate inputDate = LocalDate.of(year, month, day);
39. LocalDate epochDate = LocalDate.of(1970, 1, 1);
40. // Calculate days between dates
41. long daysSinceEpoch = ChronoUnit.DAYS.between(epochDate, inputDate);
42. // Display result
43. System.out.printf("Number of days between %02d/%02d/%04d and 01/01/1970: %d\n",
44. day, month, year, daysSinceEpoch);
45. } catch (NumberFormatException e) {
46. System.out.println("Error: Please enter valid numbers for date.");
47. } catch (Exception e) {
48. System.out.println("Error: Invalid date. Please ensure the date is valid.");
49. }
50. }
51. }

**Output**

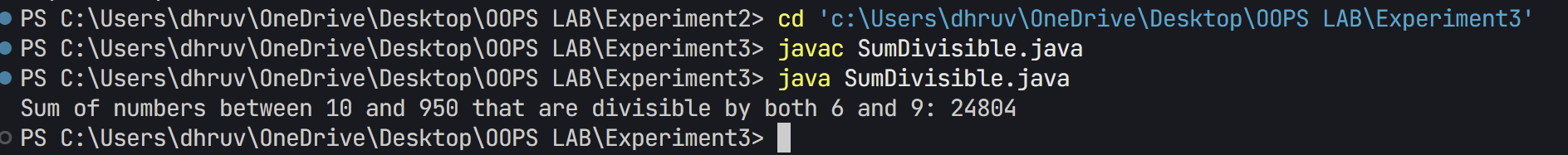
****

## EXPERIMENT – 3

## TITLE: Basic Java Programming (Loops & Arrays)

1. Write a program to calculate the sum of all integers between 10 and 950 that are divisible by both 6 and 9.
2. package Experiment3;
3. /\*\*
4. \* Question: Write a Java program to find the sum of numbers divisible by both 6 and 9.
5. \*
6. \* Problem Statement:
7. \* - Calculate the sum of all integers between 10 and 950 (inclusive)
8. \* - Numbers must be divisible by both 6 and 9 (i.e., divisible by LCM(6,9) = 18)
9. \* - Use a loop to iterate through the range
10. \* - Apply the divisibility test using modulo operator
11. \* - Display the final sum
12. \*
13. \* Learning Objectives:
14. \* - Understanding loops and conditions
15. \* - Working with modulo operator
16. \* - Number divisibility concepts
17. \* - Basic arithmetic operations
18. \*/
19. public class SumDivisible {
20. public static void main(String[] args) {
21. int sum = 0;
23. // Loop through numbers from 10 to 950
24. for (int i = 10; i <= 950; i++) {
25. // Check if number is divisible by both 6 and 9
26. if (i % 6 == 0 && i % 9 == 0) {
27. sum += i;
28. }
29. }
31. // Display result
32. System.out.println("Sum of numbers between 10 and 950 that are divisible by both 6 and 9: " + sum);
33. }
34. }

Output



1. Write a Java program that takes an integer as input and calculates the sum of its digits using a while loop.

package Experiment3;

/\*\*

 \* Question: Write a Java program to calculate the sum of digits in an integer.

 \*

 \* Problem Statement:

 \* - Create a program that calculates the sum of all digits in a given integer

 \* - Program should handle both positive and negative numbers

 \* - Use a while loop for digit extraction

 \* - Display both the original number and the sum of its digits

 \*

 \* Example:

 \* Input: 12345

 \* Output: Sum of digits in 12345: 15 (1+2+3+4+5)

 \*

 \* Learning Objectives:

 \* - Working with loops

 \* - Basic arithmetic operations

 \* - Number manipulation

 \* - Input validation

 \*/

import java.util.Scanner;

public class DigitSum {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Get input from user

        System.out.print("Enter an integer: ");

        try {

            int number = scanner.nextInt();

            int originalNumber = number;

            int sum = 0;

            // Take absolute value for negative numbers

            number = Math.abs(number);

            // Calculate sum of digits using while loop

            while (number > 0) {

                sum += number % 10;  // Add last digit to sum

                number /= 10;        // Remove last digit

            }

            // Display result

            System.out.printf("Sum of digits in %d: %d\n", originalNumber, sum);

        } catch (Exception e) {

            System.out.println("Error: Please enter a valid integer.");

        } finally {

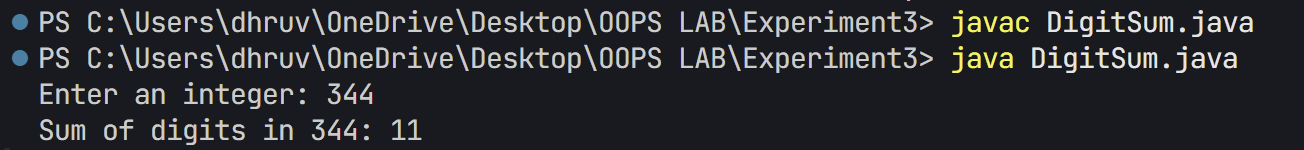
            scanner.close();

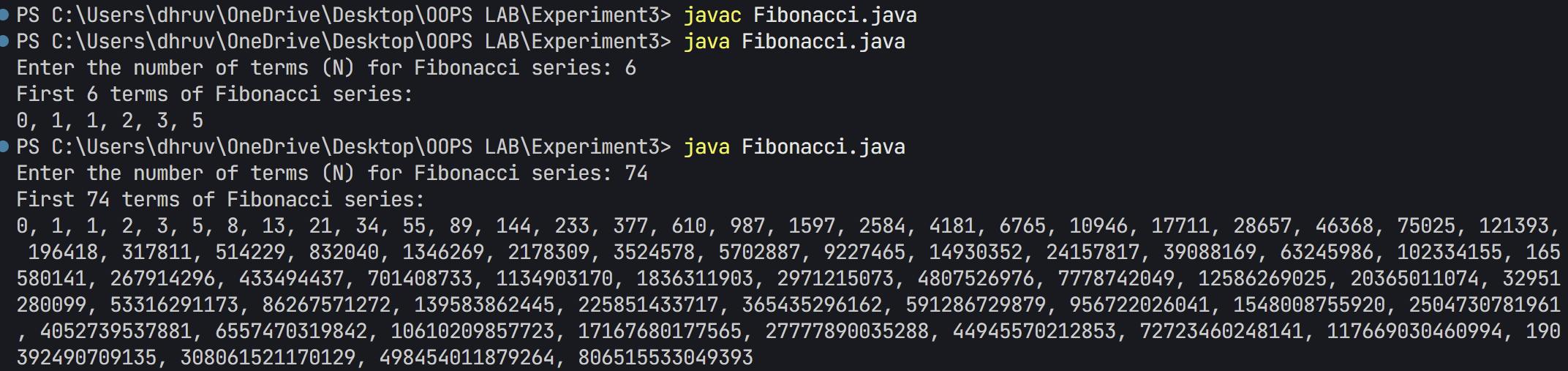
        }

    }

}

Output

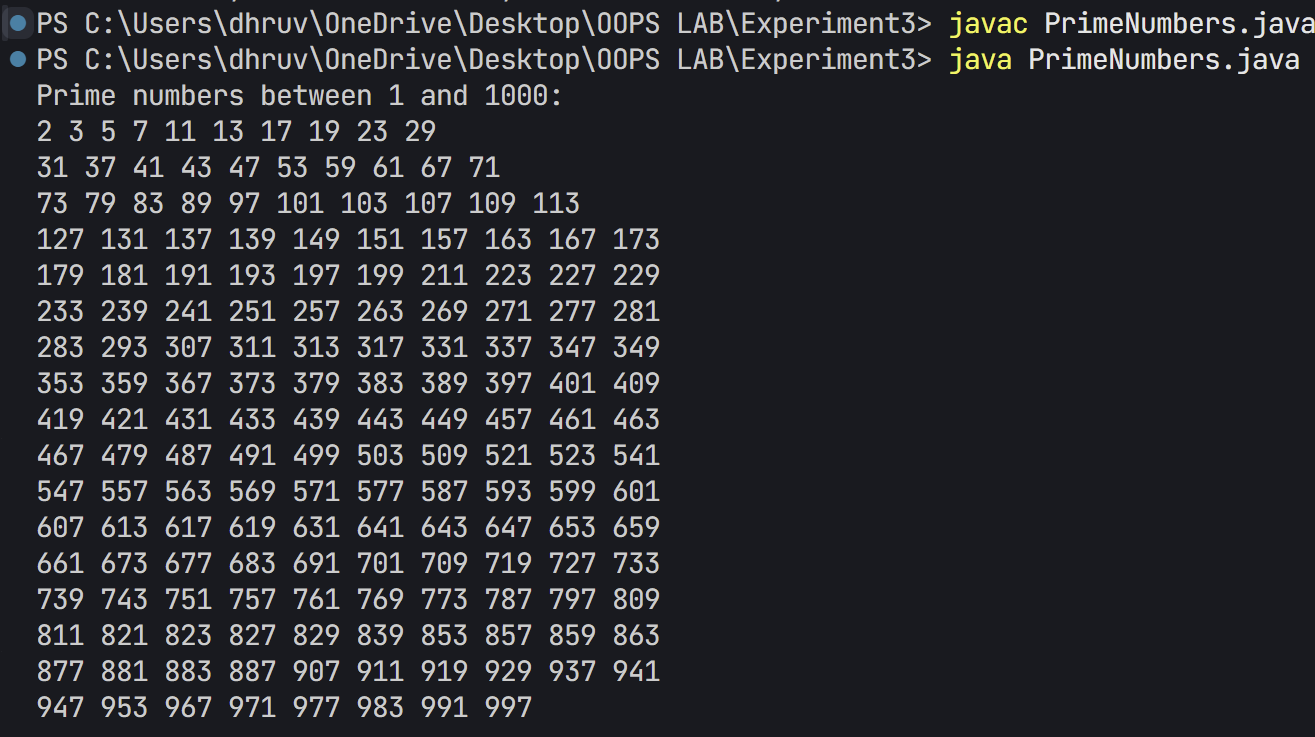


1. Write a Java program that prints the first N terms of the Fibonacci series using a loop.
2. package Experiment3;
3. /\*\*
4. \* Question: Fibonacci Series Generation
5. \*
6. \* Create a Java program that generates and displays the Fibonacci series
7. \* up to N terms. The program should demonstrate proper input handling,
8. \* loop implementation, and series generation logic.
9. \*
10. \* Requirements:
11. \* 1. Accept user input for number of terms (N)
12. \* 2. Validate input:
13. \*    - Ensure N is positive
14. \*    - Handle invalid input gracefully
15. \* 3. Generate and display the series:
16. \*    - First term: 0
17. \*    - Second term: 1
18. \*    - Subsequent terms: sum of previous two terms
19. \* 4. Format output with proper separators
20. \*
21. \* Learning Objectives:
22. \* - Working with loops and conditions
23. \* - Handling user input with Scanner
24. \* - Exception handling
25. \* - Number series generation
26. \* - Basic arithmetic operations
27. \*/
28. import java.util.Scanner;
29. public class Fibonacci {
30. public static void main(String[] args) {
31. Scanner scanner = new Scanner(System.in);
33. // Get input from user
34. System.out.print("Enter the number of terms (N) for Fibonacci series: ");
36. try {
37. int n = scanner.nextInt();
39. // Validate input
40. if (n <= 0) {
41. System.out.println("Please enter a positive number.");
42. return;
43. }
45. System.out.println("First " + n + " terms of Fibonacci series:");
47. long first = 0, second = 1;
49. // Print first term
50. System.out.print(first);
52. // If N > 1, print second term
53. if (n > 1) {
54. System.out.print(", " + second);
55. }
57. // Print remaining terms using loop
58. for (int i = 3; i <= n; i++) {
59. long next = first + second;
60. System.out.print(", " + next);
61. first = second;
62. second = next;
63. }
65. System.out.println(); // New line after series
67. } catch (Exception e) {
68. System.out.println("Error: Please enter a valid positive integer.");
69. } finally {
70. scanner.close();
71. }
72. }
73. }

4. Write a Java program to count and display the total number of prime numbers between 1 and 1000.

1. package Experiment3;
2. /\*\*
3. \* Question: Prime Number Generator and Counter
4. \*
5. \* Create a Java program that finds and displays all prime numbers between 1 and 1000.
6. \* The program should demonstrate efficient prime number checking algorithm and proper
7. \* output formatting.
8. \*
9. \* Requirements:
10. \* 1. Find all prime numbers in range 1-1000
11. \* 2. Display the numbers in a formatted grid (10 numbers per line)
12. \* 3. Implement efficient prime checking algorithm using square root
13. \* 4. Count and display total number of prime numbers found
14. \*
15. \* Learning Objectives:
16. \* - Understanding prime number concepts
17. \* - Implementing nested loops
18. \* - Using Math class methods (sqrt)
19. \* - Output formatting techniques
20. \* - Algorithm optimization basics
21. \*/
22. public class PrimeNumbers {
23. public static void main(String[] args) {
24. int count = 0;
25. System.out.println("Prime numbers between 1 and 1000:");
27. // Check each number from 2 to 1000
28. for (int num = 2; num <= 1000; num++) {
29. boolean isPrime = true;
31. // Check if number is prime
32. for (int i = 2; i <= Math.sqrt(num); i++) {
33. if (num % i == 0) {
34. isPrime = false;
35. break;
36. }
37. }
39. // If prime, print it and increment counter
40. if (isPrime) {
41. System.out.print(num + " ");
42. count++;
44. // Add new line after every 10 numbers for better readability
45. if (count % 10 == 0) {
46. System.out.println();
47. }
48. }
49. }
51. // Print total count
52. System.out.println("\n\nTotal number of prime numbers: " + count);
53. }
54. }

Output



1. Write a Java program that counts how many times a given number appears in an array.

Input: arr = [2, 3, 2, 5, 2, 6], target = 2

Output: 3

import java.util.Scanner;

/\*\*

 \* Question: Write a Java program to count the occurrences of a specific number in an array.

 \*

 \* Problem Statement:

 \* - Create a program that counts how many times a user-specified number appears in a predefined array

 \* - The program should have a fixed array of integers

 \* - User should be able to input a target number to search for

 \* - Program must display both the array contents and the count of occurrences

 \*

 \* Example:

 \* If array is [2, 3, 2, 5, 2, 6] and user inputs 2

 \* Output should show that 2 appears 3 times in the array

 \*/

public class CountOccurrences {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Initialize array

        int[] arr = {2, 3, 2, 5, 2, 6};

        // Get target number from user

        System.out.print("Enter the number to count: ");

        int target = scanner.nextInt();

        // Count occurrences

        int count = 0;

        for (int num : arr) {

            if (num == target) {

                count++;

            }

        }

        // Display result

        System.out.println("\nArray elements: ");

        for (int num : arr) {

            System.out.print(num + " ");

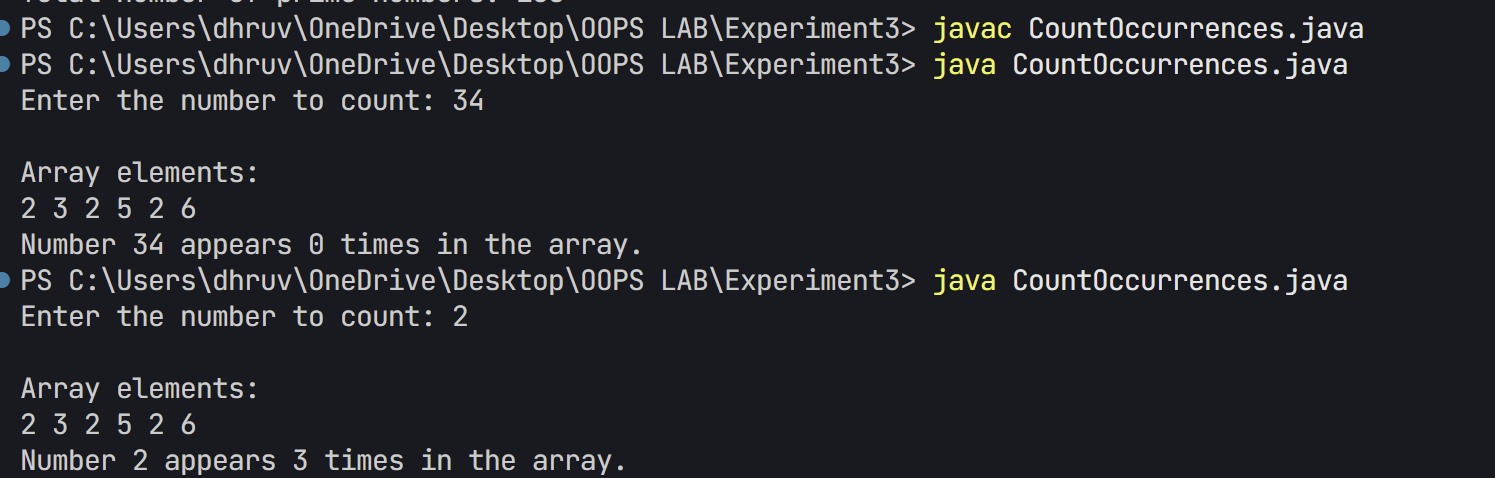
        }

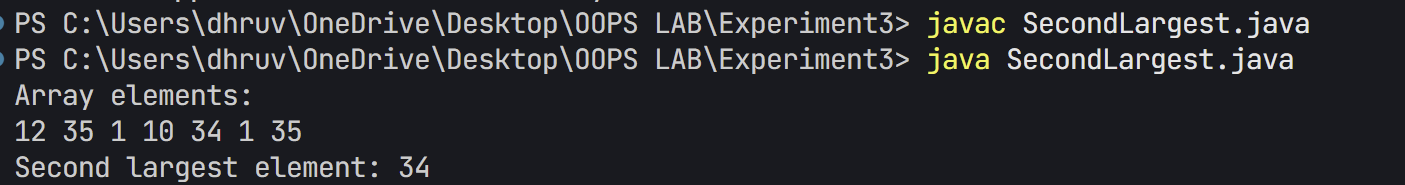
        System.out.println("\nNumber " + target + " appears " + count + " times in the array.");

        // Close scanner

        scanner.close();

    }

}

1. Write a Java program to find the second largest element in an integer array without sorting the array.
2. package Experiment3;
3. /\*\*
4. \* Question: Write a Java program to find the second largest element in an array.
5. \*
6. \* Problem Statement:
7. \* - Create a program that finds the second largest element in an array
8. \* - Do not use array sorting
9. \* - Handle duplicate elements correctly
10. \* - Use a single pass through the array
11. \* - Display both the array contents and the second largest element
12. \*
13. \* Example:
14. \* Array: [12, 35, 1, 10, 34, 1, 35]
15. \* Second largest: 34
16. \*
17. \* Learning Objectives:
18. \* - Array traversal
19. \* - Conditional logic
20. \* - Handling edge cases
21. \* - Algorithm optimization
22. \*/
23. public class SecondLargest {
24. public static void main(String[] args) {
25. // Initialize array
26. int[] arr = {12, 35, 1, 10, 34, 1, 35};
28. // Variables to track largest and second largest
29. int largest = Integer.MIN\_VALUE;
30. int secondLargest = Integer.MIN\_VALUE;
32. // Find largest and second largest
33. for (int num : arr) {
34. if (num > largest) {
35. secondLargest = largest;
36. largest = num;
37. } else if (num > secondLargest && num != largest) {
38. secondLargest = num;
39. }
40. }
42. // Display array
43. System.out.println("Array elements:");
44. for (int num : arr) {
45. System.out.print(num + " ");
46. }
48. // Display result
49. if (secondLargest == Integer.MIN\_VALUE) {
50. System.out.println("\nNo second largest element exists.");
51. } else {
52. System.out.println("\nSecond largest element: " + secondLargest);
53. }
54. }
55. }

7. WAP to print the following pattern using loop

?

# # #

? ? ? ? ?

# # # # # # #

? ? ? ? ? ? ? ? ?

package Experiment3;

/\*\*

 \* Question: Write a Java program to print a specific pattern using symbols.

 \*

 \* Problem Statement:

 \* - Create a program that prints a pyramid pattern using # and ? symbols

 \* - Pattern should be 4 rows high

 \* - Odd-numbered rows use ? symbols

 \* - Even-numbered rows use # symbols

 \* - Each row should be properly centered

 \* - Number of symbols in each row follows the pattern: 1,3,5,7

 \*

 \* Expected Pattern:

 \* ?

 \* # # #

 \* ? ? ? ? ?

 \* # # # # # # #

 \*

 \* Learning Objectives:

 \* - Nested loops

 \* - Pattern logic

 \* - Space formatting

 \* - Symbol alternation

 \*/

public class PatternPrinting {

    public static void main(String[] args) {

        // Number of rows in pattern

        int rows = 4;

        // Loop for each row

        for (int i = 0; i < rows; i++) {

            // Calculate number of symbols for current row

            int symbols = 2 \* i + 1;

            // Print spaces for alignment

            for (int space = 0; space < (7 - symbols) / 2; space++) {

                System.out.print("  ");

            }

            // Print symbols

            for (int j = 0; j < symbols; j++) {

                // Alternate between ? and # for each row

                if (i % 2 == 0) {

                    System.out.print("? ");

                } else {

                    System.out.print("# ");

                }

            }

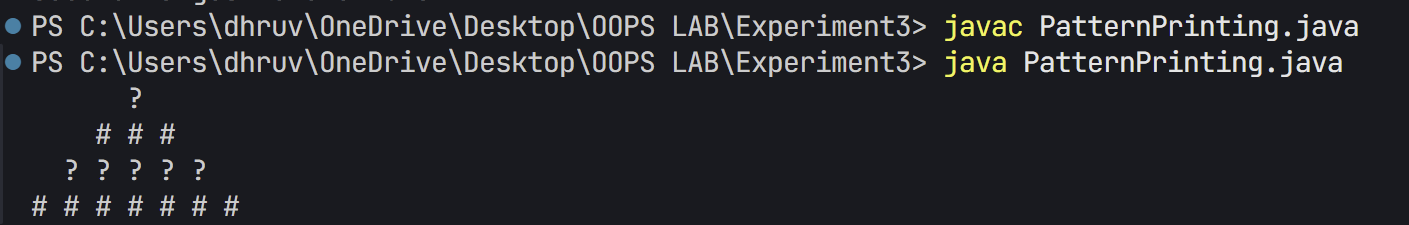
            // Move to next line

            System.out.println();

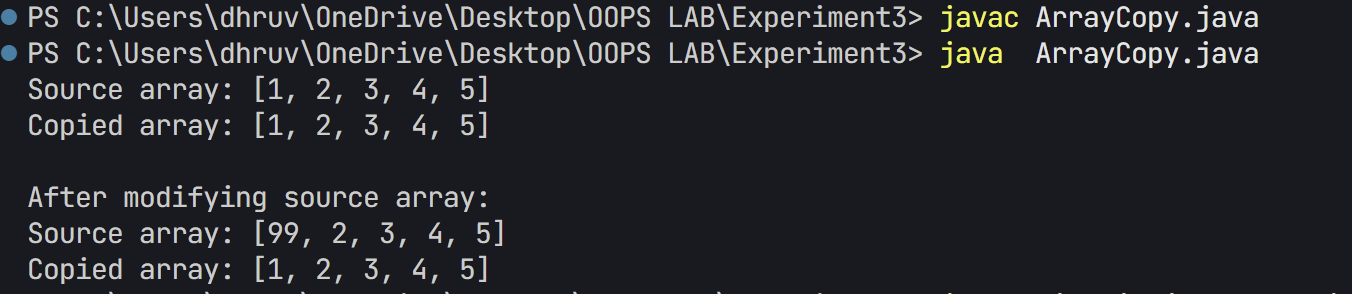
        }

    }

}



Additional Questions:

1. Write a Java program that copies all elements from one array to another using a loop.
2. package Experiment3;
3. import java.util.Arrays;
4. /\*\*
5. \* Question 1: Array Copy Implementation
6. \*
7. \* Write a Java program that copies all elements from one array to another using a loop.
8. \* The program should demonstrate proper array manipulation techniques and verify that
9. \* the arrays are separate in memory.
10. \*
11. \* Requirements:
12. \* 1. Create a source array with initial values
13. \* 2. Create a destination array of the same size
14. \* 3. Copy elements one by one using a loop (not using built-in methods)
15. \* 4. Display both arrays before and after modifying the source array
16. \* 5. Demonstrate that arrays are separate in memory by modifying source array
17. \*
18. \* Learning Objectives:
19. \* - Understanding array initialization and copying
20. \* - Working with array indices and loops
21. \* - Demonstrating array reference vs value copying
22. \* - Using Arrays.toString() for array output
23. \* - Basic array manipulation techniques
24. \*/
25. public class ArrayCopy {
26. public static void main(String[] args) {
27. // Source array
28. int[] sourceArray = {1, 2, 3, 4, 5};
30. // Create destination array with same length
31. int[] destinationArray = new int[sourceArray.length];
33. // Copy elements using loop
34. for (int i = 0; i < sourceArray.length; i++) {
35. destinationArray[i] = sourceArray[i];
36. }
38. // Display results
39. System.out.println("Source array: " + Arrays.toString(sourceArray));
40. System.out.println("Copied array: " + Arrays.toString(destinationArray));
42. // Verify arrays are distinct objects
43. sourceArray[0] = 99; // Modify source array
44. System.out.println("\nAfter modifying source array:");
45. System.out.println("Source array: " + Arrays.toString(sourceArray));
46. System.out.println("Copied array: " + Arrays.toString(destinationArray));
47. }
48. }
49. Given an array containing N-1 unique numbers from 1 to N, write a Java program to find the missing number.

Input: [1, 5, 6, 2, 4]

Output: 3

package Experiment3;

/\*\*

 \* Question 2: Find Missing Number in Sequence

 \*

 \* Given an array containing N-1 unique numbers from 1 to N, write a Java program to find

 \* the missing number. The program should use an efficient approach based on the sum formula

 \* for sequence of numbers.

 \*

 \* Requirements:

 \* 1. Handle input array with N-1 numbers (one number missing from sequence 1 to N)

 \* 2. Use mathematical approach (sum formula) to find missing number

 \* 3. Example case: Input array [1, 5, 6, 2, 4] → Output: 3

 \* 4. Display both input array and the missing number

 \*

 \* Learning Objectives:

 \* - Understanding arithmetic sequences

 \* - Implementing efficient mathematical solutions

 \* - Array traversal and sum calculation

 \* - Problem-solving using mathematical approach

 \*/

public class FindMissingNumber {

    public static void main(String[] args) {

        // Initialize array with missing number

        int[] arr = {1, 5, 6, 2, 4};

        // Calculate N (the range of numbers)

        int n = arr.length + 1;

        // Calculate expected sum of numbers from 1 to N

        int expectedSum = (n \* (n + 1)) / 2;

        // Calculate actual sum of array

        int actualSum = 0;

        for (int num : arr) {

            actualSum += num;

        }

        // Missing number is the difference

        int missingNumber = expectedSum - actualSum;

        // Display array

        System.out.print("Array elements: ");

        for (int num : arr) {

            System.out.print(num + " ");

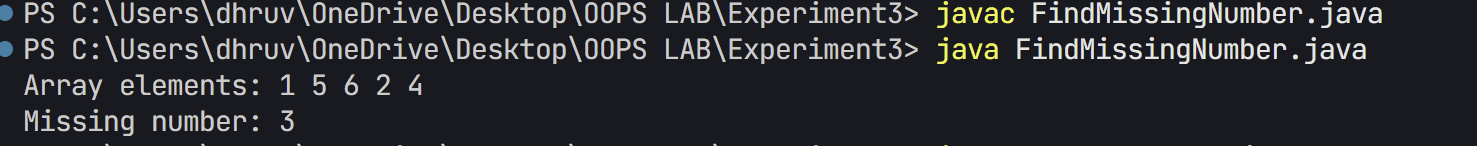
        }

        // Display result

        System.out.println("\nMissing number: " + missingNumber);

    }

}



1. Write a Java program to rotate an array right by K positions.

Input: arr = [1, 2, 3, 4, 5], K = 2

Output: [4, 5, 1, 2, 3]

package Experiment3;

import java.util.Arrays;

/\*\*

 \* Question 3: Array Rotation Implementation

 \*

 \* Write a Java program to rotate an array right by K positions. The program should

 \* demonstrate efficient array manipulation using temporary array and modulo operation

 \* for position calculation.

 \*

 \* Requirements:

 \* 1. Take an array and rotation count K as input

 \* 2. Rotate array elements K positions to the right

 \* 3. Example case: Input array [1, 2, 3, 4, 5] with K = 2 → Output: [4, 5, 1, 2, 3]

 \* 4. Handle K values larger than array length using modulo

 \* 5. Display both original and rotated arrays

 \*

 \* Learning Objectives:

 \* - Array manipulation techniques

 \* - Understanding array rotation algorithm

 \* - Using modulo for array index wrapping

 \* - Implementing efficient solution with Arrays utility

 \*/

public class RotateArray {

    public static void main(String[] args) {

        // Initialize array

        int[] arr = {1, 2, 3, 4, 5};

        int k = 2; // Number of positions to rotate

        // Create copy of original array

        int[] original = Arrays.copyOf(arr, arr.length);

        // Ensure k is within array bounds

        k = k % arr.length;

        // Rotate array

        for (int i = 0; i < arr.length; i++) {

            int newPosition = (i + k) % arr.length;

            arr[newPosition] = original[i];

        }

        // Display results

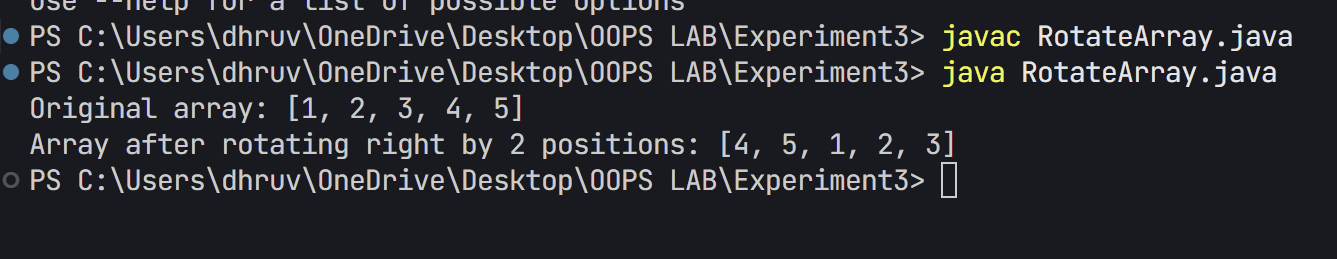
        System.out.println("Original array: " + Arrays.toString(original));

        System.out.println("Array after rotating right by " + k + " positions: " + Arrays.toString(arr));

    }

}

**Output**

****

## EXPERIMENT – 4

## TITLE: Classes (Constructors, Access modifiers, Method Overloading, static & non static data members, this)

1. Create a Student class with attributes for name and age. Implement a default constructor to assign default values and a parameterized constructor to initialize the attributes with user-defined values. Create objects using both constructors and display their details.
2. package Experiment4;
3. /\*\*
4. \* Question: Student Information System with Static Members
5. \*
6. \* Create a Java program that implements a Student Information System to demonstrate
7. \* the use of static and non-static members, along with different types of constructors.
8. \*
9. \* Requirements:
10. \* 1. Implement Student class with:
11. \*    - Instance variables for student details (name, age)
12. \*    - Static variable for university name
13. \*    - Default and parameterized constructors
14. \*    - Static and non-static methods
15. \* 2. Demonstrate:
16. \*    - Object creation using both constructors
17. \*    - Static member access
18. \*    - Proper information display
19. \*
20. \* Learning Objectives:
21. \* - Understanding static vs non-static members
22. \* - Working with multiple constructors
23. \* - Implementing proper access modifiers
24. \* - Basic class design principles
25. \* - Object instantiation techniques
26. \*/
27. public class Student {
28. private String name;
29. private int age;
30. public static String universityName = "MyUniversity";
32. // Default constructor
33. public Student() {
34. this.name = "Unknown";
35. this.age = 0;
36. }
38. // Parameterized constructor
39. public Student(String name, int age) {
40. this.name = name;
41. this.age = age;
42. }
44. // Static method to display university name
45. public static void displayUniversity() {
46. System.out.println("University: " + universityName);
47. }
49. // Method to display student details
50. public void displayDetails() {
51. System.out.println("Name: " + name);
52. System.out.println("Age: " + age);
53. }
55. public static void main(String[] args) {
56. // Create objects using both constructors
57. Student student1 = new Student(); // Default constructor
58. Student student2 = new Student("John Doe", 20); // Parameterized constructor
60. System.out.println("Student 1 (Default constructor):")
61. student1.displayDetails();
63. System.out.println("\nStudent 2 (Parameterized constructor):")
64. student2.displayDetails();
66. // Demonstrate static method and variable
67. System.out.println("\nUniversity Information:")
68. displayUniversity();
69. }
70. }

Output



2, Create a BankAccount class with a private variable balance to store the account balance. Implement a public method deposit(double amount) to add funds, a protected method withdraw(double amount) to deduct funds, and a default-access method checkBalance() to display the current balance. Create an object of the class and demonstrate which methods and variables can be accessed both inside and outside the class.

package Experiment4;

/\*\*

 \* Question: Bank Account System with Access Modifiers

 \*

 \* Create a Java program that implements a basic banking system to demonstrate

 \* the use of different access modifiers and encapsulation principles.

 \*

 \* Requirements:

 \* 1. Implement BankAccount class with:

 \*    - Private balance variable for data hiding

 \*    - Public method for deposits

 \*    - Protected method for withdrawals

 \*    - Default access method for balance checking

 \* 2. Demonstrate:

 \*    - Proper encapsulation of account balance

 \*    - Transaction validation

 \*    - Access level restrictions

 \*

 \* Learning Objectives:

 \* - Understanding access modifiers (public, private, protected, default)

 \* - Implementing encapsulation

 \* - Basic transaction validation

 \* - Class member organization

 \* - Error handling for invalid operations

 \*/

public class BankAccount {

    private double balance; // Private variable

    // Constructor

    public BankAccount() {

        this.balance = 0.0;

    }

    // Public method to deposit money

    public void deposit(double amount) {

        if (amount > 0) {

            balance += amount;

            System.out.println("Deposited: $" + amount);

        } else {

            System.out.println("Invalid deposit amount");

        }

    }

    // Protected method to withdraw money

    protected void withdraw(double amount) {

        if (amount > 0 && amount <= balance) {

            balance -= amount;

            System.out.println("Withdrawn: $" + amount);

        } else {

            System.out.println("Invalid withdrawal amount or insufficient funds");

        }

    }

    // Default access method to check balance

    void checkBalance() {

        System.out.println("Current Balance: $" + balance);

    }

    public static void main(String[] args) {

        BankAccount account = new BankAccount();

        // Test public method

        account.deposit(1000);

        // Test protected method

        account.withdraw(500);

        // Test default access method

        account.checkBalance();

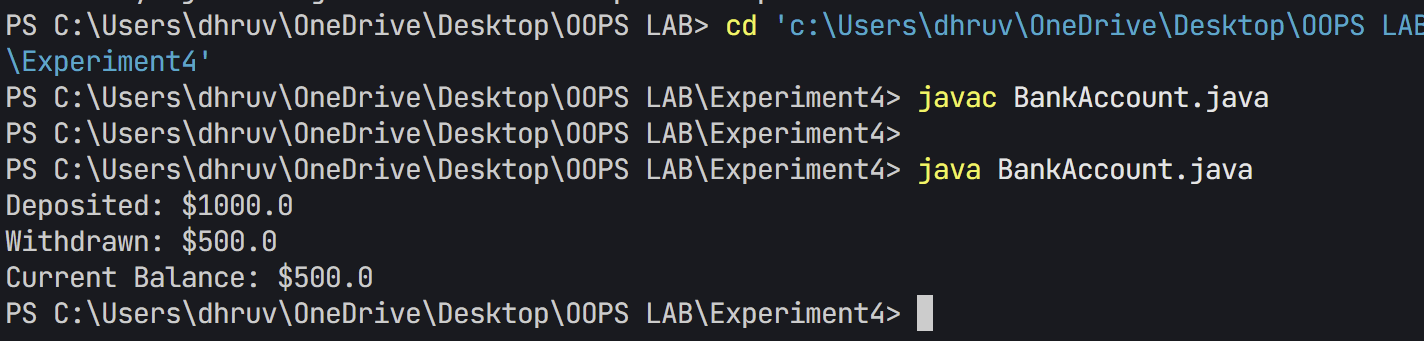
        // Try to access private variable (will not work)

        // System.out.println(account.balance); // This would cause compilation error

    }

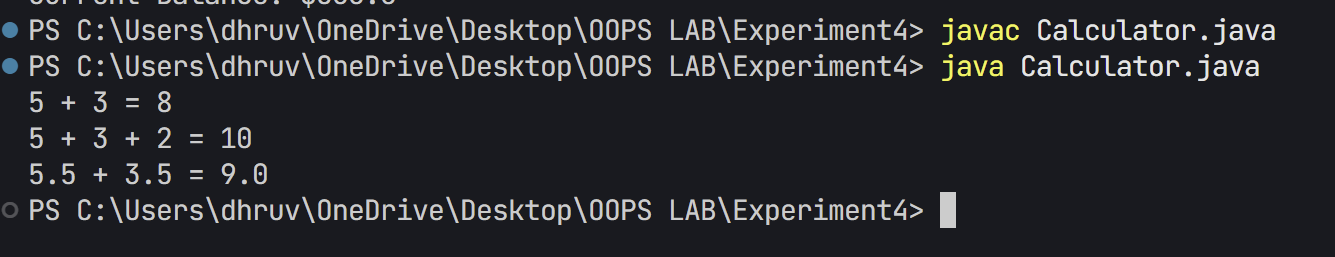
}

Output



1. Create a Calculator class that contains a method add() to perform addition. Overload the add() method to handle different types and numbers of parameters, such as adding two integers, two doubles, and three integers. Create an object of the class and demonstrate all method variations.
2. package Experiment4;
3. /\*\*
4. \* Question: Method Overloading in Calculator Class
5. \*
6. \* Create a Calculator class that demonstrates the concept of method overloading
7. \* by implementing multiple versions of an addition method with different parameters.
8. \*
9. \* Requirements:
10. \* 1. Implement three overloaded 'add' methods:
11. \*    - Method to add two integers
12. \*    - Method to add three integers
13. \*    - Method to add two double values
14. \* 2. Each method should:
15. \*    - Accept appropriate number and type of parameters
16. \*    - Return the sum of the parameters
17. \*    - Use appropriate return type (int or double)
18. \* 3. Demonstrate the usage of all overloaded methods
19. \*
20. \* Learning Objectives:
21. \* - Understanding method overloading concept
22. \* - Working with different data types
23. \* - Method parameter and return type selection
24. \* - Object-oriented programming basics
25. \* - Code organization and readability
26. \*/
27. public class Calculator {
28. // Method to add two integers
29. public int add(int num1, int num2) {
30. return num1 + num2;
31. }
33. // Method to add three integers
34. public int add(int num1, int num2, int num3) {
35. return num1 + num2 + num3;
36. }
38. // Method to add two doubles
39. public double add(double num1, double num2) {
40. return num1 + num2;
41. }
43. public static void main(String[] args) {
44. Calculator calc = new Calculator();
46. // Test two integers addition
47. int result1 = calc.add(5, 3);
48. System.out.println("5 + 3 = " + result1);
50. // Test three integers addition
51. int result2 = calc.add(5, 3, 2);
52. System.out.println("5 + 3 + 2 = " + result2);
54. // Test two doubles addition
55. double result3 = calc.add(5.5, 3.5);
56. System.out.println("5.5 + 3.5 = " + result3);
57. }
58. }

Output



1. Create a Student class that has a static variable universityName and a non-static variable studentName. Include a static method to display the university name. Then, create multiple student objects to demonstrate how the static variable is shared among all instances, while the non-static variable holds unique values for each object.

class ExperimentFour{

    static String universityName = "UPES DEHRADUN";

    String studentName;

    ExperimentFour(String name){

        this.studentName = name;

    }

    static void displayUniversityName(){

        System.out.println("University Name:"+ universityName);

    }

    void displayStudentInfo(){

        System.out.println("Student Name:"+ studentName);

        System.out.println("University Name:"+ universityName);

    }

    public static void main(String[] args) {

        ExperimentFour.displayUniversityName();

        ExperimentFour student1 = new ExperimentFour("Dhruv");

        ExperimentFour student2 = new ExperimentFour("palak");

        System.out.println("\nStudent 1 Details:");

        student1.displayStudentInfo();

        System.out.println("\nStudent 2 Details:");

        student2.displayStudentInfo();

        ExperimentFour.universityName = "DIT UNIVERSITY";

        System.out.println("\nAfter changing University Name:");

        student1.displayStudentInfo();

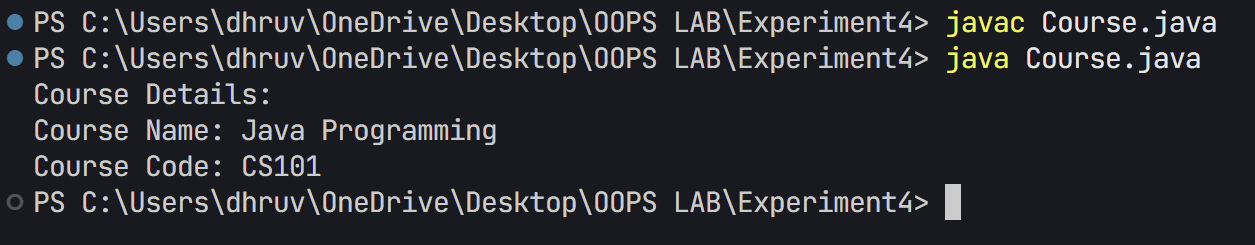
        student2.displayStudentInfo();

    }

}

1. A student is developing a course registration system that allows students to enroll in courses. Each course has a course name and a course code. Implement a Course class with appropriate attributes and use the “this” keyword to differentiate between class attributes and constructor parameters during initialization. Create an object of the Course class and display the course details. [Keep the variable name of formal arguments in constructor and instance variables same.
2. package Experiment4;
3. /\*\*
4. \* Question: Course Registration System with 'this' Keyword
5. \*
6. \* Create a Java program that implements a Course Registration System to demonstrate
7. \* the proper usage of the 'this' keyword in distinguishing between instance variables
8. \* and method parameters.
9. \*
10. \* Requirements:
11. \* 1. Implement Course class with:
12. \*    - Instance variables for course details (name, code)
13. \*    - Constructor using 'this' keyword to handle name shadowing
14. \*    - Methods to display course information
15. \*
16. \* 2. Demonstrate:
17. \*    - Proper parameter to instance variable assignment
18. \*    - Clear usage of 'this' keyword
19. \*    - Proper information display
20. \*
21. \* Learning Objectives:
22. \* - Understanding the 'this' keyword
23. \* - Handling parameter shadowing
24. \* - Proper constructor implementation
25. \* - Basic class design principles
26. \* - Object state management
27. \*/
28. public class Course {
29. private String courseName;
30. private String courseCode;
32. // Constructor using 'this' to differentiate between parameters and instance variables
33. public Course(String courseName, String courseCode) {
34. this.courseName = courseName;
35. this.courseCode = courseCode;
36. }
38. // Method to display course details
39. public void displayCourseDetails() {
40. System.out.println("Course Name: " + this.courseName);
41. System.out.println("Course Code: " + this.courseCode);
42. }
44. public static void main(String[] args) {
45. // Create a course object
46. Course javaProgramming = new Course("Java Programming", "CS101");
48. // Display course details
49. System.out.println("Course Details:");
50. javaProgramming.displayCourseDetails();
51. }
52. }

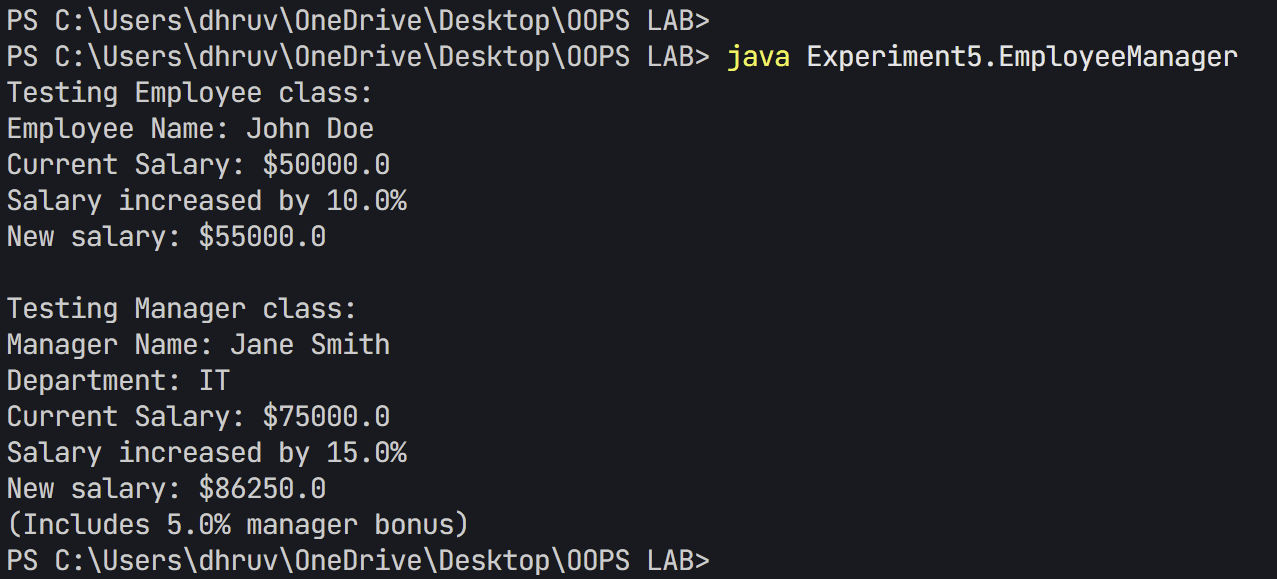
Output



**Additional Question:**

1. Develop an Employee Management System in Java to track employee details such as ID, name, department, and salary, while calculating total salary expenditure and employee count. Create an Employee class with instance variables (employeeID, name, department, and private salary), a static variable totalEmployees to track the employee count, a default constructor for initializing default values, and a parameterized constructor for setting user-provided details. Include a static method to display totalEmployees, a calculateSalary() method to return the salary, and a displayEmployeeInfo() method to show employee details. Use the this keyword in constructors to differentiate class variables from parameters. In the main method, create Employee objects using both constructors, display the total number of employees, and show salary details for each employee.
2. package Experiment4;
3. /\*\*
4. \* Question: Employee Management System with OOP Concepts
5. \*
6. \* Create a Java program that implements an Employee Management System using
7. \* Object-Oriented Programming concepts. The program should demonstrate the use
8. \* of constructors, static members, and encapsulation.
9. \*
10. \* Requirements:
11. \* 1. Implement Employee class with:
12. \*    - Private instance variables (employeeID, name, department, salary)
13. \*    - Static variable for tracking total employees
14. \*    - Default and parameterized constructors
15. \*    - Methods for salary calculation and information display
16. \* 2. Demonstrate:
17. \*    - Object creation using both constructors
18. \*    - Static method usage
19. \*    - Proper encapsulation
20. \*
21. \* Learning Objectives:
22. \* - Understanding class and object concepts
23. \* - Working with constructors
24. \* - Implementing encapsulation
25. \* - Using static members
26. \* - Basic OOP principles
27. \*/
28. public class Employee {
29. private int employeeID;
30. private String name;
31. private String department;
32. private double salary;
33. private static int totalEmployees = 0;
35. // Default constructor
36. public Employee() {
37. this.employeeID = ++totalEmployees;
38. this.name = "Unknown";
39. this.department = "Not Assigned";
40. this.salary = 0.0;
41. }
43. // Parameterized constructor
44. public Employee(int employeeID, String name, String department, double salary) {
45. this.employeeID = employeeID;
46. this.name = name;
47. this.department = department;
48. this.salary = salary;
49. totalEmployees++;
50. }
52. // Static method to display total employees
53. public static void displayTotalEmployees() {
54. System.out.println("Total number of employees: " + totalEmployees);
55. }
57. // Method to calculate and return salary
58. public double calculateSalary() {
59. return this.salary;
60. }
62. // Method to display employee information
63. public void displayEmployeeInfo() {
64. System.out.println("\nEmployee Details:");
65. System.out.println("ID: " + this.employeeID);
66. System.out.println("Name: " + this.name);
67. System.out.println("Department: " + this.department);
68. System.out.println("Salary: $" + this.salary);
69. }
71. public static void main(String[] args) {
72. // Create employees using both constructors
73. Employee emp1 = new Employee(); // Default constructor
74. Employee emp2 = new Employee(2, "John Doe", "IT", 50000.0); // Parameterized constructor
75. Employee emp3 = new Employee(3, "Jane Smith", "HR", 45000.0);
77. // Display total number of employees
78. System.out.println("Employee Management System");
79. displayTotalEmployees();
81. // Display employee details and salaries
82. emp1.displayEmployeeInfo();
83. System.out.println("Calculated Salary: $" + emp1.calculateSalary());
85. emp2.displayEmployeeInfo();
86. System.out.println("Calculated Salary: $" + emp2.calculateSalary());
88. emp3.displayEmployeeInfo();
89. System.out.println("Calculated Salary: $" + emp3.calculateSalary());
90. }
91. }

**Output**

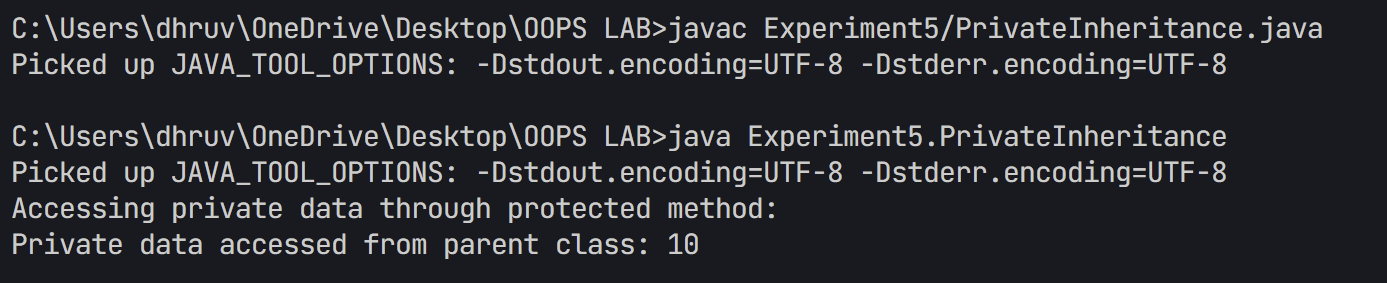
****

## EXPERIMENT – 5

## TITLE Inheritance

1. Write a Java program to demonstrate that a private member of a superclass cannot be accessed directly from a derived class.
2. package Experiment5;
3. /\*
4. Problem Statement:
5. Demonstrate the behavior of private members in inheritance by creating a parent-child class relationship.
6. Show how private members can only be accessed through protected or public methods of the parent class.
7. Requirements:
8. 1. Create a Parent class with private data member
9. 2. Implement protected method to access private data
10. 3. Create Child class that attempts to access private member
11. 4. Demonstrate proper way to access private data through inheritance
12. Key Concepts Demonstrated:
13. - Private Access Modifier
14. - Protected Access Modifier
15. - Inheritance and Encapsulation
16. - Data Hiding
17. - Proper Access Methods
18. \*/
19. class Parent {
20. private int privateData = 10;
22. protected void displayPrivateData() {
23. System.out.println("Private data accessed from parent class: " + privateData);
24. }
25. }
26. class Child extends Parent {
27. void tryToAccessPrivateData() {
28. // This line would cause a compilation error if uncommented
29. // System.out.println("Trying to access private data: " + privateData);
31. // Can only access private data through protected method
32. System.out.println("Accessing private data through protected method:");
33. displayPrivateData();
34. }
35. }
36. public class PrivateInheritance {
37. public static void main(String[] args) {
38. Child child = new Child();
39. child.tryToAccessPrivateData();
40. }
41. }

Output



1. Create a Java program with a Player class and derive three subclasses: Cricket\_Player, Football\_Player, and Hockey\_Player. Implement attributes such as name, age, and position, and methods like play() and train() to represent these players.

package Experiment5;

/\*\*

 \* Question: Sports Player Hierarchy System

 \*

 \* Create a Java program that implements a sports player management system

 \* to demonstrate inheritance, polymorphism, and class hierarchy concepts.

 \*

 \* Requirements:

 \* 1. Implement base Player class with:

 \*    - Common attributes (name, age, position)

 \*    - Basic player actions (play, train)

 \* 2. Create specialized player classes:

 \*    - Cricket\_Player with batting style

 \*    - Football\_Player with jersey number

 \* 3. Demonstrate:

 \*    - Method overriding

 \*    - Inheritance hierarchy

 \*    - Runtime polymorphism

 \*

 \* Learning Objectives:

 \* - Understanding inheritance principles

 \* - Implementing method overriding

 \* - Using protected access modifier

 \* - Demonstrating polymorphic behavior

 \* - Class hierarchy design

 \*/

class Player {

    protected String name;

    protected int age;

    protected String position;

    public Player(String name, int age, String position) {

        this.name = name;

        this.age = age;

        this.position = position;

    }

    public void play() {

        System.out.println(name + " is playing");

    }

    public void train() {

        System.out.println(name + " is training");

    }

}

class Cricket\_Player extends Player {

    private String battingStyle;

    public Cricket\_Player(String name, int age, String position, String battingStyle) {

        super(name, age, position);

        this.battingStyle = battingStyle;

    }

    @Override

    public void play() {

        System.out.println(name + " is playing cricket as a " + position);

    }

    @Override

    public void train() {

        System.out.println(name + " is practicing " + battingStyle + " batting");

    }

}

class Football\_Player extends Player {

    private int jerseyNumber;

    public Football\_Player(String name, int age, String position, int jerseyNumber) {

        super(name, age, position);

        this.jerseyNumber = jerseyNumber;

    }

    @Override

    public void play() {

        System.out.println(name + " is playing football with jersey number " + jerseyNumber);

    }

    @Override

    public void train() {

        System.out.println(name + " is practicing football drills");

    }

}

class Hockey\_Player extends Player {

    private String stickType;

    public Hockey\_Player(String name, int age, String position, String stickType) {

        super(name, age, position);

        this.stickType = stickType;

    }

    @Override

    public void play() {

        System.out.println(name + " is playing hockey using " + stickType + " stick");

    }

    @Override

    public void train() {

        System.out.println(name + " is practicing hockey techniques");

    }

}

public class PlayerHierarchy {

    public static void main(String[] args) {

        Cricket\_Player cricketer = new Cricket\_Player("Virat", 32, "Batsman", "Right-handed");

        Football\_Player footballer = new Football\_Player("Messi", 34, "Forward", 10);

        Hockey\_Player hockey = new Hockey\_Player("Singh", 28, "Center", "Composite");

        // Demonstrating polymorphism

        Player[] players = {cricketer, footballer, hockey};

        for (Player player : players) {

            player.play();

            player.train();

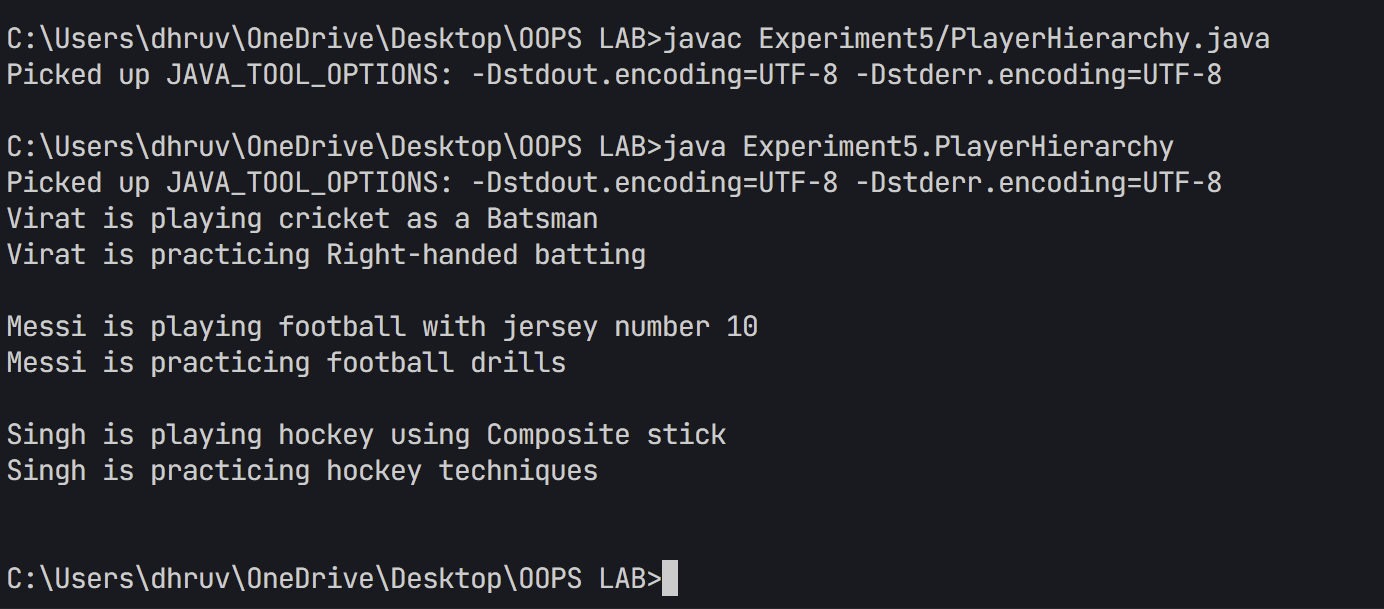
            System.out.println();

        }

    }

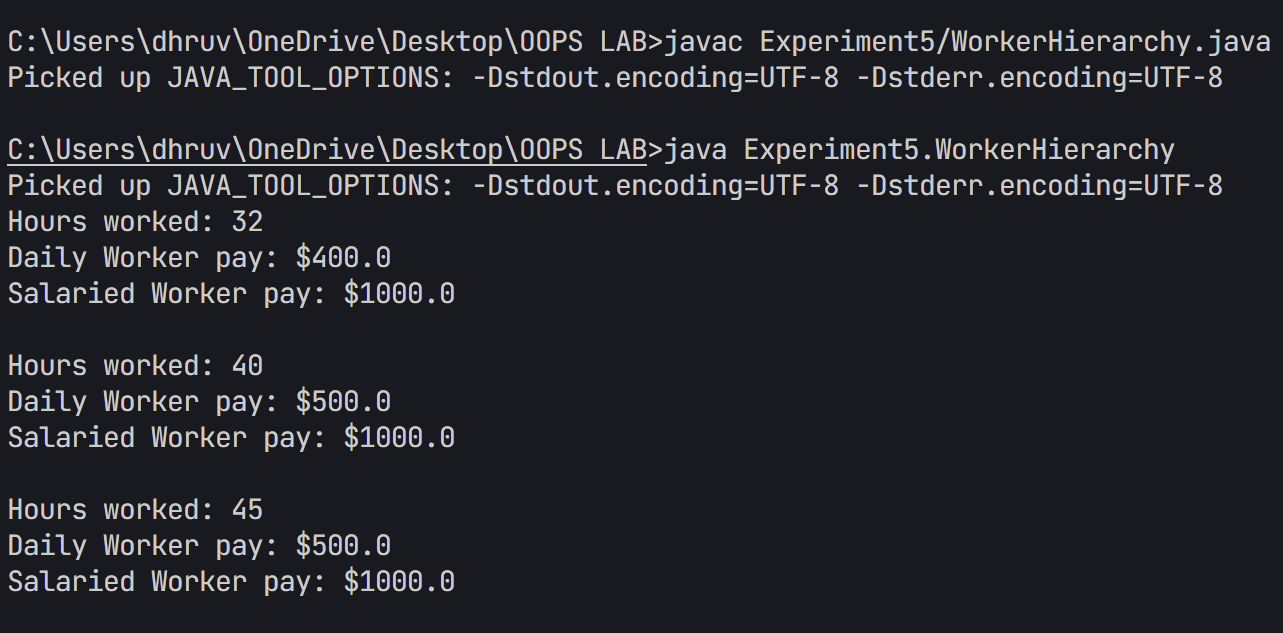
}

Output



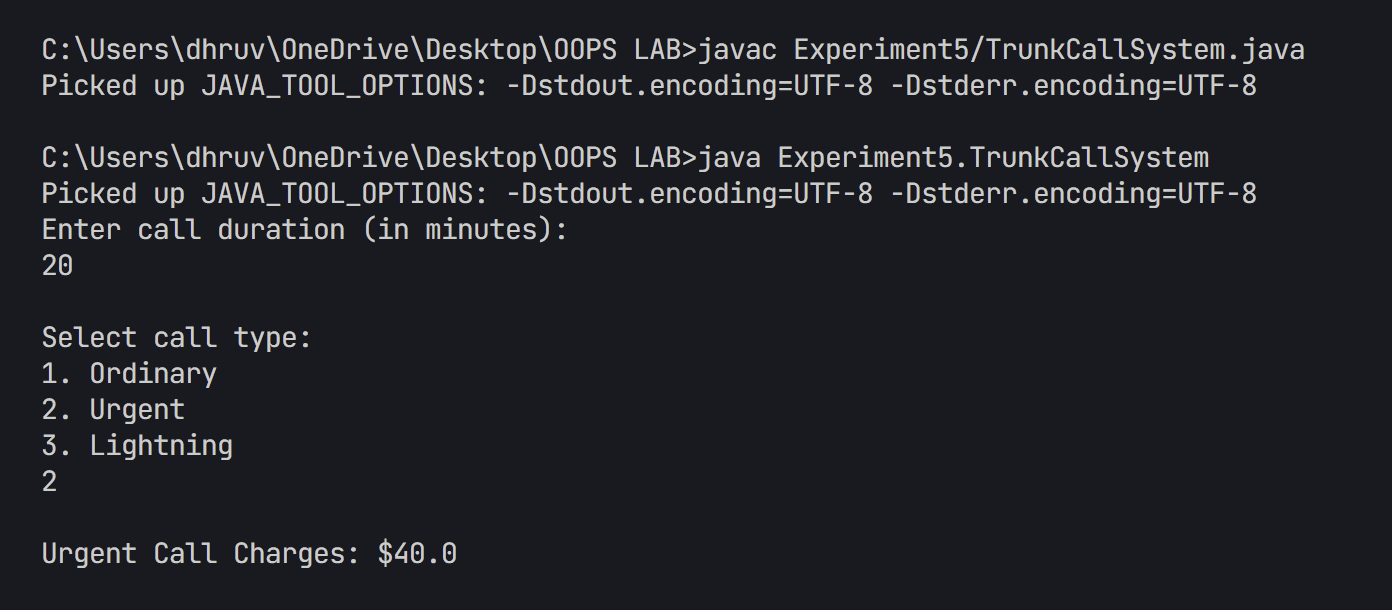
1. Define a Worker class with DailyWorker and SalariedWorker as its subclasses. Each worker has a name and salary rate. Implement a method computePay(int hours) to compute weekly pay. DailyWorker is paid based on the number of days worked (assuming 8 hours per day), whereas SalariedWorker receives a fixed wage for 40 hours per week, regardless of actual hours worked. Use polymorphism to implement this program and test worker salary calculations.
2. package Experiment5;
3. /\*
4. Problem Statement:
5. Implement a worker payment system that demonstrates the use of abstract classes and inheritance.
6. The system should handle two types of workers:
7. 1. Daily workers who are paid based on the number of days worked (8 hours = 1 day)
8. 2. Salaried workers who receive a fixed weekly salary regardless of extra hours
9. Requirements:
10. 1. Create an abstract Worker class with name and salary rate attributes
11. 2. Implement concrete DailyWorker and SalariedWorker classes
12. 3. Use abstract method for pay computation with different implementations
13. 4. Test the system with various working hours
14. Key Concepts Demonstrated:
15. - Abstract Classes and Methods
16. - Inheritance and Method Overriding
17. - Polymorphism through abstract class reference
18. - Different salary calculation strategies
19. - Constructor chaining using super()
20. \*/
21. // Abstract base class for all workers
22. abstract class Worker {
23. protected String name;
24. protected double salaryRate;
26. public Worker(String name, double salaryRate) {
27. this.name = name;
28. this.salaryRate = salaryRate;
29. }
31. // Abstract method to be implemented by subclasses
32. public abstract double computePay(int hours);
33. }
34. class DailyWorker extends Worker {
35. public DailyWorker(String name, double dailyRate) {
36. super(name, dailyRate);
37. }
39. @Override
40. public double computePay(int hours) {
41. // Calculate days worked (assuming 8 hours per day)
42. int daysWorked = hours / 8;
43. return daysWorked \* salaryRate;
44. }
45. }
46. class SalariedWorker extends Worker {
47. public SalariedWorker(String name, double weeklyRate) {
48. super(name, weeklyRate);
49. }
51. @Override
52. public double computePay(int hours) {
53. // Salaried workers get fixed weekly pay regardless of hours
54. // as long as they work at least 40 hours
55. return salaryRate;
56. }
57. }
58. public class WorkerHierarchy {
59. public static void main(String[] args) {
60. Worker dailyWorker = new DailyWorker("John", 100.0); // $100 per day
61. Worker salariedWorker = new SalariedWorker("Jane", 1000.0); // $1000 per week
63. // Test with different hours
64. int[] hoursWorked = {32, 40, 45};
66. for (int hours : hoursWorked) {
67. System.out.println("Hours worked: " + hours);
68. System.out.println("Daily Worker pay: $" + dailyWorker.computePay(hours));
69. System.out.println("Salaried Worker pay: $" + salariedWorker.computePay(hours));
70. System.out.println();
71. }
72. }
73. }

Output



1. Implement a Java program to calculate trunk call charges based on duration and type (Ordinary, Urgent, or Lightning). Use polymorphism to manage different charge rates for each type. Implement a Java program to calculate trunk call charges based on duration (in minutes) and type (Ordinary, Urgent, or Lightning). Use polymorphism to manage different charge rates for each type. The program should take user input for duration and type and display the total charge.
2. package Experiment5;
3. import java.util.Scanner;
4. /\*
5. Problem Statement:
6. Implement a Trunk Call Billing System that calculates charges for different types of calls.
7. The system should support three types of calls with different rate calculations:
8. 1. Ordinary calls (base rate)
9. 2. Urgent calls (double rate)
10. 3. Lightning calls (triple rate)
11. Requirements:
12. 1. Create an abstract base class for trunk calls
13. 2. Implement three different call types with specific rate calculations
14. 3. Use runtime polymorphism for charge calculation
15. 4. Accept user input for call duration and type
16. 5. Display calculated charges based on call type
17. Key Concepts Demonstrated:
18. - Abstract Classes and Methods
19. - Inheritance Hierarchy
20. - Runtime Polymorphism
21. - Method Overriding
22. - User Input Processing
23. - Rate Calculation Strategies
24. \*/
25. // Abstract base class for all call types
26. abstract class TrunkCall {
27. protected int duration;
28. protected double ratePerMinute;
30. public TrunkCall(int duration) {
31. this.duration = duration;
32. }
34. public abstract double calculateCharges();
35. }
36. class OrdinaryCall extends TrunkCall {
37. public OrdinaryCall(int duration) {
38. super(duration);
39. this.ratePerMinute = 1.0; // Base rate for ordinary calls
40. }
42. @Override
43. public double calculateCharges() {
44. return duration \* ratePerMinute;
45. }
46. }
47. class UrgentCall extends TrunkCall {
48. public UrgentCall(int duration) {
49. super(duration);
50. this.ratePerMinute = 2.0; // Double rate for urgent calls
51. }
53. @Override
54. public double calculateCharges() {
55. return duration \* ratePerMinute;
56. }
57. }
58. class LightningCall extends TrunkCall {
59. public LightningCall(int duration) {
60. super(duration);
61. this.ratePerMinute = 3.0; // Triple rate for lightning calls
62. }
64. @Override
65. public double calculateCharges() {
66. return duration \* ratePerMinute;
67. }
68. }
69. public class TrunkCallSystem {
70. public static void main(String[] args) {
71. Scanner scanner = new Scanner(System.in);
73. System.out.println("Enter call duration (in minutes): ");
74. int duration = scanner.nextInt();
76. System.out.println("\nSelect call type:");
77. System.out.println("1. Ordinary");
78. System.out.println("2. Urgent");
79. System.out.println("3. Lightning");
80. int choice = scanner.nextInt();
82. TrunkCall call;
83. switch (choice) {
84. case 1:
85. call = new OrdinaryCall(duration);
86. System.out.println("\nOrdinary Call Charges: $" + call.calculateCharges());
87. break;
88. case 2:
89. call = new UrgentCall(duration);
90. System.out.println("\nUrgent Call Charges: $" + call.calculateCharges());
91. break;
92. case 3:
93. call = new LightningCall(duration);
94. System.out.println("\nLightning Call Charges: $" + call.calculateCharges());
95. break;
96. default:
97. System.out.println("Invalid choice!");
98. }
100. scanner.close();
101. }
102. }

Output



1. Design a Java class Employee with attributes name, empid, and salary. Implement a default constructor, a parameterized constructor, and methods to return the employee’s name and salary. Add a method increaseSalary(double percentage) to raise the salary by a user-specified percentage. Create a subclass Manager with an additional instance variable department. Develop a test program to validate these functionalities.

package Experiment5;

/\*

Problem Statement:

Implement a worker payment system that demonstrates the use of abstract classes and inheritance.

The system should handle two types of workers:

1. Daily workers who are paid based on the number of days worked (8 hours = 1 day)

2. Salaried workers who receive a fixed weekly salary regardless of extra hours

Requirements:

1. Create an abstract Worker class with name and salary rate attributes

2. Implement concrete DailyWorker and SalariedWorker classes

3. Use abstract method for pay computation with different implementations

4. Test the system with various working hours

Key Concepts Demonstrated:

- Abstract Classes and Methods

- Inheritance and Method Overriding

- Polymorphism through abstract class reference

- Different salary calculation strategies

- Constructor chaining using super()

\*/

// Abstract base class for all workers

abstract class Worker {

    protected String name;

    protected double salaryRate;

    public Worker(String name, double salaryRate) {

        this.name = name;

        this.salaryRate = salaryRate;

    }

    // Abstract method to be implemented by subclasses

    public abstract double computePay(int hours);

}

class DailyWorker extends Worker {

    public DailyWorker(String name, double dailyRate) {

        super(name, dailyRate);

    }

    @Override

    public double computePay(int hours) {

        // Calculate days worked (assuming 8 hours per day)

        int daysWorked = hours / 8;

        return daysWorked \* salaryRate;

    }

}

class SalariedWorker extends Worker {

    public SalariedWorker(String name, double weeklyRate) {

        super(name, weeklyRate);

    }

    @Override

    public double computePay(int hours) {

        // Salaried workers get fixed weekly pay regardless of hours

        // as long as they work at least 40 hours

        return salaryRate;

    }

}

public class WorkerHierarchy {

    public static void main(String[] args) {

        Worker dailyWorker = new DailyWorker("John", 100.0); // $100 per day

        Worker salariedWorker = new SalariedWorker("Jane", 1000.0); // $1000 per week

        // Test with different hours

        int[] hoursWorked = {32, 40, 45};

        for (int hours : hoursWorked) {

            System.out.println("Hours worked: " + hours);

            System.out.println("Daily Worker pay: $" + dailyWorker.computePay(hours));

            System.out.println("Salaried Worker pay: $" + salariedWorker.computePay(hours));

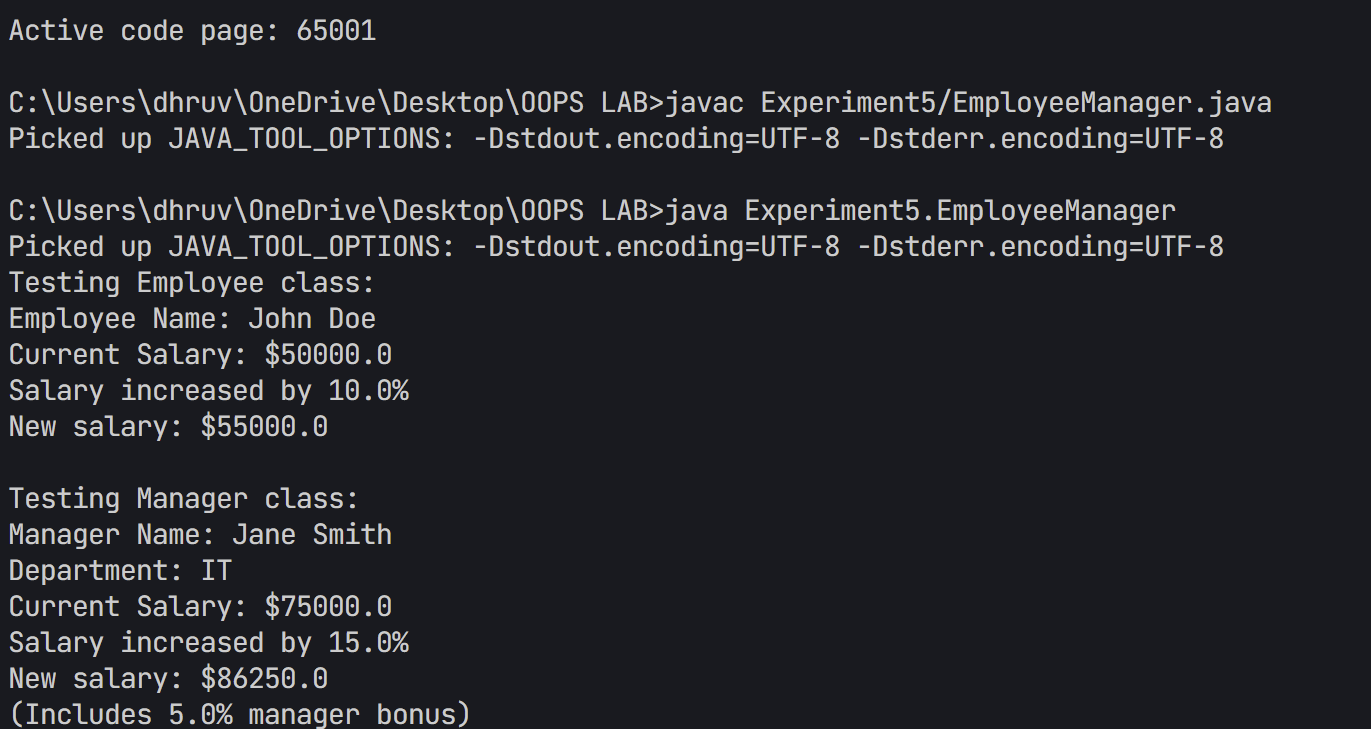
            System.out.println();

        }

    }

}

Output

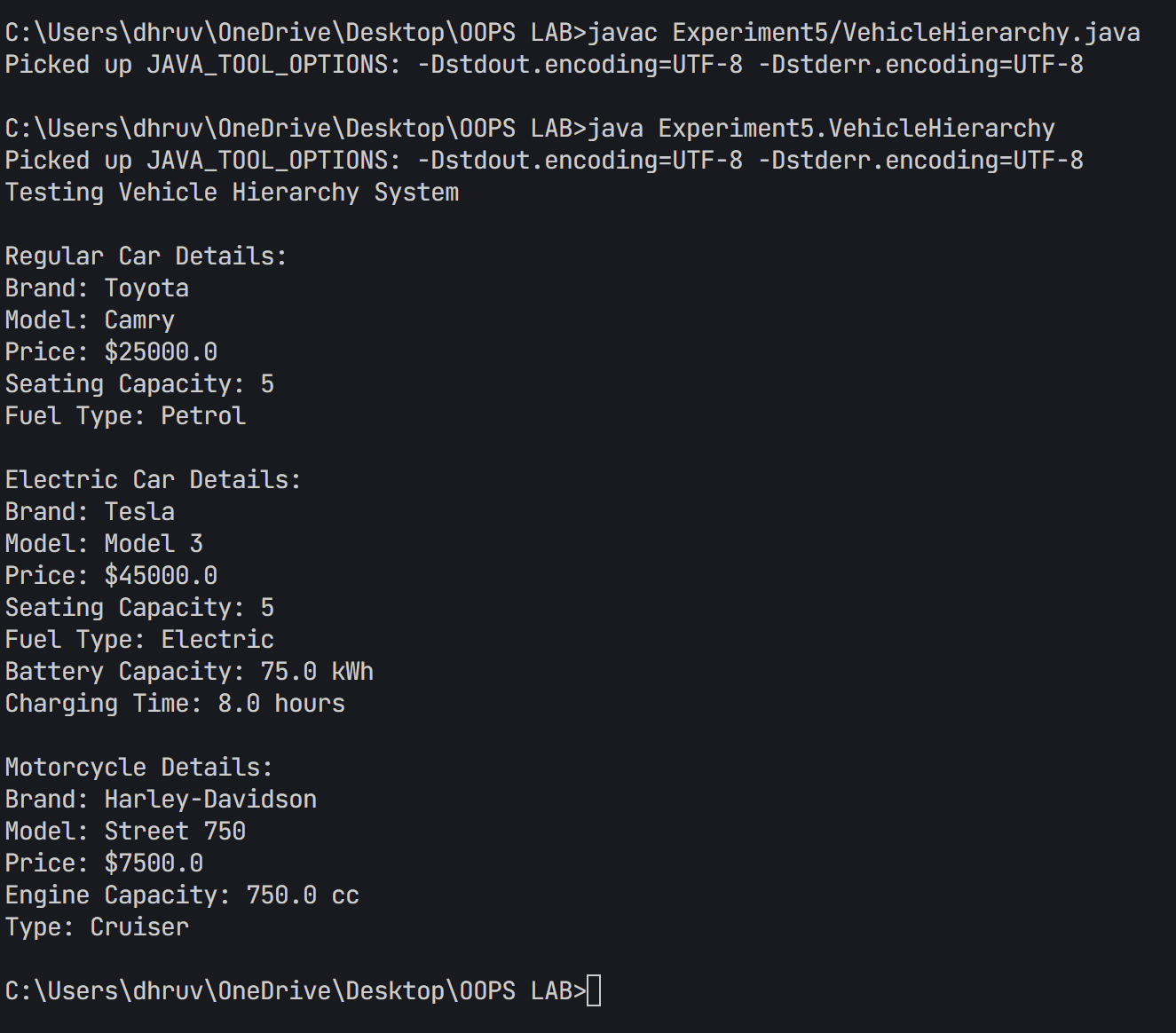


**Additional Questions:**

* 1. A vehicle manufacturing company produces different types of vehicles, such as cars and motorcycles. The base class Vehicle contains common properties like brand, model, and price. The class Car extends Vehicle by adding attributes like seatingCapacity and fuelType. Further, a subclass ElectricCar extends Car, introducing additional attributes like batteryCapacity and chargingTime. The Motorcycle class extends Vehicle and adds engineCapacity and type (e.g., "Sport", "Cruiser"). Implement this vehicle hierarchy system using multilevel inheritance in Java. Use constructor chaining to initialize attributes efficiently and demonstrate polymorphism by overriding a method displayDetails() in each subclass.

1. package Experiment5;
2. /\*
3. Problem Statement:
4. Implement a Vehicle Management System that demonstrates hierarchical inheritance
5. with different types of vehicles including cars, electric cars, and motorcycles.
6. Requirements:
7. 1. Create a base Vehicle class with common attributes (brand, model, price)
8. 2. Implement Car class extending Vehicle with car-specific features
9. 3. Create ElectricCar class extending Car with electric vehicle properties
10. 4. Implement Motorcycle class extending Vehicle with bike-specific attributes
11. 5. Demonstrate proper inheritance relationships and method overriding
12. Key Concepts Demonstrated:
13. - Hierarchical Inheritance (Vehicle -> Car/Motorcycle)
14. - Multilevel Inheritance (Vehicle -> Car -> ElectricCar)
15. - Method Overriding
16. - Constructor Chaining
17. - Protected and Private Access Modifiers
18. - Specialized Class Attributes
19. \*/
20. // Base class Vehicle
21. class Vehicle {
22. protected String brand;
23. protected String model;
24. protected double price;
26. // Default constructor
27. public Vehicle() {
28. this.brand = "Unknown";
29. this.model = "Unknown";
30. this.price = 0.0;
31. }
33. // Parameterized constructor
34. public Vehicle(String brand, String model, double price) {
35. this.brand = brand;
36. this.model = model;
37. this.price = price;
38. }
40. // Method to display details
41. public void displayDetails() {
42. System.out.println("Brand: " + brand);
43. System.out.println("Model: " + model);
44. System.out.println("Price: $" + price);
45. }
46. }
47. // Car class extending Vehicle
48. class Car extends Vehicle {
49. private int seatingCapacity;
50. private String fuelType;
52. // Default constructor
53. public Car() {
54. super();
55. this.seatingCapacity = 0;
56. this.fuelType = "Unknown";
57. }
59. // Parameterized constructor
60. public Car(String brand, String model, double price, int seatingCapacity, String fuelType) {
61. super(brand, model, price);
62. this.seatingCapacity = seatingCapacity;
63. this.fuelType = fuelType;
64. }
66. // Override displayDetails method
67. @Override
68. public void displayDetails() {
69. super.displayDetails();
70. System.out.println("Seating Capacity: " + seatingCapacity);
71. System.out.println("Fuel Type: " + fuelType);
72. }
73. }
74. // ElectricCar class extending Car
75. class ElectricCar extends Car {
76. private double batteryCapacity;
77. private double chargingTime;
79. // Default constructor
80. public ElectricCar() {
81. super();
82. this.batteryCapacity = 0.0;
83. this.chargingTime = 0.0;
84. }
86. // Parameterized constructor
87. public ElectricCar(String brand, String model, double price, int seatingCapacity,
88. double batteryCapacity, double chargingTime) {
89. super(brand, model, price, seatingCapacity, "Electric");
90. this.batteryCapacity = batteryCapacity;
91. this.chargingTime = chargingTime;
92. }
94. // Override displayDetails method
95. @Override
96. public void displayDetails() {
97. super.displayDetails();
98. System.out.println("Battery Capacity: " + batteryCapacity + " kWh");
99. System.out.println("Charging Time: " + chargingTime + " hours");
100. }
101. }
102. // Motorcycle class extending Vehicle
103. class Motorcycle extends Vehicle {
104. private double engineCapacity;
105. private String type;
107. // Default constructor
108. public Motorcycle() {
109. super();
110. this.engineCapacity = 0.0;
111. this.type = "Unknown";
112. }
114. // Parameterized constructor
115. public Motorcycle(String brand, String model, double price, double engineCapacity, String type) {
116. super(brand, model, price);
117. this.engineCapacity = engineCapacity;
118. this.type = type;
119. }
121. // Override displayDetails method
122. @Override
123. public void displayDetails() {
124. super.displayDetails();
125. System.out.println("Engine Capacity: " + engineCapacity + " cc");
126. System.out.println("Type: " + type);
127. }
128. }
129. // Main class to test the vehicle hierarchy
130. public class VehicleHierarchy {
131. public static void main(String[] args) {
132. System.out.println("Testing Vehicle Hierarchy System\n");
134. // Test Car
135. System.out.println("Regular Car Details:");
136. Car car = new Car("Toyota", "Camry", 25000.0, 5, "Petrol");
137. car.displayDetails();
138. System.out.println();
140. // Test ElectricCar
141. System.out.println("Electric Car Details:");
142. ElectricCar electricCar = new ElectricCar("Tesla", "Model 3", 45000.0, 5, 75.0, 8.0);
143. electricCar.displayDetails();
144. System.out.println();
146. // Test Motorcycle
147. System.out.println("Motorcycle Details:");
148. Motorcycle motorcycle = new Motorcycle("Harley-Davidson", "Street 750", 7500.0, 750.0, "Cruiser");
149. motorcycle.displayDetails();
150. }
151. }

Output



1. A university has different types of people associated with it, including staff members and students. The base class Person contains common attributes such as name, age, and address.The class Staff extends Person and adds attributes like staffId and department. Further, a subclass Professor extends Staff by introducing an additional attribute, specialization, and a method conductLecture(). Similarly, the Student class extends Person and adds studentId and course. Finally, the subclass GraduateStudent extends Student, adding researchTopic and a method submitThesis(). Implement this university management system in Java using multilevel inheritance and method overriding. Demonstrate polymorphism by creating an array of Person objects containing instances of Professor and GraduateStudent, and call their respective methods.

package Experiment5;

/\*

Problem Statement:

Implement a University Management System that demonstrates multilevel inheritance

and hierarchical relationships between different types of university members.

Requirements:

1. Create a base Person class with common attributes (name, age, address)

2. Implement Staff class extending Person with additional staff-specific details

3. Create Professor class extending Staff with teaching capabilities

4. Implement Student class extending Person with student-specific attributes

5. Demonstrate proper constructor chaining and method overriding

Key Concepts Demonstrated:

- Multilevel Inheritance (Person -> Staff -> Professor)

- Hierarchical Inheritance (Person -> Student)

- Method Overriding

- Constructor Chaining

- Protected Access Modifier

- Polymorphic Behavior

\*/

// Base class Person

class Person {

    protected String name;

    protected int age;

    protected String address;

    // Default constructor

    public Person() {

        this.name = "Unknown";

        this.age = 0;

        this.address = "Unknown";

    }

    // Parameterized constructor

    public Person(String name, int age, String address) {

        this.name = name;

        this.age = age;

        this.address = address;

    }

    // Method to display details

    public void displayDetails() {

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

        System.out.println("Age: " + age);

        System.out.println("Address: " + address);

    }

}

// Staff class extending Person

class Staff extends Person {

    protected String staffId;

    protected String department;

    // Default constructor

    public Staff() {

        super();

        this.staffId = "Unknown";

        this.department = "Unknown";

    }

    // Parameterized constructor

    public Staff(String name, int age, String address, String staffId, String department) {

        super(name, age, address);

        this.staffId = staffId;

        this.department = department;

    }

    // Override displayDetails method

    @Override

    public void displayDetails() {

        super.displayDetails();

        System.out.println("Staff ID: " + staffId);

        System.out.println("Department: " + department);

    }

}

// Professor class extending Staff

class Professor extends Staff {

    private String specialization;

    // Default constructor

    public Professor() {

        super();

        this.specialization = "Unknown";

    }

    // Parameterized constructor

    public Professor(String name, int age, String address, String staffId,

                    String department, String specialization) {

        super(name, age, address, staffId, department);

        this.specialization = specialization;

    }

    // Method to conduct lecture

    public void conductLecture() {

        System.out.println("Professor " + name + " is conducting a lecture in " + specialization);

    }

    // Override displayDetails method

    @Override

    public void displayDetails() {

        super.displayDetails();

        System.out.println("Specialization: " + specialization);

    }

}

// Student class extending Person

class Student extends Person {

    protected String studentId;

    protected String course;

    // Default constructor

    public Student() {

        super();

        this.studentId = "Unknown";

        this.course = "Unknown";

    }

    // Parameterized constructor

    public Student(String name, int age, String address, String studentId, String course) {

        super(name, age, address);

        this.studentId = studentId;

        this.course = course;

    }

    // Override displayDetails method

    @Override

    public void displayDetails() {

        super.displayDetails();

        System.out.println("Student ID: " + studentId);

        System.out.println("Course: " + course);

    }

}

// GraduateStudent class extending Student

class GraduateStudent extends Student {

    private String researchTopic;

    // Default constructor

    public GraduateStudent() {

        super();

        this.researchTopic = "Unknown";

    }

    // Parameterized constructor

    public GraduateStudent(String name, int age, String address, String studentId,

                          String course, String researchTopic) {

        super(name, age, address, studentId, course);

        this.researchTopic = researchTopic;

    }

    // Method to submit thesis

    public void submitThesis() {

        System.out.println("Graduate Student " + name + " is submitting thesis on " + researchTopic);

    }

    // Override displayDetails method

    @Override

    public void displayDetails() {

        super.displayDetails();

        System.out.println("Research Topic: " + researchTopic);

    }

}

// Main class to test the university system

public class UniversitySystem {

    public static void main(String[] args) {

        System.out.println("Testing University Personnel System\n");

        // Create an array of Person objects

        Person[] universityPersonnel = new Person[2];

        // Initialize with Professor and GraduateStudent instances

        universityPersonnel[0] = new Professor("Dr. Smith", 45, "123 Academic Ave",

                                              "PROF001", "Computer Science", "Artificial Intelligence");

        universityPersonnel[1] = new GraduateStudent("Jane Doe", 25, "456 Research Hall",

                                                    "GS001", "Computer Science", "Machine Learning");

        // Demonstrate polymorphism

        for (Person person : universityPersonnel) {

            System.out.println("\nDetails for University Personnel:");

            person.displayDetails();

            // Use instanceof to call specific methods

            if (person instanceof Professor) {

                ((Professor) person).conductLecture();

            } else if (person instanceof GraduateStudent) {

                ((GraduateStudent) person).submitThesis();

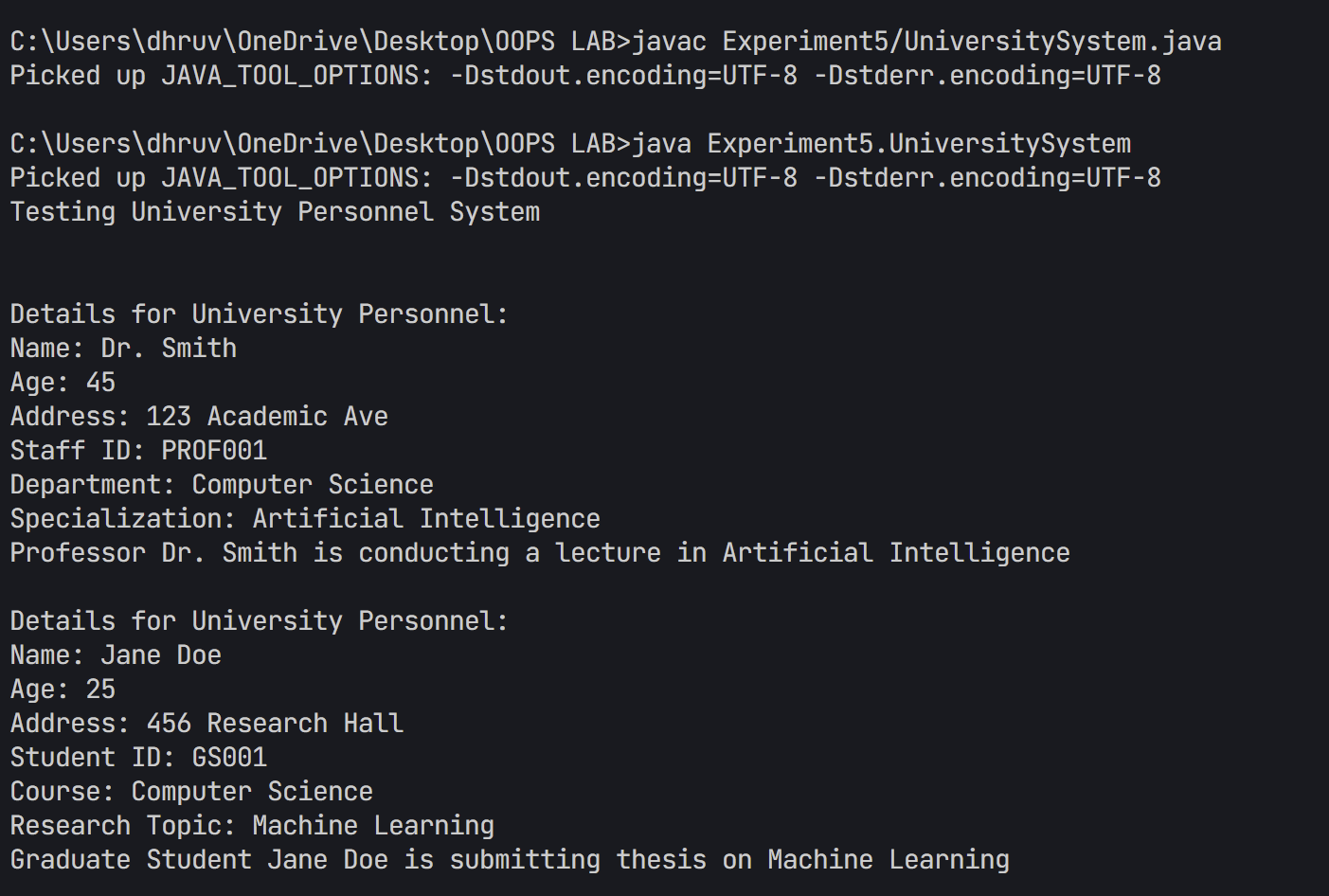
            }

        }

    }

}

Output



## EXPERIMENT – 6

## TITLE Packages & use of final

1. Create a package named Balance containing a class Account with a method Display\_Balance that displays the account balance. Write another program to import the Balance package and call the Display\_Balance method from the Account class.
2. package Experiment6.Balance;
3. /\*\*
4. \* Question: Package Creation and Access Control in Java
5. \*
6. \* Create a Java program that demonstrates the creation and usage of packages
7. \* along with proper access control mechanisms. Implement a banking account
8. \* system where account details are encapsulated within a package.
9. \*
10. \* Requirements:
11. \* 1. Create a package named 'Balance' to contain account-related classes
12. \* 2. Implement Account class with:
13. \*    - Private balance field for data hiding
14. \*    - Public constructor for initialization
15. \*    - Public method to display balance
16. \* 3. Demonstrate:
17. \*    - Package creation and organization
18. \*    - Access modifiers usage
19. \*    - Encapsulation principles
20. \*
21. \* Learning Objectives:
22. \* - Understanding package creation
23. \* - Implementing access control
24. \* - Applying encapsulation
25. \* - Managing class visibility
26. \*/
27. public class Account {
28. private double balance;
29. public Account(double initialBalance) {
30. this.balance = initialBalance;
31. }
32. public void Display\_Balance() {
33. System.out.println("Current Account Balance: $" + balance);
34. }
35. Create a package p containing a class A with four methods, each having different access modifiers (e.g., public, protected, default, and private). Write another class B in a different package Q with a main() method. Demonstrate how to access the methods of class A from class B, considering the access modifiers.
36. package p;
37. /\*\*
38. \* Question: Access Modifiers and Package Interaction
39. \*
40. \* Create a Java program that demonstrates the behavior of different access modifiers
41. \* when accessing class members across packages. Implement two classes in separate
42. \* packages to show access restrictions.
43. \*
44. \* Requirements:
45. \* 1. Create two packages 'p' and 'Q'
46. \* 2. Implement class A in package 'p' with:
47. \*    - Methods using all access modifiers (public, protected, default, private)
48. \*    - Clear demonstration of each modifier's scope
49. \* 3. Create class B in package 'Q' that attempts to access A's members
50. \* 4. Show:
51. \*    - Which members are accessible across packages
52. \*    - Compilation errors for restricted access
53. \*
54. \* Learning Objectives:
55. \* - Understanding access modifiers
56. \* - Package-level access control
57. \* - Cross-package member access
58. \* - Visibility rules in Java
59. \*/
60. public class A {
61. public void publicMethod() {
62. System.out.println("This is a public method");
63. }
65. protected void protectedMethod() {
66. System.out.println("This is a protected method");
67. }
69. void defaultMethod() {
70. System.out.println("This is a default (package-private) method");
71. }
73. private void privateMethod() {
74. System.out.println("This is a private method");
75. }

package Q;

import p.A;

public class B {

    public static void main(String[] args) {

        A objA = new A();

        System.out.println("Accessing methods from class A:");

        // Public method - accessible everywhere

        objA.publicMethod();

        // Protected method - not accessible in different package without inheritance

        // objA.protectedMethod(); // This will cause compilation error

        // Default method - not accessible in different package

        // objA.defaultMethod(); // This will cause compilation error

        // Private method - not accessible outside class A

        // objA.privateMethod(); // This will cause compilation error

        System.out.println("\nNote: Protected, default, and private methods are not accessible");

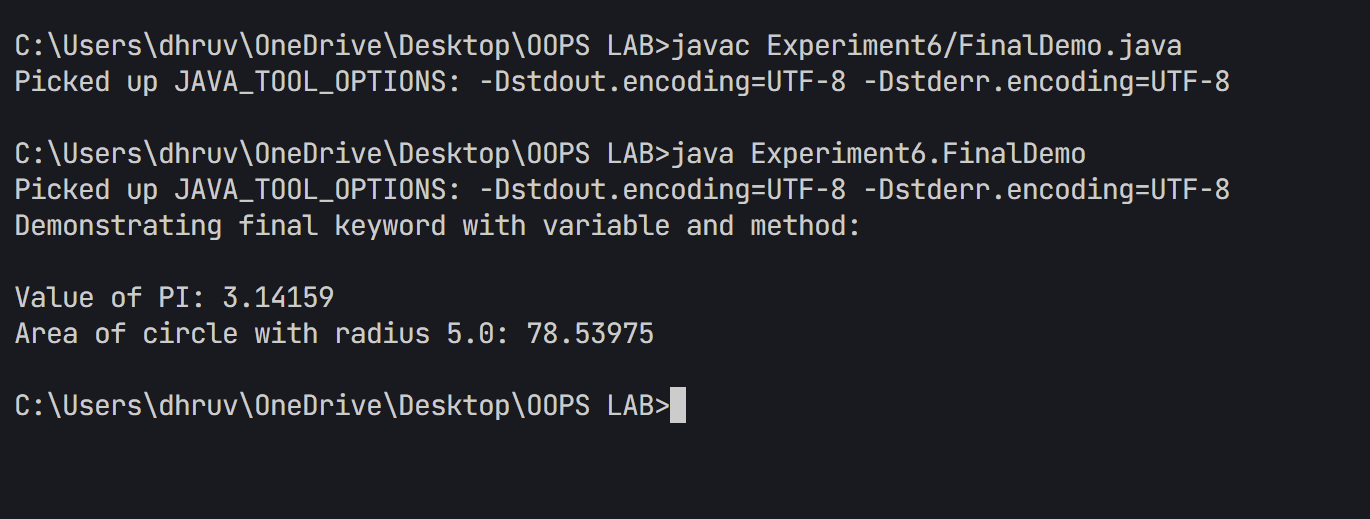
        System.out.println("due to access modifier restrictions across packages.");

    }

}

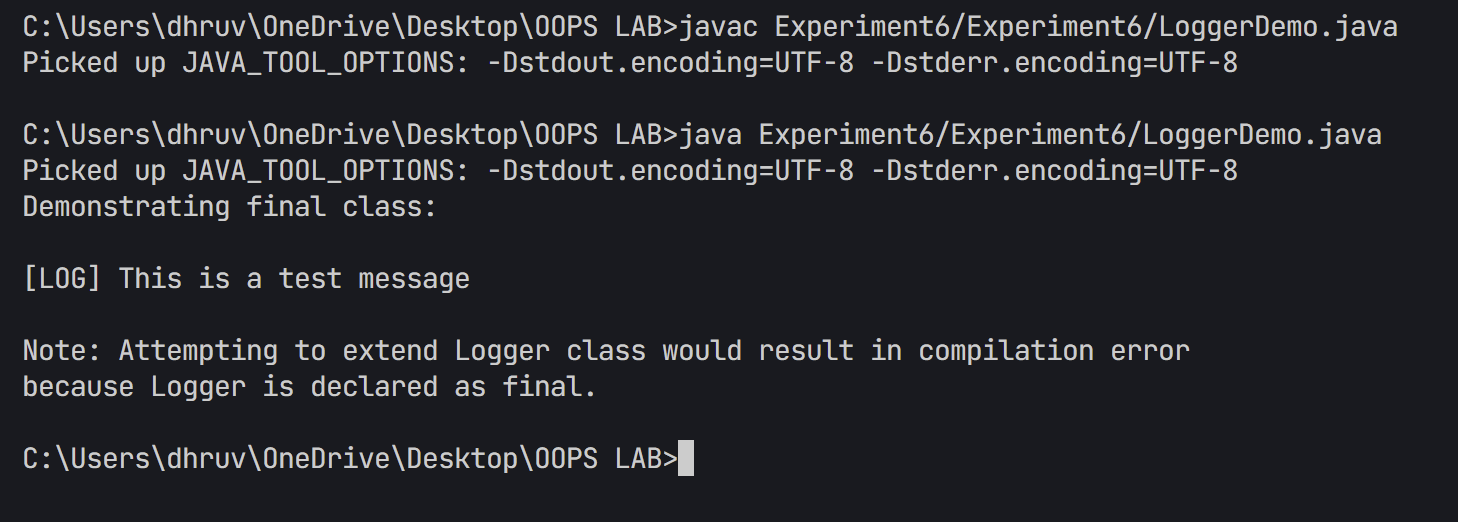
1. Write a Java program to demonstrate the use of the final keyword with a variable and a method. Create a class MathConstants with a final variable PI (value = 3.14159) and a final method displayPI() that prints the value of PI. Create another class Circle that extends MathConstants and includes a method calculateArea(double radius) to calculate and print the area of a circle using the formula: area = PI \* radius \* radius. In the main() method (in a separate class or Circle), test the calculateArea() method and observe the behavior when attempting to modify the PI variable or override the displayPI() method.
2. package Experiment6;
3. /\*\*
4. \* Question: Understanding Final Keyword in Java
5. \*
6. \* Create a Java program that demonstrates the usage and implications of the final
7. \* keyword in different contexts - variables, methods, and classes.
8. \*
9. \* Requirements:
10. \* 1. Demonstrate final keyword with:
11. \*    - Final variable (constant)
12. \*    - Final method that cannot be overridden
13. \*    - Class inheritance restrictions
14. \* 2. Implement:
15. \*    - A class with final constant
16. \*    - Method attempting to modify final variable
17. \*    - Inheritance scenario with final method
18. \* 3. Show:
19. \*    - Compilation errors when trying to modify final members
20. \*    - Proper usage in calculations
21. \*
22. \* Learning Objectives:
23. \* - Understanding final keyword usage
24. \* - Constant declaration and usage
25. \* - Method overriding restrictions
26. \* - Inheritance limitations
27. \* - Java compilation error handling
28. \*/
29. class MathConstants {
30. final double PI = 3.14159;
32. final void displayPI() {
33. System.out.println("Value of PI: " + PI);
34. }
35. }
36. class Circle extends MathConstants {
37. public void calculateArea(double radius) {
38. // Cannot modify PI as it is final
39. // PI = 3.14; // This would cause compilation error
41. double area = PI \* radius \* radius;
42. System.out.println("Area of circle with radius " + radius + ": " + area);
43. }
45. // Cannot override displayPI() as it is final
46. // void displayPI() { // This would cause compilation error
47. //     System.out.println("Attempting to override final method");
48. // }
49. }
50. public class FinalDemo {
51. public static void main(String[] args) {
52. Circle circle = new Circle();
54. System.out.println("Demonstrating final keyword with variable and method:\n");
56. // Display the value of PI
57. circle.displayPI();
59. // Calculate and display area
60. circle.calculateArea(5.0);
61. }
62. }

Output



1. Write a Java program to demonstrate the use of the final keyword with a class. Create a final class Logger with a method logMessage(String message) that prints the message to the console. Attempt to create another class ExtendedLogger that extends the Logger class, and observe and explain the result. In the main() method (in a separate class), create an object of the Logger class and call the logMessage() method to print a sample message.
2. package Experiment6;
3. final class Logger {
4. public void logMessage(String message) {
5. System.out.println("[LOG] " + message);
6. }
7. }
8. // This class will cause compilation error as Logger is final
9. // class ExtendedLogger extends Logger {
10. //     public void logMessage(String message) {
11. //         System.out.println("[EXTENDED LOG] " + message);
12. //     }
13. // }
14. public class LoggerDemo {
15. public static void main(String[] args) {
16. System.out.println("Demonstrating final class:\n");
18. Logger logger = new Logger();
19. logger.logMessage("This is a test message");
21. System.out.println("\nNote: Attempting to extend Logger class would result in compilation error");
22. System.out.println("because Logger is declared as final.");
23. }
24. }

**Output**

****

## EXPERIMENT – 7

## TITLE Abstract Classes and Interfaces

1. Write a Java program to create an abstract class Shape with an abstract method calculateArea(). Derive two classes Rectangle and Circle from Shape and override the calculateArea() method to calculate and print the area of a rectangle and a circle, respectively. Use the main() method to create objects of Rectangle and Circle and test their calculateArea() methods.
2. package Experiment7;
3. /\*\*
4. \* Question: Abstract Shape Class Implementation
5. \*
6. \* Create a Java program that implements an abstract Shape class and its concrete
7. \* implementations to demonstrate abstraction and inheritance in geometric calculations.
8. \*
9. \* Requirements:
10. \* 1. Create abstract Shape class with:
11. \*    - Abstract method for area calculation
12. \* 2. Implement concrete classes:
13. \*    - Rectangle with length and width
14. \*    - Circle with radius
15. \* 3. Demonstrate:
16. \*    - Abstract method implementation
17. \*    - Area calculations
18. \*    - Proper constant usage (PI)
19. \*
20. \* Learning Objectives:
21. \* - Understanding abstract classes
22. \* - Implementing abstract methods
23. \* - Working with geometric calculations
24. \* - Class hierarchy design
25. \* - Constant declaration and usage
26. \*/
27. abstract class Shape {
28. abstract double calculateArea();
29. }
30. class Rectangle extends Shape {
31. private double length;
32. private double width;
34. public Rectangle(double length, double width) {
35. this.length = length;
36. this.width = width;
37. }
39. @Override
40. double calculateArea() {
41. return length \* width;
42. }
43. }
44. class Circle extends Shape {
45. private double radius;
46. private static final double PI = 3.14159;
48. public Circle(double radius) {
49. this.radius = radius;
50. }
52. @Override
53. double calculateArea() {
54. return PI \* radius \* radius;
55. }
56. }
57. public class ShapeDemo {
58. public static void main(String[] args) {
59. Rectangle rectangle = new Rectangle(5, 3);
60. Circle circle = new Circle(4);
62. System.out.println("Area of Rectangle: " + rectangle.calculateArea());
63. System.out.println("Area of Circle: " + circle.calculateArea());
64. }
65. }
66. Write a Java program to create an abstract class Employee with abstract methods calculateSalary() and displayDetails(). Derive two classes Manager and Developer from Employee and implement the methods to calculate the salary (e.g., based on fixed salary or hourly wage) and display employee details (e.g., name, role, salary). In the main() method, create objects of Manager and Developer and test their functionality.

package Experiment7;

/\*

Problem Statement:

Implement an Employee Management System using Abstract Classes that demonstrates the following requirements:

1. Create an abstract Employee class with basic attributes (name, role, base salary)

2. Implement two types of employees: Manager (fixed salary + bonus) and Developer (hourly rate)

3. Use abstract methods for salary calculation and detail display

4. Demonstrate polymorphic behavior through a test program

Key Concepts Demonstrated:

- Abstract Classes and Methods

- Inheritance and Method Overriding

- Encapsulation through protected/private fields

- Different salary calculation strategies

- Polymorphic behavior

- Constructor chaining using super()

\*/

abstract class Employee {

    protected String name;

    protected String role;

    protected double baseSalary;

    public Employee(String name, String role, double baseSalary) {

        this.name = name;

        this.role = role;

        this.baseSalary = baseSalary;

    }

    abstract double calculateSalary();

    abstract void displayDetails();

}

class Manager extends Employee {

    private double bonus;

    public Manager(String name, double baseSalary, double bonus) {

        super(name, "Manager", baseSalary);

        this.bonus = bonus;

    }

    @Override

    double calculateSalary() {

        return baseSalary + bonus;

    }

    @Override

    void displayDetails() {

        System.out.println("\nManager Details:");

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

        System.out.println("Role: " + role);

        System.out.println("Base Salary: $" + baseSalary);

        System.out.println("Bonus: $" + bonus);

        System.out.println("Total Salary: $" + calculateSalary());

    }

}

class Developer extends Employee {

    private int hoursWorked;

    private double hourlyRate;

    public Developer(String name, int hoursWorked, double hourlyRate) {

        super(name, "Developer", 0); // Base salary will be calculated from hours

        this.hoursWorked = hoursWorked;

        this.hourlyRate = hourlyRate;

    }

    @Override

    double calculateSalary() {

        return hoursWorked \* hourlyRate;

    }

    @Override

    void displayDetails() {

        System.out.println("\nDeveloper Details:");

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

        System.out.println("Role: " + role);

        System.out.println("Hours Worked: " + hoursWorked);

        System.out.println("Hourly Rate: $" + hourlyRate);

        System.out.println("Total Salary: $" + calculateSalary());

    }

}

public class EmployeeDemo {

    public static void main(String[] args) {

        Manager manager = new Manager("John Doe", 5000, 1000);

        Developer developer = new Developer("Jane Smith", 160, 25);

        manager.displayDetails();

        developer.displayDetails();

    }

}

1. Write a Java program to create an interface Bank with methods deposit(double amount) and withdraw(double amount). Implement this interface in a class Account that overrides these methods to perform deposit and withdrawal operations on a balance variable. Create another class BankDemo with a main() method to test the functionality by depositing and withdrawing amounts and displaying the updated balance.
2. package Experiment7;
3. /\*\*
4. \* Question: Bank Account Interface Implementation
5. \*
6. \* Create a Java program that implements a banking system using interfaces.
7. \* The program should demonstrate interface implementation, encapsulation,
8. \* and basic banking operations.
9. \*
10. \* Requirements:
11. \* 1. Create a Bank interface with methods:
12. \*    - deposit(double amount)
13. \*    - withdraw(double amount)
14. \* 2. Implement Account class with:
15. \*    - Balance tracking
16. \*    - Transaction validation
17. \*    - Proper error handling
18. \* 3. Demonstrate usage with:
19. \*    - Account creation
20. \*    - Multiple transactions
21. \*    - Invalid operation handling
22. \*
23. \* Learning Objectives:
24. \* - Understanding interfaces
25. \* - Implementing interface methods
26. \* - Encapsulation principles
27. \* - Transaction validation
28. \* - Error handling
29. \*/
30. interface Bank {
31. void deposit(double amount);
32. void withdraw(double amount);
33. }
34. class Account implements Bank {
35. private double balance;
37. public Account(double initialBalance) {
38. this.balance = initialBalance;
39. }
41. @Override
42. public void deposit(double amount) {
43. if (amount > 0) {
44. balance += amount;
45. System.out.println("Deposited: $" + amount);
46. System.out.println("Current Balance: $" + balance);
47. } else {
48. System.out.println("Invalid deposit amount");
49. }
50. }
52. @Override
53. public void withdraw(double amount) {
54. if (amount > 0 && amount <= balance) {
55. balance -= amount;
56. System.out.println("Withdrawn: $" + amount);
57. System.out.println("Current Balance: $" + balance);
58. } else {
59. System.out.println("Invalid withdrawal amount or insufficient balance");
60. }
61. }
63. public double getBalance() {
64. return balance;
65. }
66. }
67. public class BankDemo {
68. public static void main(String[] args) {
69. Account account = new Account(1000);
71. System.out.println("Initial Balance: $" + account.getBalance());
73. account.deposit(500);
74. account.withdraw(200);
75. account.withdraw(1500); // This should fail
76. account.deposit(-100); // This should fail
77. }
78. }
79. Write a Java program to create an interface Playable with methods play(), pause(), and stop(). Implement this interface in a class MusicPlayer that overrides these methods to print appropriate messages (e.g., "Music is playing," "Music is paused," "Music is stopped"). Create another class TestPlayer with a main() method to test the functionality by calling the play(), pause(), and stop() methods.

package Experiment7;

/\*

Problem Statement:

Implement a Media Player interface system that demonstrates the following requirements:

1. Create a Playable interface with basic media control methods

2. Implement a MusicPlayer class with state management

3. Include methods for play, pause, and stop operations

4. Demonstrate state transitions and current state tracking

Key Concepts Demonstrated:

- Interface design and implementation

- State management in objects

- Method overriding

- Basic media control operations

- Object state tracking and reporting

\*/

interface Playable {

    void play();

    void pause();

    void stop();

}

class MusicPlayer implements Playable {

    private String currentState = "Stopped";

    @Override

    public void play() {

        currentState = "Playing";

        System.out.println("Music is playing");

    }

    @Override

    public void pause() {

        currentState = "Paused";

        System.out.println("Music is paused");

    }

    @Override

    public void stop() {

        currentState = "Stopped";

        System.out.println("Music is stopped");

    }

    public String getCurrentState() {

        return currentState;

    }

}

public class TestPlayer {

    public static void main(String[] args) {

        MusicPlayer player = new MusicPlayer();

        System.out.println("Testing Music Player Controls:\n");

        player.play();

        System.out.println("Current State: " + player.getCurrentState());

        player.pause();

        System.out.println("Current State: " + player.getCurrentState());

        player.play();

        System.out.println("Current State: " + player.getCurrentState());

        player.stop();

        System.out.println("Current State: " + player.getCurrentState());

    }

}

**Additional Question:**

* 1. Write a Java program that defines an interface StackInterface with three methods: push(), pop(), and display(). Implement this interface in a class called StackClass. The StackClass should also contain the main method, where a switch-case structure is used to allow users to select and perform stack operations.

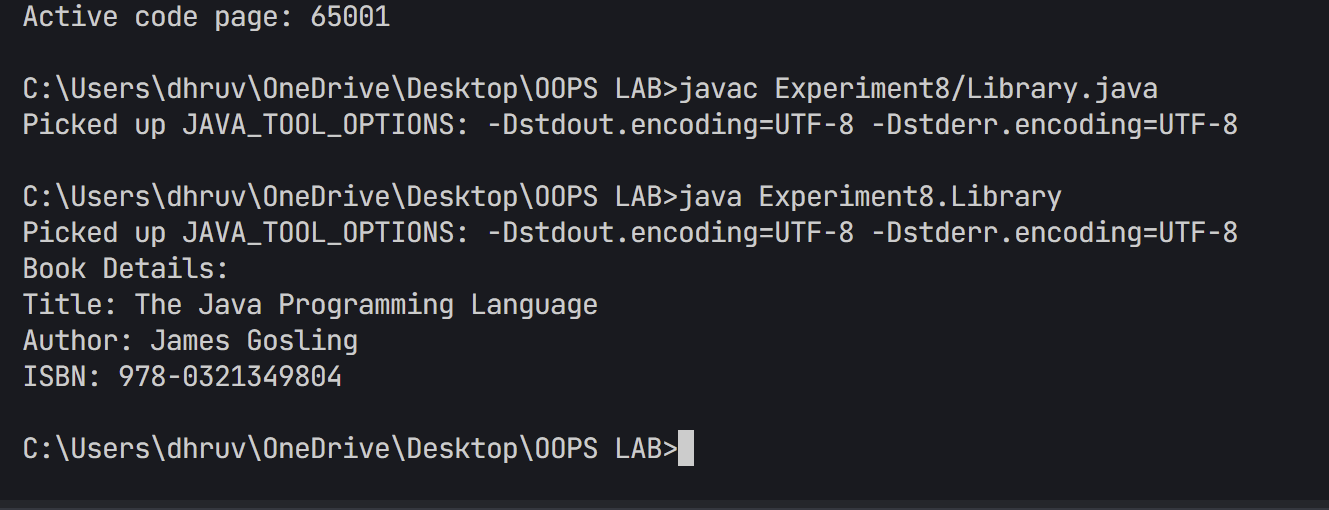
1. package Experiment7;
2. import java.util.Scanner;
3. /\*
4. Problem Statement:
5. Implement a Stack data structure using interface that demonstrates the following requirements:
6. 1. Create a StackInterface with basic stack operations (push, pop, display)
7. 2. Implement a fixed-size stack with array implementation
8. 3. Include proper boundary checking (overflow/underflow)
9. 4. Create an interactive menu-driven program to test stack operations
10. Key Concepts Demonstrated:
11. - Interface implementation
12. - Array-based data structure
13. - Stack operations (LIFO principle)
14. - Input/Output handling
15. - Error handling for edge cases
16. - Interactive menu implementation
17. \*/
18. interface StackInterface {
19. void push();
20. void pop();
21. void display();
22. }
23. public class StackClass implements StackInterface {
24. private int[] stack;
25. private int top;
26. private static final int MAX\_SIZE = 5;
27. private Scanner scanner;
29. public StackClass() {
30. stack = new int[MAX\_SIZE];
31. top = -1;
32. scanner = new Scanner(System.in);
33. }
35. @Override
36. public void push() {
37. if (top >= MAX\_SIZE - 1) {
38. System.out.println("\nStack Overflow! Cannot push more elements.");
39. return;
40. }
42. System.out.print("Enter element to push: ");
43. int element = scanner.nextInt();
44. stack[++top] = element;
45. System.out.println("Pushed " + element + " to stack");
46. }
48. @Override
49. public void pop() {
50. if (top < 0) {
51. System.out.println("\nStack Underflow! Cannot pop from empty stack.");
52. return;
53. }
55. int element = stack[top--];
56. System.out.println("Popped " + element + " from stack");
57. }
59. @Override
60. public void display() {
61. if (top < 0) {
62. System.out.println("\nStack is empty!");
63. return;
64. }
66. System.out.println("\nStack elements:");
67. for (int i = top; i >= 0; i--) {
68. System.out.println(stack[i]);
69. }
70. }
72. public static void main(String[] args) {
73. StackClass stack = new StackClass();
74. Scanner scanner = new Scanner(System.in);
75. int choice;
77. do {
78. System.out.println("\nStack Operations:");
79. System.out.println("1. Push");
80. System.out.println("2. Pop");
81. System.out.println("3. Display");
82. System.out.println("4. Exit");
83. System.out.print("Enter your choice: ");
85. choice = scanner.nextInt();
87. switch (choice) {
88. case 1:
89. stack.push();
90. break;
91. case 2:
92. stack.pop();
93. break;
94. case 3:
95. stack.display();
96. break;
97. case 4:
98. System.out.println("Exiting program...");
99. break;
100. default:
101. System.out.println("Invalid choice! Please try again.");
102. }
103. } while (choice != 4);
105. scanner.close();
106. stack.scanner.close();
107. }
108. }

## Experiment 8

## TITLE Inner/Nested Classes, Exception Handling & File Handling

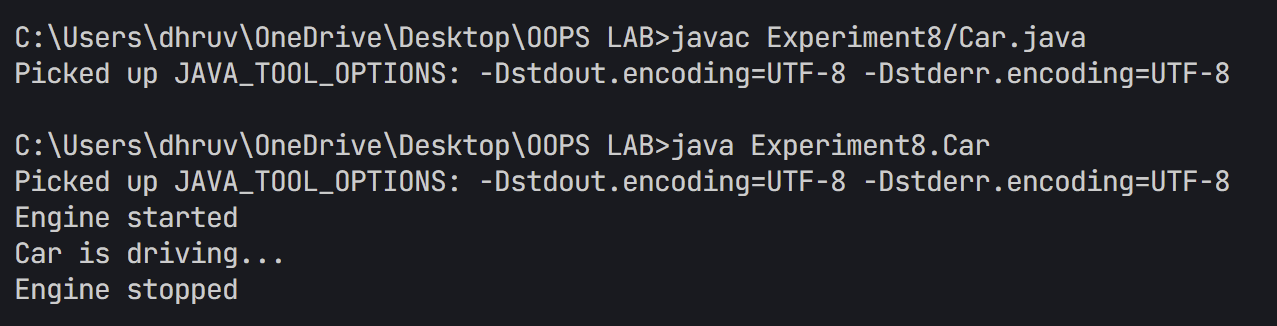
1. Create an outer class Library with a static nested class Book. The Book class should have attributes like title, author, and ISBN, and a method displayDetails() to print these details. In the main method, create an instance of the Book class and call displayDetails() to show the book information.
2. package Experiment8;
3. /\*\*
4. \* Question: Library Management using Static Nested Classes
5. \*
6. \* Create a Java program that demonstrates the use of static nested classes
7. \* using a Library-Book relationship. The program should:
8. \* 1. Define a Library class with a static nested Book class
9. \* 2. Implement the Book class with:
10. \*    - Private fields for book details
11. \*    - Constructor for initialization
12. \*    - Method to display book information
13. \* 3. Show how to:
14. \*    - Create instances of static nested class
15. \*    - Access and display book details
16. \*    - Maintain proper encapsulation
17. \*
18. \* This program demonstrates:
19. \* - Static nested class implementation
20. \* - Object-oriented principles
21. \* - Data encapsulation
22. \* - Constructor usage
23. \* - Method implementation
24. \*/
25. public class Library {
26. // Static nested class Book
27. public static class Book {
28. private String title;
29. private String author;
30. private String isbn;
32. public Book(String title, String author, String isbn) {
33. this.title = title;
34. this.author = author;
35. this.isbn = isbn;
36. }
38. public void displayDetails() {
39. System.out.println("Book Details:");
40. System.out.println("Title: " + title);
41. System.out.println("Author: " + author);
42. System.out.println("ISBN: " + isbn);
43. }
44. }
46. public static void main(String[] args) {
47. // Creating an instance of the Book class
48. Book book = new Book("The Java Programming Language", "James Gosling", "978-0321349804");
50. // Displaying book details
51. book.displayDetails();
52. }
53. }

Output



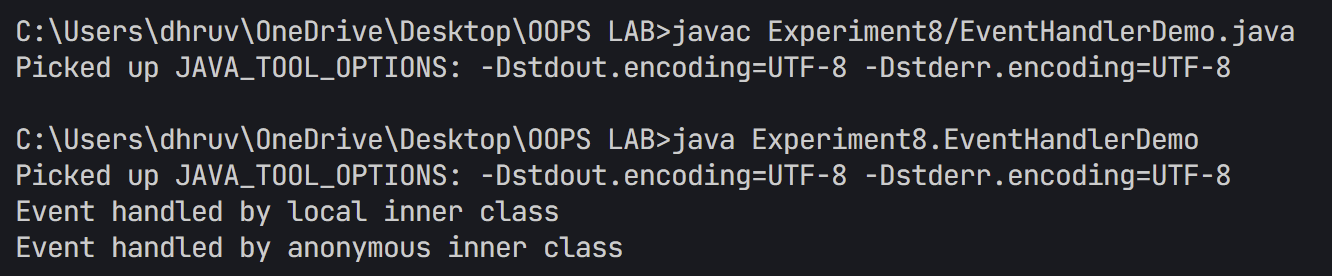
1. Create an outer class Car with an inner class Engine. The Engine class should have a method start() that prints "Engine started" and a method stop() that prints "Engine stopped". The Car class should have a method drive() that creates an instance of the Engine class and calls its start() and stop() methods.
2. package Experiment8;
3. /\*\*
4. \* Question: Inner Class Implementation with Car Engine
5. \*
6. \* Create a Java program that demonstrates the use of inner classes using a Car-Engine relationship.
7. \* The program should:
8. \* 1. Define a Car class with an inner Engine class
9. \* 2. Implement methods in the Engine class for basic operations:
10. \*    - Starting the engine
11. \*    - Stopping the engine
12. \* 3. Show how to:
13. \*    - Create an instance of the inner class
14. \*    - Access inner class methods from outer class
15. \*    - Maintain encapsulation of engine operations
16. \*
17. \* This program demonstrates:
18. \* - Inner class implementation
19. \* - Object-oriented encapsulation
20. \* - Class relationship modeling
21. \* - Method invocation between classes
22. \*/
23. public class Car {
24. // Inner class Engine
25. private class Engine {
26. public void start() {
27. System.out.println("Engine started");
28. }
30. public void stop() {
31. System.out.println("Engine stopped");
32. }
33. }
35. public void drive() {
36. // Creating an instance of the Engine class
37. Engine engine = new Engine();
39. // Calling Engine methods
40. engine.start();
41. System.out.println("Car is driving...");
42. engine.stop();
43. }
45. public static void main(String[] args) {
46. // Creating an instance of Car
47. Car car = new Car();
49. // Calling drive method
50. car.drive();
51. }
52. }

Output



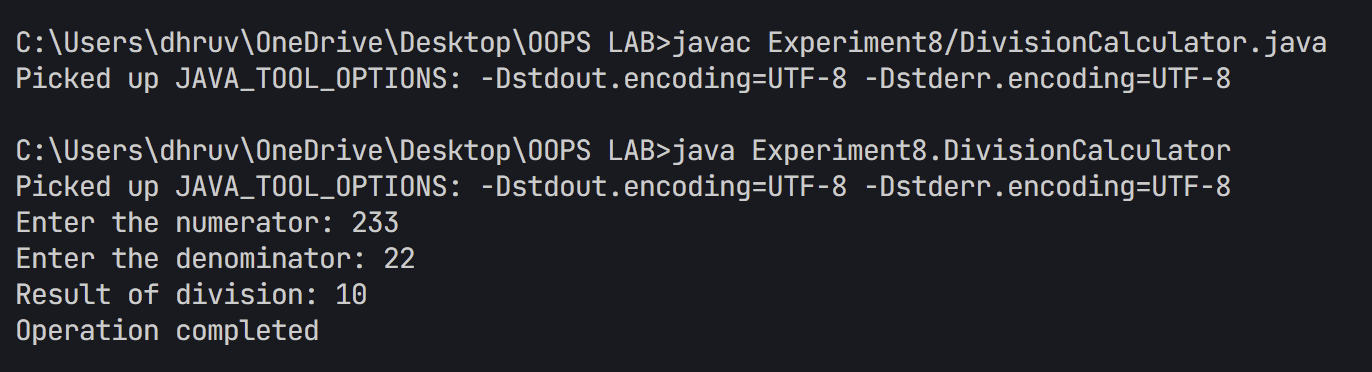
1. Create an interface EventHandler with a method handleEvent(). In the main method, demonstrate the use of:
   1. A local inner class inside a method registerEvent() that implements EventHandler and prints "Event handled by local inner class".
   2. An anonymous inner class that implements EventHandler and prints "Event handled by anonymous inner class".
2. package Experiment8;
3. /\*\*
4. \* Question: Event Handler Implementation using Inner Classes
5. \*
6. \* Create a Java program that demonstrates the use of local and anonymous inner classes
7. \* for event handling. The program should:
8. \* 1. Define an EventHandler interface
9. \* 2. Implement the interface using:
10. \*    - A local inner class
11. \*    - An anonymous inner class
12. \* 3. Show how to:
13. \*    - Create and use local inner classes within methods
14. \*    - Implement interfaces using anonymous classes
15. \*    - Handle events using different inner class types
16. \*
17. \* This program demonstrates:
18. \* - Interface implementation
19. \* - Local inner classes
20. \* - Anonymous inner classes
21. \* - Event handling patterns
22. \* - Method overriding
23. \*/
24. public class EventHandlerDemo {
25. // Interface definition
26. interface EventHandler {
27. void handleEvent();
28. }
30. public void registerEvent() {
31. // Local inner class implementation
32. class LocalEventHandler implements EventHandler {
33. @Override
34. public void handleEvent() {
35. System.out.println("Event handled by local inner class");
36. }
37. }
39. // Creating and using local inner class
40. EventHandler localHandler = new LocalEventHandler();
41. localHandler.handleEvent();
42. }
44. public static void main(String[] args) {
45. EventHandlerDemo demo = new EventHandlerDemo();
47. // Using local inner class
48. demo.registerEvent();
50. // Using anonymous inner class
51. EventHandler anonymousHandler = new EventHandler() {
52. @Override
53. public void handleEvent() {
54. System.out.println("Event handled by anonymous inner class");
55. }
56. };
58. anonymousHandler.handleEvent();
59. }
60. }

Output



1. Write a Java program that takes two integers as input from the user and performs division. Handle the ArithmeticException that occurs if the denominator is zero. Use a try-catch block to catch the exception and display an appropriate error message. Additionally, use a finally block to print "Operation completed" regardless of whether an exception occurs or not.
2. package Experiment8;
3. import java.util.Scanner;
4. /\*\*
5. \* Question: Implement a Division Calculator with Exception Handling
6. \*
7. \* Create a Java program that performs division of two numbers with proper exception handling.
8. \* The program should:
9. \* 1. Accept two numbers from the user (numerator and denominator)
10. \* 2. Handle potential exceptions:
11. \*    - ArithmeticException for division by zero
12. \*    - Other exceptions for invalid input
13. \* 3. Display the result of division if successful
14. \* 4. Show appropriate error messages for exceptions
15. \* 5. Demonstrate proper resource management using finally block
16. \*
17. \* This program demonstrates:
18. \* - Exception handling using try-catch blocks
19. \* - Multiple catch blocks for different exceptions
20. \* - Using Scanner class for user input
21. \* - Proper resource cleanup in finally block
22. \* - Basic arithmetic operations and error checking
23. \*/
24. public class DivisionCalculator {
25. public static void main(String[] args) {
26. Scanner scanner = new Scanner(System.in);
28. try {
29. // Get input from user
30. System.out.print("Enter the numerator: ");
31. int numerator = scanner.nextInt();
33. System.out.print("Enter the denominator: ");
34. int denominator = scanner.nextInt();
36. // Perform division
37. int result = numerator / denominator;
38. System.out.println("Result of division: " + result);
40. } catch (ArithmeticException e) {
41. System.out.println("Error: Cannot divide by zero!");
43. } catch (Exception e) {
44. System.out.println("Error: Invalid input!");
46. } finally {
47. System.out.println("Operation completed");
48. scanner.close();
49. }
50. }
51. }

Output



1. Write a Java program that creates an array of 5 integers and asks the user to enter an index to access the array element. Handle the ArrayIndexOutOfBoundsException if the user enters an invalid index. Use a try-catch block to catch the exception and display an appropriate error message. Use the finally block to print "Array access attempted."

package Experiment8;

/\*\*

 \* Question: Array Access with Exception Handling

 \*

 \* Create a Java program that demonstrates array bounds checking and exception handling.

 \* The program should:

 \* 1. Create and initialize an array with values

 \* 2. Allow user to access array elements by index

 \* 3. Handle potential exceptions:

 \*    - ArrayIndexOutOfBoundsException for invalid indices

 \*    - Other exceptions for invalid input

 \* 4. Display appropriate error messages

 \* 5. Use proper resource management

 \*

 \* This program demonstrates:

 \* - Array manipulation and access

 \* - Exception handling with try-catch blocks

 \* - User input validation

 \* - Scanner class usage

 \* - Resource cleanup in finally block

 \*/

import java.util.Scanner;

public class ArrayAccessDemo {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int[] numbers = new int[5];

        // Initialize array with some values

        for (int i = 0; i < numbers.length; i++) {

            numbers[i] = (i + 1) \* 10;

        }

        try {

            // Display array contents

            System.out.println("Array contents:");

            for (int i = 0; i < numbers.length; i++) {

                System.out.println("Index " + i + ": " + numbers[i]);

            }

            // Get index from user

            System.out.print("\nEnter an index (0-4) to access the array: ");

            int index = scanner.nextInt();

            // Access and display element

            System.out.println("Value at index " + index + ": " + numbers[index]);

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("Error: Invalid array index! Please enter an index between 0 and 4.");

        } catch (Exception e) {

            System.out.println("Error: Invalid input!");

        } finally {

            System.out.println("Array access attempted.");

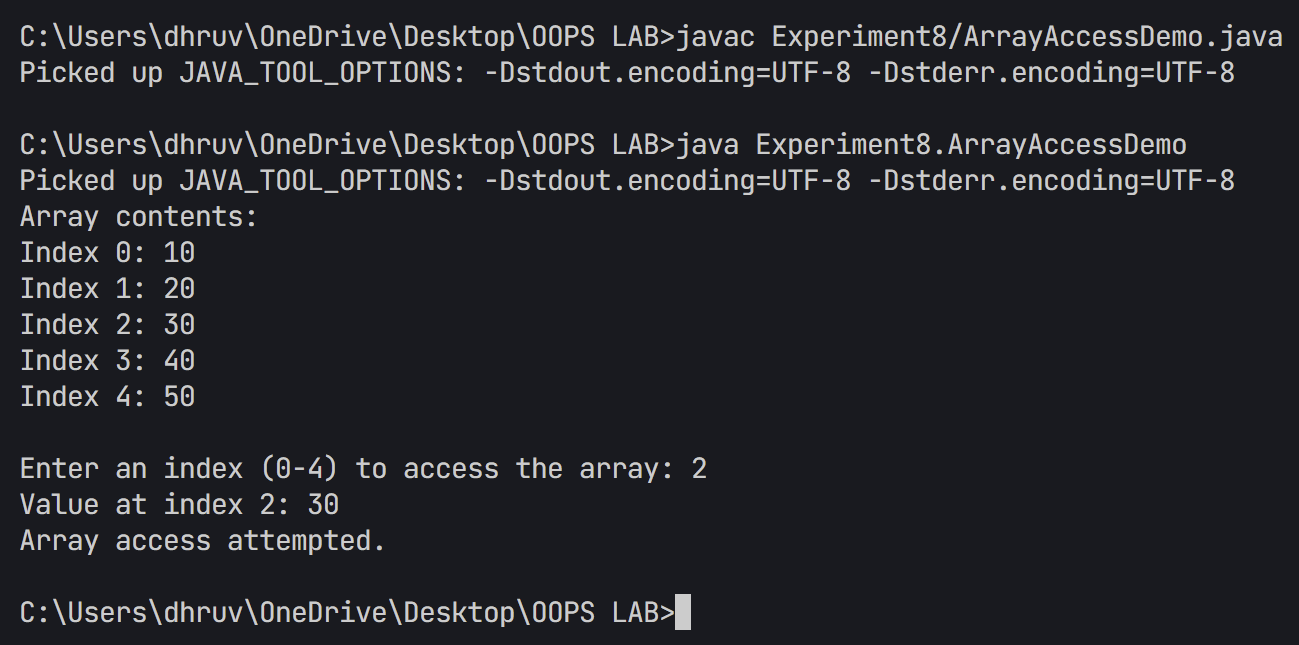
            scanner.close();

        }

    }

}

Output



1. Write a Java program that reads a file name from the user and attempts to open and read the file. Define a method readFile() that throws a FileNotFoundException using the throws keyword. In the main method, call this method and handle the exception using a try-catch block. Display an appropriate message if the file is not found. Use a finally block to ensure a message like "File operation attempted" is printed.
2. package Experiment8;
3. /\*\*
4. \* Question: File Reading with Exception Handling
5. \*
6. \* Create a Java program that demonstrates file reading operations with proper exception handling.
7. \* The program should:
8. \* 1. Accept a filename from the user
9. \* 2. Implement a method that throws FileNotFoundException
10. \* 3. Handle potential exceptions:
11. \*    - FileNotFoundException for missing files
12. \*    - Other I/O related exceptions
13. \* 4. Use proper resource management
14. \* 5. Display file contents or appropriate error messages
15. \*
16. \* This program demonstrates:
17. \* - File handling in Java
18. \* - Exception handling with throws clause
19. \* - Try-catch-finally blocks
20. \* - Scanner usage for file reading
21. \* - Resource cleanup
22. \*/
23. import java.io.File;
24. import java.io.FileNotFoundException;
25. import java.util.Scanner;
26. public class FileReadDemo {
27. // Method that throws FileNotFoundException
28. public static void readFile(String fileName) throws FileNotFoundException {
29. File file = new File(fileName);
30. Scanner fileReader = new Scanner(file);
32. try {
33. System.out.println("File contents:");
34. while (fileReader.hasNextLine()) {
35. String line = fileReader.nextLine();
36. System.out.println(line);
37. }
38. } finally {
39. fileReader.close();
40. }
41. }
43. public static void main(String[] args) {
44. Scanner scanner = new Scanner(System.in);
46. try {
47. System.out.print("Enter the file name to read: ");
48. String fileName = scanner.nextLine();
50. readFile(fileName);
52. } catch (FileNotFoundException e) {
53. System.out.println("Error: File not found! Please check the file name and path.");
55. } finally {
56. System.out.println("File operation attempted.");
57. scanner.close();
58. }
59. }
60. }

C:\Users\dhruv\OneDrive\Desktop\OOPS LAB>javac Experiment8/FileReadDemo.java

Picked up JAVA\_TOOL\_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8

C:\Users\dhruv\OneDrive\Desktop\OOPS LAB>java Experiment8.FileReadDemo

Picked up JAVA\_TOOL\_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8

Enter the file name to read: readme.md

File contents:

# Object-Oriented Programming Lab

## Student Information

- \*\*Name:\*\* Dhruv Shah

- \*\*Batch:\*\* B-11 AIML

- \*\*SAP ID:\*\* 500118979

- \*\*University:\*\* UPES Dehradun

## Course Overview

This repository contains comprehensive implementations of Object-Oriented Programming concepts in Java through various experiments. Each experiment is carefully designed to provide hands-on experience with different aspects of OOP and their practical applications in software development.

## Prerequisites

- Java Development Kit (JDK) 8 or higher

- Any Java IDE (Eclipse, IntelliJ IDEA, or NetBeans recommended)

- Basic understanding of programming concepts

- Git for version control

## Development Environment Setup

1. \*\*Install JDK\*\*

- Download and install the latest JDK from Oracle's website

- Set up JAVA\_HOME environment variable

- Add Java to system PATH

2. \*\*IDE Setup\*\*

- Install your preferred Java IDE

- Configure JDK in IDE settings

- Import the project from version control

## Experiments

### Experiment 1: Introduction to Java

\*\*Programs Implemented:\*\*

- `HelloWorld.java`: First Java program demonstrating basic syntax

- `DisplayAddress.java`: String manipulation and output formatting

- `GradeSheet.java`: Basic input/output operations with decision making

\*\*Key Concepts Covered:\*\*

- Java program structure and compilation process

- Variables, data types, and type conversion

- Command-line input/output operations

- Documentation: Detailed explanation of JDK, JRE, and JVM in `JDK\_JRE\_JVM.md`

### Experiment 2: Control Structures

\*\*Programs Implemented:\*\*

- `CommandLineCalculator.java`: Basic arithmetic operations

- `DayOfWeek.java`: Switch case implementation

- `NumberClassifier.java`: Nested if-else statements

- `StudentGrade.java`: Grade calculation using control structures

- `TriangleArea.java`: Mathematical calculations

\*\*Key Concepts:\*\*

- Decision-making statements (if-else, switch)

- Looping structures (for, while, do-while)

- Break and continue statements

- Command-line argument processing

### Experiment 3: Arrays and Basic Programs

\*\*Programs Implemented:\*\*

- `ArrayCopy.java`: Array manipulation techniques

- `Fibonacci.java`: Series generation using arrays

- `PrimeNumbers.java`: Mathematical computations

- `PatternPrinting.java`: Nested loops for pattern generation

- `SecondLargest.java`: Array sorting and searching

\*\*Advanced Concepts:\*\*

- Multi-dimensional arrays

- Array sorting algorithms

- Pattern recognition and implementation

- Mathematical series implementations

### Experiment 4: Classes and Objects

\*\*Programs Implemented:\*\*

- `BankAccount.java`: Banking system implementation

- `Calculator.java`: Method overloading demonstration

- `Student.java`: Student management system

- `Employee.java`: Employee data management

\*\*OOP Concepts Covered:\*\*

- Class and object creation

- Constructor overloading

- Instance and static members

- Access modifiers and encapsulation

### Experiment 5: Inheritance

\*\*Programs Implemented:\*\*

- `EmployeeManager.java`: Multi-level inheritance

- `VehicleHierarchy.java`: Hierarchical inheritance

- `WorkerHierarchy.java`: Single inheritance

- `UniversitySystem.java`: Complex inheritance structure

\*\*Advanced OOP Concepts:\*\*

- Types of inheritance (single, multiple, multilevel)

- Method overriding and dynamic method dispatch

- Super keyword and constructor chaining

- Abstract classes and methods

### Experiment 6: Package and Access Control

\*\*Programs Implemented:\*\*

- Custom package creation with `Balance` and `Account` classes

- Access modifier demonstration with packages 'p' and 'Q'

- `FinalDemo.java`: Final keyword implementation

- `LoggerDemo.java`: Logging implementation

\*\*Key Features:\*\*

- Package creation and importing

- Access control implementation

- Final classes, methods, and variables

- Java logging framework usage

### Experiment 7: Abstract Classes and Interfaces

\*\*Programs Implemented:\*\*

- `BankDemo.java`: Banking interface implementation

- `ShapeDemo.java`: Abstract shape hierarchy

- `EmployeeDemo.java`: Employee management system

- `TestPlayer.java`: Sports player interface

\*\*Advanced Concepts:\*\*

- Abstract class implementation

- Interface design and implementation

- Multiple interface inheritance

- Polymorphic behavior

### Experiment 8: Exception Handling

\*\*Programs Implemented:\*\*

- `DivisionCalculator.java`: Arithmetic exception handling

- `FileReadDemo.java`: File handling exceptions

- `ArrayAccessDemo.java`: Array bounds checking

- `StudentFileReader.java`: File I/O with exception handling

\*\*Exception Handling Concepts:\*\*

- Try-catch blocks

- Multiple catch blocks

- Throw and throws keywords

- Custom exception creation

### Experiment 9: GUI Programming

\*\*Programs Implemented:\*\*

- `Calculator.java`: GUI calculator application

- `LoginForm.java`: User authentication interface

- `ToDoList.java`: Task management application

- `EmployeeManagement.java`: Employee database GUI

\*\*GUI Concepts:\*\*

- AWT and Swing components

- Event handling mechanisms

- Layout managers

- Dialog boxes and forms

### Experiment 10: Collections

\*\*Programs Implemented:\*\*

- `ProductInventory.java`: Inventory management system

- `EmployeeArrayList.java`: Employee database using ArrayList

- `NameDeduplication.java`: HashSet implementation

- `PrimeNumberChecker.java`: Mathematical operations with collections

\*\*Collection Framework Concepts:\*\*

- ArrayList and LinkedList implementation

- Set interface and implementations

- Map interface and implementations

- Collection utility methods

## Key Features

- Comprehensive implementation of OOP concepts

- Well-documented code with detailed comments

- Practical real-world examples and applications

- Progressive learning approach from basic to advanced concepts

- Extensive error handling and input validation

- Modular and maintainable code structure

## Tools and Technologies

- Java Development Kit (JDK)

- Integrated Development Environment (IDE)

- Git for version control

- Command Line Interface

- GUI frameworks (AWT/Swing)

## How to Use

1. Clone the repository using Git:

```bash

git clone [repository-url]

```

2. Navigate to specific experiment folders

3. Compile Java files using javac:

```bash

javac FileName.java

```

4. Run compiled programs:

```bash

java FileName

```

5. Follow in-code comments for detailed implementation understanding

## Troubleshooting Common Issues

1. \*\*Compilation Errors\*\*

- Verify JDK installation and PATH settings

- Check for syntax errors in code

- Ensure all required files are present

2. \*\*Runtime Errors\*\*

- Verify input format and data types

- Check file paths for file operations

- Ensure sufficient system resources

## Learning Outcomes

- Solid understanding of Object-Oriented Programming principles

- Proficiency in Java programming language

- Strong problem-solving and analytical skills

- Experience with software design patterns

- Practical knowledge of GUI programming

- Understanding of exception handling mechanisms

- Proficiency in using collection frameworks

- Experience with file handling and I/O operations

## Contribution Guidelines

1. Fork the repository

2. Create a new branch for your feature

3. Commit your changes with descriptive messages

4. Push to your fork and submit a pull request

5. Ensure your code follows the established conventions

## License

This project is licensed under the MIT License - see the LICENSE file for details.

File operation attempted.

1. Write a Java program that takes user input for a student's name, roll number, and grade, and writes this information to a file named student.txt using **FileWriter**. Ensure the program appends the data to the file if it already exists. Handle any exceptions using try-catch and display an appropriate message if an error occurs.

*Sample File Content:*

Name: Aman, Roll Number: 120112, Grade: A

Name: Parul, Roll Number: 120131, Grade: B

package Experiment8;

/\*\*

 \* Question: Writing Student Records to File

 \*

 \* Create a Java program that demonstrates writing student records to a file

 \* with proper exception handling. The program should:

 \* 1. Accept student details from user input:

 \*    - Name

 \*    - Roll Number

 \*    - Grade

 \* 2. Write the information to a text file

 \* 3. Handle potential exceptions:

 \*    - IOException for writing errors

 \* 4. Implement proper resource management

 \* 5. Provide success/error feedback

 \*

 \* This program demonstrates:

 \* - File writing operations

 \* - Scanner for user input

 \* - Exception handling

 \* - Resource cleanup

 \* - Data formatting

 \*/

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

public class StudentFileWriter {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        try {

            // Open FileWriter in append mode

            FileWriter writer = new FileWriter("student.txt", true);

            // Get student details

            System.out.print("Enter student name: ");

            String name = scanner.nextLine();

            System.out.print("Enter roll number: ");

            String rollNumber = scanner.nextLine();

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

            String grade = scanner.nextLine();

            // Write to file

            String studentRecord = String.format("Name: %s, Roll Number: %s, Grade: %s\n",

                                               name, rollNumber, grade);

            writer.write(studentRecord);

            System.out.println("Student information successfully written to file.");

            writer.close();

        } catch (IOException e) {

            System.out.println("Error: Unable to write to file!");

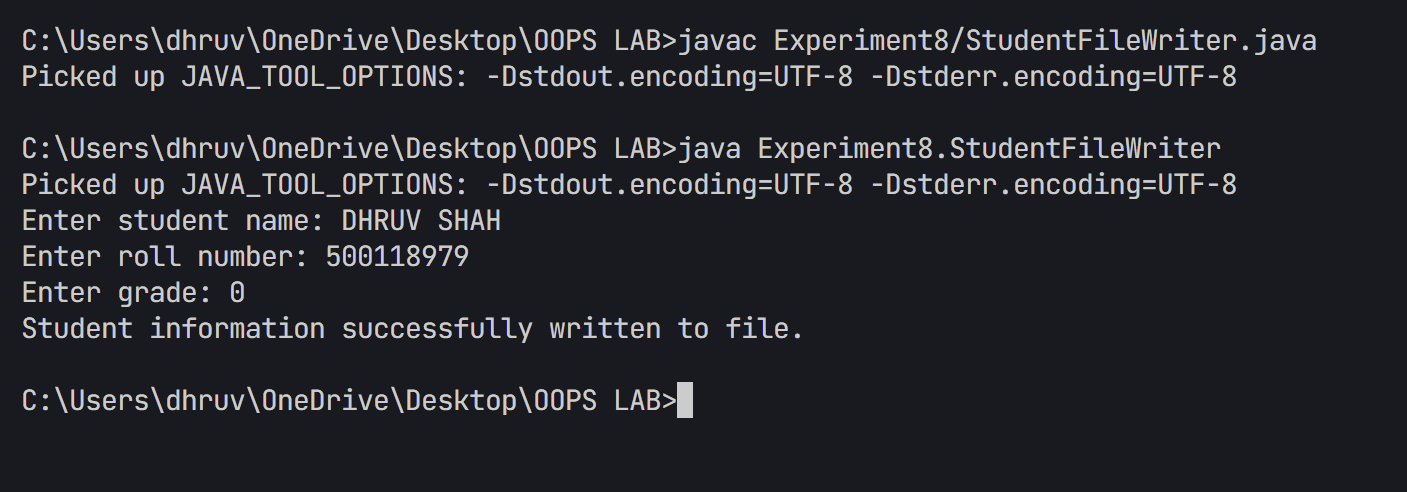
            e.printStackTrace();

        } finally {

            scanner.close();

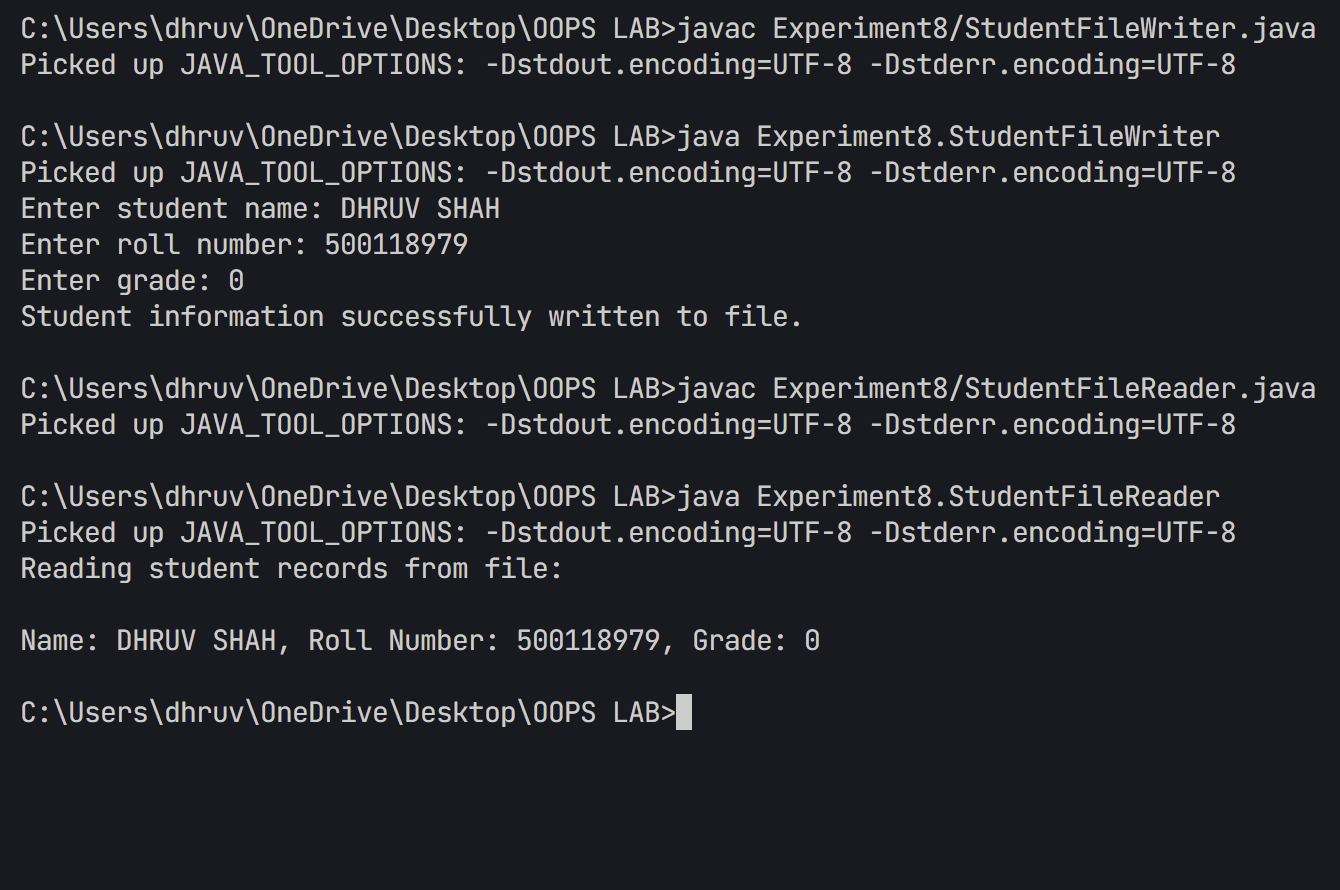
        }

    }

}

1. Write a Java program that reads the contents of a file named student.txt using **FileReader** and displays the data on the console. Handle FileNotFoundException if the file does not exist and display an appropriate error message. Use a try-catch block for exception handling.
2. package Experiment8;
3. /\*\*
4. \* Question: Reading Student Records from File
5. \*
6. \* Create a Java program that demonstrates reading student records from a file
7. \* with proper exception handling. The program should:
8. \* 1. Read student information from a text file
9. \* 2. Handle potential exceptions:
10. \*    - FileNotFoundException for missing files
11. \*    - IOException for reading errors
12. \* 3. Use BufferedReader for efficient file reading
13. \* 4. Display student records or error messages
14. \*
15. \* This program demonstrates:
16. \* - File I/O operations
17. \* - BufferedReader usage
18. \* - Exception handling hierarchy
19. \* - Error reporting
20. \* - Resource management
21. \*/
22. import java.io.FileReader;
23. import java.io.BufferedReader;
24. import java.io.FileNotFoundException;
25. import java.io.IOException;
26. public class StudentFileReader {
27. public static void main(String[] args) {
28. try {
29. // Create FileReader object
30. FileReader fr = new FileReader("student.txt");
31. BufferedReader reader = new BufferedReader(fr);
33. System.out.println("Reading student records from file:\n");
34. String line;
36. // Read file line by line
37. while ((line = reader.readLine()) != null) {
38. System.out.println(line);
39. }
41. reader.close();
43. } catch (FileNotFoundException e) {
44. System.out.println("Error: student.txt file not found!");
45. System.out.println("Please make sure the file exists in the current directory.");
47. } catch (IOException e) {
48. System.out.println("Error: An error occurred while reading the file.");
49. e.printStackTrace();
50. }
51. }
52. }

Output



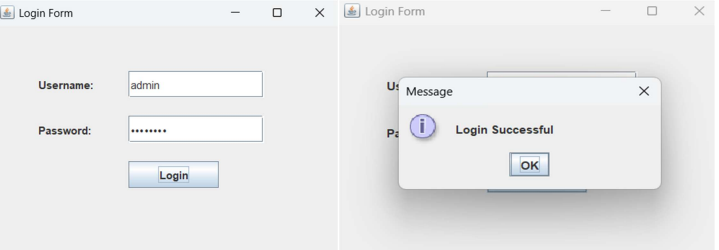
## Experiment 9

## TITLE Swing and JDBC

* 1. Write a Java Swing program to create a Login Form using JTextField, JPasswordField, JButton, and JLabel. When the login button is clicked, validate if the username is "admin" and the password is "password".

1. /\*\*
2. \* Question: User Authentication GUI with Java Swing
3. \*
4. \* Create a graphical login form application using Java Swing components.
5. \* The program should demonstrate GUI development, event handling, and
6. \* basic user authentication in a secure and user-friendly interface.
7. \*
8. \* Requirements:
9. \* 1. Implement a login form with:
10. \*    - Username input field
11. \*    - Password input field (masked)
12. \*    - Login button
13. \*    - Status message display
14. \* 2. Include the following features:
15. \*    - Input validation
16. \*    - Password masking
17. \*    - Visual feedback for login status
18. \*    - Proper layout and spacing
19. \* 3. Design considerations:
20. \*    - Professional appearance
21. \*    - Intuitive layout
22. \*    - Responsive interface
23. \*    - Secure password handling
24. \*
25. \* Learning Objectives:
26. \* - Java Swing GUI development
27. \* - Event handling in Java
28. \* - Layout management (GridBagLayout)
29. \* - User input validation
30. \* - Basic security practices
31. \*/
32. package Experiment9;
33. import javax.swing.\*;
34. import java.awt.\*;
35. import java.awt.event.\*;
36. public class LoginForm extends JFrame {
37. private JTextField usernameField;
38. private JPasswordField passwordField;
39. private JButton loginButton;
40. private JLabel messageLabel;
41. public LoginForm() {
42. setTitle("Login Form");
43. setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
44. setSize(300, 200);
45. setLocationRelativeTo(null);
46. setLayout(new GridBagLayout());
47. GridBagConstraints gbc = new GridBagConstraints();
48. gbc.insets = new Insets(5, 5, 5, 5);
49. // Username label and field
50. gbc.gridx = 0;
51. gbc.gridy = 0;
52. add(new JLabel("Username:"), gbc);
53. gbc.gridx = 1;
54. gbc.fill = GridBagConstraints.HORIZONTAL;
55. usernameField = new JTextField(15);
56. add(usernameField, gbc);
57. // Password label and field
58. gbc.gridx = 0;
59. gbc.gridy = 1;
60. add(new JLabel("Password:"), gbc);
61. gbc.gridx = 1;
62. passwordField = new JPasswordField(15);
63. add(passwordField, gbc);
64. // Login button
65. gbc.gridx = 0;
66. gbc.gridy = 2;
67. gbc.gridwidth = 2;
68. gbc.fill = GridBagConstraints.NONE;
69. loginButton = new JButton("Login");
70. add(loginButton, gbc);
71. // Message label
72. gbc.gridy = 3;
73. messageLabel = new JLabel(" ");
74. messageLabel.setForeground(Color.RED);
75. add(messageLabel, gbc);
76. // Add action listener to login button
77. loginButton.addActionListener(new ActionListener() {
78. public void actionPerformed(ActionEvent e) {
79. String username = usernameField.getText();
80. String password = new String(passwordField.getPassword());
81. if (username.equals("admin") && password.equals("password")) {
82. messageLabel.setForeground(Color.GREEN);
83. messageLabel.setText("Login successful!");
84. } else {
85. messageLabel.setForeground(Color.RED);
86. messageLabel.setText("Invalid username or password!");
87. }
88. }
89. });
90. }
91. public static void main(String[] args) {
92. SwingUtilities.invokeLater(new Runnable() {
93. public void run() {
94. new LoginForm().setVisible(true);
95. }
96. });
97. }
98. }

Output



* + - * 1. Design a simple calculator using Java Swing with buttons for digits (0-9), addition (+), subtraction (-), multiplication (\*), and division (/). Implement event handling for button clicks and display the result in a JTextField.

1. package Experiment9;
2. /\*\*
3. \* Question: GUI Calculator with Java Swing
4. \*
5. \* Create a graphical calculator application using Java Swing components.
6. \* The program should demonstrate GUI development, event handling, and
7. \* basic arithmetic operations in a user-friendly interface.
8. \*
9. \* Requirements:
10. \* 1. Implement a calculator with:
11. \*    - Numeric keypad (0-9)
12. \*    - Basic operations (+, -, \*, /)
13. \*    - Additional features (%, ±, C, ←)
14. \*    - Display field for results
15. \* 2. Include the following features:
16. \*    - Clear button functionality
17. \*    - Error handling for invalid operations
18. \*    - Proper decimal point handling
19. \*    - Memory of last operation
20. \* 3. Design considerations:
21. \*    - Professional appearance
22. \*    - Intuitive layout
23. \*    - Responsive buttons
24. \*
25. \* Learning Objectives:
26. \* - Java Swing GUI development
27. \* - Event handling in Java
28. \* - Layout management
29. \* - User input processing
30. \* - Calculator logic implementation
31. \*/
32. import javax.swing.\*;
33. import java.awt.\*;
34. import java.awt.event.\*;
35. public class Calculator extends JFrame {
36. private JTextField display;
37. private double result = 0;
38. private String lastCommand = "=";
39. private boolean start = true;
40. private boolean error = false;
41. public Calculator() {
42. setTitle("Calculator");
43. setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
44. setSize(300, 400);
45. setLocationRelativeTo(null);
46. // Create display field
47. display = new JTextField("0");
48. display.setEditable(false);
49. display.setHorizontalAlignment(JTextField.RIGHT);
50. display.setFont(new Font("Arial", Font.PLAIN, 20));
51. // Create button panel
52. JPanel buttonPanel = new JPanel();
53. buttonPanel.setLayout(new GridLayout(5, 4, 5, 5));
54. // Add buttons
55. String[] buttonLabels = {
56. "C", "←", "%", "/",
57. "7", "8", "9", "\*",
58. "4", "5", "6", "-",
59. "1", "2", "3", "+",
60. "±", "0", ".", "="
61. };
62. for (String label : buttonLabels) {
63. JButton button = new JButton(label);
64. buttonPanel.add(button);
65. if (label.matches("[0-9.]")) {
66. button.addActionListener(new NumberListener());
67. button.setBackground(new Color(240, 240, 240));
68. } else if (label.equals("=")) {
69. button.addActionListener(new OperatorListener());
70. button.setBackground(new Color(63, 81, 181));
71. button.setForeground(Color.WHITE);
72. } else if (label.equals("C")) {
73. button.addActionListener(e -> {
74. display.setText("0");
75. result = 0;
76. start = true;
77. lastCommand = "=";
78. error = false;
79. });
80. button.setBackground(new Color(244, 67, 54));
81. button.setForeground(Color.WHITE);
82. } else if (label.equals("←")) {
83. button.addActionListener(e -> {
84. if (!start && !error) {
85. String text = display.getText();
86. if (text.length() > 0) {
87. display.setText(text.substring(0, text.length() - 1));
88. if (display.getText().isEmpty()) {
89. display.setText("0");
90. start = true;
91. }
92. }
93. }
94. });
95. button.setBackground(new Color(255, 152, 0));
96. button.setForeground(Color.WHITE);
97. } else if (label.equals("±")) {
98. button.addActionListener(e -> {
99. if (!error) {
100. double x = Double.parseDouble(display.getText());
101. display.setText("" + (-x));
102. }
103. });
104. button.setBackground(new Color(156, 39, 176));
105. button.setForeground(Color.WHITE);
106. } else {
107. button.addActionListener(new OperatorListener());
108. button.setBackground(new Color(33, 150, 243));
109. button.setForeground(Color.WHITE);
110. }
111. }
112. // Layout components
113. setLayout(new BorderLayout(5, 5));
114. add(display, BorderLayout.NORTH);
115. add(buttonPanel, BorderLayout.CENTER);
116. // Add padding
117. ((JPanel)getContentPane()).setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
118. }
119. private class NumberListener implements ActionListener {
120. public void actionPerformed(ActionEvent event) {
121. String digit = event.getActionCommand();
122. if (start) {
123. display.setText(digit);
124. start = false;
125. } else {
126. display.setText(display.getText() + digit);
127. }
128. }
129. }
130. private class OperatorListener implements ActionListener {
131. public void actionPerformed(ActionEvent event) {
132. String command = event.getActionCommand();
134. if (start) {
135. if (command.equals("-")) {
136. display.setText(command);
137. start = false;
138. } else {
139. lastCommand = command;
140. }
141. } else {
142. calculate(Double.parseDouble(display.getText()));
143. lastCommand = command;
144. start = true;
145. }
146. }
147. }
148. public void calculate(double x) {
149. try {
150. if (lastCommand.equals("+")) result += x;
151. else if (lastCommand.equals("-")) result -= x;
152. else if (lastCommand.equals("\*")) result \*= x;
153. else if (lastCommand.equals("/")) {
154. if (x == 0) throw new ArithmeticException("Cannot divide by zero");
155. result /= x;
156. }
157. else if (lastCommand.equals("%")) {
158. if (x == 0) throw new ArithmeticException("Cannot calculate modulo with zero");
159. result %= x;
160. }
161. else if (lastCommand.equals("=")) result = x;
163. if (Double.isInfinite(result) || Double.isNaN(result)) {
164. throw new ArithmeticException("Result is undefined");
165. }
167. display.setText(String.format("%.8g", result));
168. error = false;
169. } catch (ArithmeticException e) {
170. display.setText("Error: " + e.getMessage());
171. error = true;
172. }
173. }
174. public static void main(String[] args) {
175. SwingUtilities.invokeLater(new Runnable() {
176. public void run() {
177. new Calculator().setVisible(true);
178. }
179. });
180. }
181. }

Output



* + - * 1. Write a Java Swing program to implement a To-Do List using JList. Provide:

1. A JTextField to enter tasks
2. An Add button to add tasks to the list
3. A Remove button to delete selected tasks

/\*\*

 \* Question: Task Management GUI with Java Swing

 \*

 \* Create a graphical to-do list application using Java Swing components.

 \* The program should demonstrate GUI development, event handling, and

 \* basic task management functionality in an intuitive interface.

 \*

 \* Requirements:

 \* 1. Implement a to-do list with:

 \*    - Task input field

 \*    - Add task button

 \*    - Task list display

 \*    - Remove task functionality

 \* 2. Include the following features:

 \*    - Dynamic task list updates

 \*    - Task selection capability

 \*    - Task removal confirmation

 \*    - Scrollable task list

 \* 3. Design considerations:

 \*    - Clean and organized layout

 \*    - User-friendly interface

 \*    - Proper spacing and borders

 \*    - Responsive controls

 \*

 \* Learning Objectives:

 \* - Java Swing GUI development

 \* - Event handling in Java

 \* - Layout management

 \* - List management (DefaultListModel)

 \* - Component organization

 \*/

package Experiment9;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.border.EmptyBorder;

public class ToDoList extends JFrame {

    private DefaultListModel<String> listModel;

    private JList<String> todoList;

    private JTextField taskField;

    private JButton addButton;

    private JButton removeButton;

    public ToDoList() {

        setTitle("To-Do List");

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        setSize(400, 500);

        setLocationRelativeTo(null);

        // Create main panel with border layout

        JPanel mainPanel = new JPanel(new BorderLayout(10, 10));

        mainPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

        // Create input panel for task entry

        JPanel inputPanel = new JPanel(new BorderLayout(5, 0));

        taskField = new JTextField();

        addButton = new JButton("Add Task");

        inputPanel.add(taskField, BorderLayout.CENTER);

        inputPanel.add(addButton, BorderLayout.EAST);

        // Create list model and JList

        listModel = new DefaultListModel<>();

        todoList = new JList<>(listModel);

        todoList.setSelectionMode(ListSelectionModel.SINGLE\_SELECTION);

        JScrollPane scrollPane = new JScrollPane(todoList);

        // Create remove button

        removeButton = new JButton("Remove Selected Task");

        removeButton.setEnabled(false);

        // Add components to main panel

        mainPanel.add(inputPanel, BorderLayout.NORTH);

        mainPanel.add(scrollPane, BorderLayout.CENTER);

        mainPanel.add(removeButton, BorderLayout.SOUTH);

        // Add main panel to frame

        add(mainPanel);

        // Add action listeners

        addButton.addActionListener(e -> addTask());

        taskField.addActionListener(e -> addTask()); // Allow adding task with Enter key

        removeButton.addActionListener(e -> removeTask());

        // Add list selection listener

        todoList.addListSelectionListener(e -> {

            removeButton.setEnabled(todoList.getSelectedIndex() != -1);

        });

    }

    private void addTask() {

        String task = taskField.getText().trim();

        if (!task.isEmpty()) {

            listModel.addElement(task);

            taskField.setText("");

        }

        taskField.requestFocus();

    }

    private void removeTask() {

        int selectedIndex = todoList.getSelectedIndex();

        if (selectedIndex != -1) {

            listModel.remove(selectedIndex);

        }

    }

    public static void main(String[] args) {

        SwingUtilities.invokeLater(() -> {

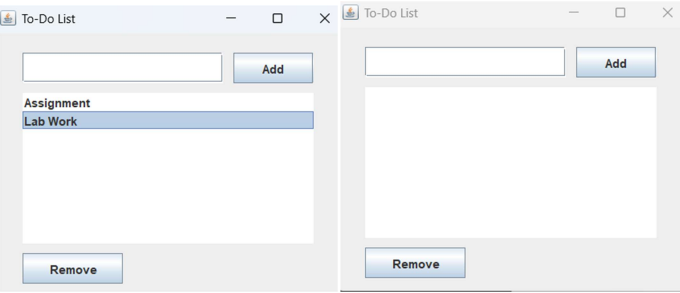
            new ToDoList().setVisible(true);

        });

    }

}

Output



1. Write a Swing application with JDBC to insert and retrieve employee details (ID, Name, Department, Salary) into/from a MySQL database. Use

JTextField for input

JButton for inserting data

JTable for displaying retrieved records

package Experiment9;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

import java.sql.\*;

import javax.swing.table.DefaultTableModel;

public class EmployeeManagement extends JFrame {

    private JTextField idField, nameField, deptField, salaryField;

    private JButton insertButton, viewButton, clearButton;

    private JTable employeeTable;

    private DefaultTableModel tableModel;

    private JLabel currencyLabel;

    // Database connection parameters

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

    private static final String USER = "root";

    private static final String PASSWORD = "";

    public EmployeeManagement() {

        setTitle("Employee Management System");

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        setSize(800, 600);

        setLocationRelativeTo(null);

        // Create main panel

        JPanel mainPanel = new JPanel(new BorderLayout(10, 10));

        mainPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        // Create input panel

        JPanel inputPanel = new JPanel(new GridLayout(5, 2, 5, 5));

        inputPanel.setBorder(BorderFactory.createTitledBorder("Employee Details"));

        // Add input fields

        inputPanel.add(new JLabel("Employee ID:"));

        idField = new JTextField();

        inputPanel.add(idField);

        inputPanel.add(new JLabel("Name:"));

        nameField = new JTextField();

        inputPanel.add(nameField);

        inputPanel.add(new JLabel("Department:"));

        deptField = new JTextField();

        inputPanel.add(deptField);

        JPanel salaryPanel = new JPanel(new BorderLayout(5, 0));

        inputPanel.add(new JLabel("Salary:"));

        salaryField = new JTextField();

        currencyLabel = new JLabel("₹");

        currencyLabel.setFont(new Font("Arial", Font.BOLD, 14));

        salaryPanel.add(currencyLabel, BorderLayout.WEST);

        salaryPanel.add(salaryField, BorderLayout.CENTER);

        inputPanel.add(salaryPanel);

        // Create button panel

        JPanel buttonPanel = new JPanel(new FlowLayout());

        insertButton = new JButton("Insert Record");

        viewButton = new JButton("View Records");

        clearButton = new JButton("Clear");

        buttonPanel.add(insertButton);

        buttonPanel.add(viewButton);

        buttonPanel.add(clearButton);

        inputPanel.add(buttonPanel);

        // Set button colors

        insertButton.setBackground(new Color(63, 81, 181));

        insertButton.setForeground(Color.WHITE);

        viewButton.setBackground(new Color(76, 175, 80));

        viewButton.setForeground(Color.WHITE);

        clearButton.setBackground(new Color(244, 67, 54));

        clearButton.setForeground(Color.WHITE);

        // Create table

        String[] columns = {"ID", "Name", "Department", "Salary"};

        tableModel = new DefaultTableModel(columns, 0);

        employeeTable = new JTable(tableModel);

        JScrollPane scrollPane = new JScrollPane(employeeTable);

        // Add components to main panel

        mainPanel.add(inputPanel, BorderLayout.NORTH);

        mainPanel.add(scrollPane, BorderLayout.CENTER);

        // Add main panel to frame

        add(mainPanel);

        // Add action listeners

        insertButton.addActionListener(e -> insertRecord());

        viewButton.addActionListener(e -> viewRecords());

        clearButton.addActionListener(e -> clearFields());

        // Add input validation

        salaryField.addKeyListener(new KeyAdapter() {

            public void keyTyped(KeyEvent e) {

                char c = e.getKeyChar();

                if (!Character.isDigit(c) && c != '.' && c != KeyEvent.VK\_BACK\_SPACE) {

                    e.consume();

                }

            }

        });

        idField.addKeyListener(new KeyAdapter() {

            public void keyTyped(KeyEvent e) {

                char c = e.getKeyChar();

                if (!Character.isDigit(c) && c != KeyEvent.VK\_BACK\_SPACE) {

                    e.consume();

                }

            }

        });

        // Create database table if not exists

        createTable();

    }

    private void createTable() {

        String sql = "CREATE TABLE IF NOT EXISTS employees " +

                    "(id INT PRIMARY KEY, " +

                    "name VARCHAR(100), " +

                    "department VARCHAR(100), " +

                    "salary DOUBLE)";

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

             Statement stmt = conn.createStatement()) {

            stmt.execute(sql);

        } catch (SQLException e) {

            JOptionPane.showMessageDialog(this, "Error creating table: " + e.getMessage());

        }

    }

    private void insertRecord() {

        try {

            int id = Integer.parseInt(idField.getText());

            String name = nameField.getText();

            String department = deptField.getText();

            double salary = Double.parseDouble(salaryField.getText());

            String sql = "INSERT INTO employees (id, name, department, salary) VALUES (?, ?, ?, ?)";

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

                 PreparedStatement pstmt = conn.prepareStatement(sql)) {

                pstmt.setInt(1, id);

                pstmt.setString(2, name);

                pstmt.setString(3, department);

                pstmt.setDouble(4, salary);

                pstmt.executeUpdate();

                JOptionPane.showMessageDialog(this, "Record inserted successfully!");

                clearFields();

                viewRecords(); // Refresh table

            }

        } catch (NumberFormatException e) {

            JOptionPane.showMessageDialog(this, "Please enter valid numeric values!");

        } catch (SQLException e) {

            JOptionPane.showMessageDialog(this, "Error inserting record: " + e.getMessage());

        }

    }

    private void viewRecords() {

        tableModel.setRowCount(0); // Clear existing records

        String sql = "SELECT \* FROM employees";

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

             Statement stmt = conn.createStatement();

             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {

                Object[] row = {

                    rs.getInt("id"),

                    rs.getString("name"),

                    rs.getString("department"),

                    String.format("₹%.2f", rs.getDouble("salary"))

                };

                tableModel.addRow(row);

            }

        } catch (SQLException e) {

            JOptionPane.showMessageDialog(this, "Error retrieving records: " + e.getMessage());

        }

    }

    private void clearFields() {

        idField.setText("");

        nameField.setText("");

        deptField.setText("");

        salaryField.setText("");

    }

    public static void main(String[] args) {

        // Load JDBC driver

        try {

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

        } catch (ClassNotFoundException e) {

            System.err.println("MySQL JDBC Driver not found.");

            return;

        }

        SwingUtilities.invokeLater(() -> {

            new EmployeeManagement().setVisible(true);

        });

    }

}

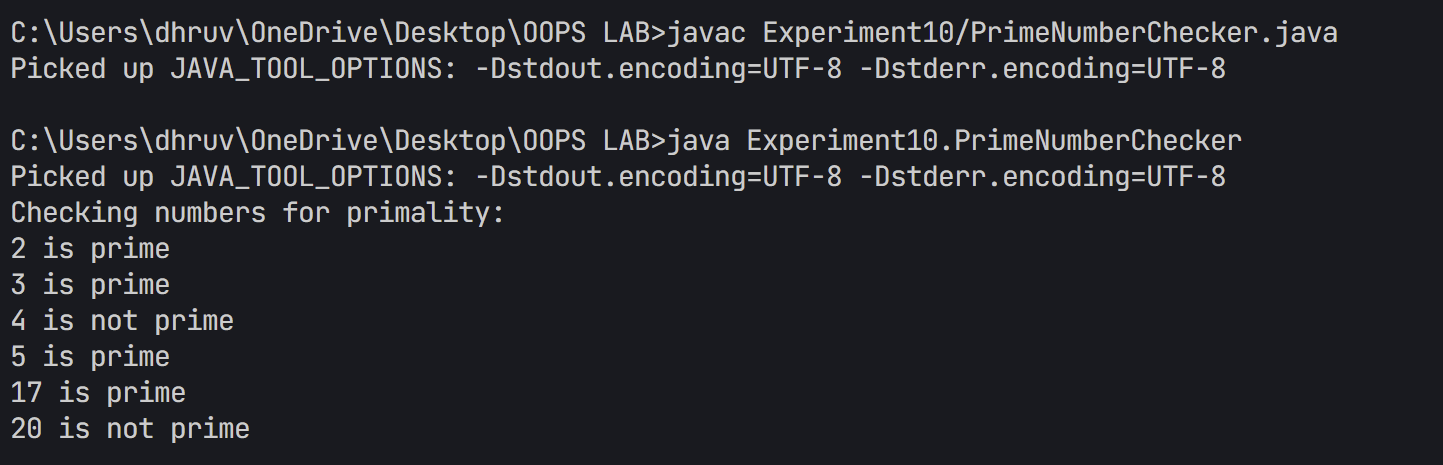
## Experiment 10

## TITLE Wrapper Classes and Collections

* + - Store a list of integers in an ArrayList<Integer> using autoboxing. Iterate through the list, unbox each value, and determine if it is a prime number, printing the results.

1. package Experiment10;
2. /\*\*
3. \* Problem Statement:
4. \* Write a Java program that demonstrates the use of ArrayList and autoboxing/unboxing to:
5. \* 1. Create an ArrayList of Integer objects
6. \* 2. Use autoboxing to add primitive integers to the ArrayList
7. \* 3. Check each number for primality using unboxing
8. \*
9. \* Requirements:
10. \* - Implement ArrayList to store numbers
11. \* - Demonstrate autoboxing when adding numbers
12. \* - Show unboxing when checking primality
13. \* - Create a method to check if a number is prime
14. \*
15. \* Expected Output:
16. \* - Display whether each number in the list is prime or not
17. \* - Show proper use of autoboxing and unboxing operations
18. \*/
19. import java.util.ArrayList;
20. public class PrimeNumberChecker {
21. public static void main(String[] args) {
22. // Create ArrayList and add numbers using autoboxing
23. ArrayList<Integer> numbers = new ArrayList<>();
24. numbers.add(2);  // Autoboxing: int to Integer
25. numbers.add(3);
26. numbers.add(4);
27. numbers.add(5);
28. numbers.add(17);
29. numbers.add(20);
30. // Iterate through list, unbox values and check for prime numbers
31. System.out.println("Checking numbers for primality:");
32. for (Integer num : numbers) {
33. int value = num;  // Unboxing: Integer to int
34. System.out.printf("%d is %s\n", value, isPrime(value) ? "prime" : "not prime");
35. }
36. }
37. // Method to check if a number is prime
38. private static boolean isPrime(int number) {
39. if (number <= 1) {
40. return false;
41. }
42. for (int i = 2; i <= Math.sqrt(number); i++) {
43. if (number % i == 0) {
44. return false;
45. }
46. }
47. return true;
48. }
49. }

Output



2. Create an ArrayList to store Employee objects with attributes like name, id, and salary. Add three employees, update one employee's salary, remove another by their id, and print the remaining employees.

package Experiment10;

import java.util.ArrayList;

/\*\*

 \* Question: Employee Management System using ArrayList

 \*

 \* Create a Java program that implements an Employee Management System using ArrayList

 \* to demonstrate the usage of Java Collections Framework and object manipulation.

 \*

 \* Requirements:

 \* 1. Implement Employee class with:

 \*    - Private member variables (encapsulation)

 \*    - Constructor for initialization

 \*    - Getter/Setter methods

 \*    - toString() method override

 \* 2. Demonstrate ArrayList operations:

 \*    - Adding employees

 \*    - Updating employee information

 \*    - Displaying employee list

 \*    - Basic data manipulation

 \*

 \* Learning Objectives:

 \* - Understanding ArrayList usage

 \* - Working with custom objects in collections

 \* - Implementing proper encapsulation

 \* - Object state manipulation

 \* - Collection traversal techniques

 \*/

class Employee {

    private String name;

    private int id;

    private double salary;

    public Employee(String name, int id, double salary) {

        this.name = name;

        this.id = id;

        this.salary = salary;

    }

    public int getId() {

        return id;

    }

    public void setSalary(double salary) {

        this.salary = salary;

    }

    @Override

    public String toString() {

        return String.format("Employee[id=%d, name='%s', salary=%.2f]", id, name, salary);

    }

}

public class EmployeeArrayList {

    public static void main(String[] args) {

        // Create ArrayList to store Employee objects

        ArrayList<Employee> employees = new ArrayList<>();

        // Add three employees

        employees.add(new Employee("John Doe", 101, 50000.0));

        employees.add(new Employee("Jane Smith", 102, 55000.0));

        employees.add(new Employee("Bob Johnson", 103, 45000.0));

        // Print initial list

        System.out.println("Initial Employee List:");

        employees.forEach(System.out::println);

        // Update salary of employee with ID 102

        for (Employee emp : employees) {

            if (emp.getId() == 102) {

                emp.setSalary(60000.0);

                System.out.println("\nUpdated salary for employee ID 102");

                break;

            }

        }

        // Remove employee with ID 103

        employees.removeIf(emp -> emp.getId() == 103);

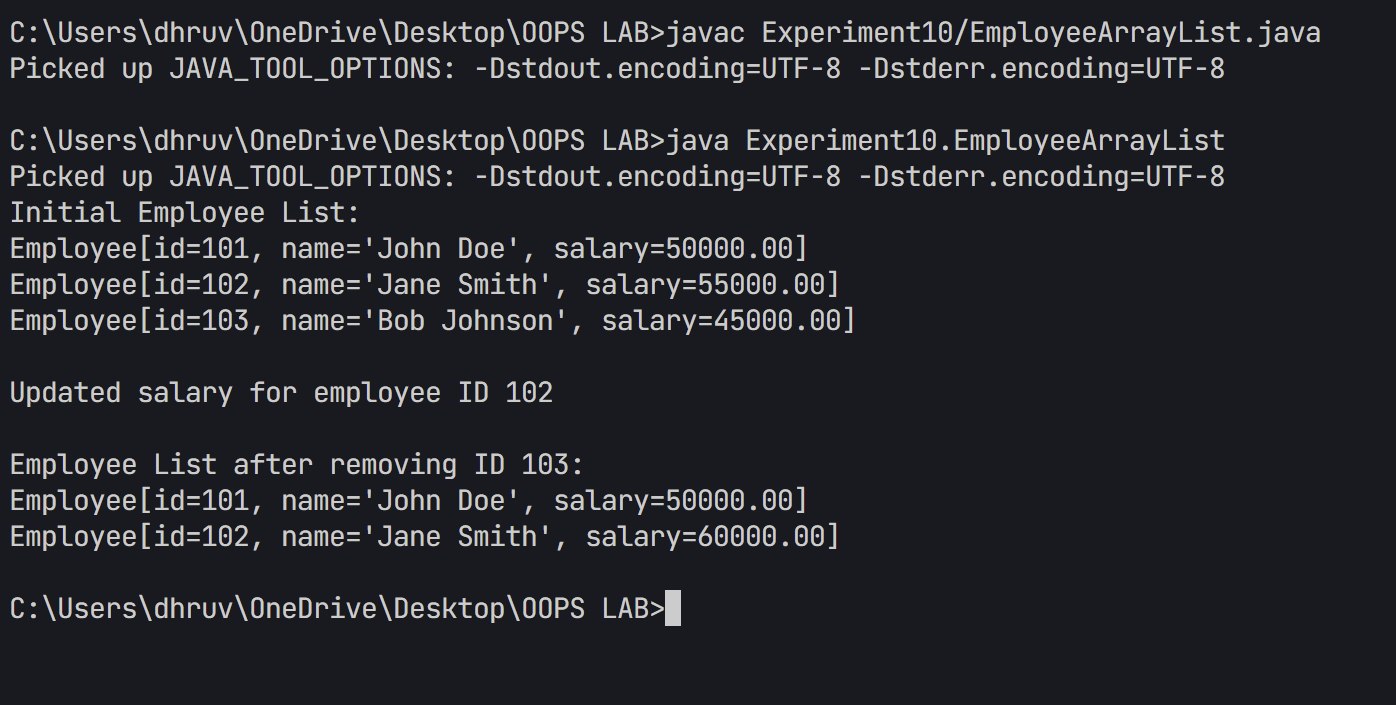
        System.out.println("\nEmployee List after removing ID 103:");

        employees.forEach(System.out::println);

    }

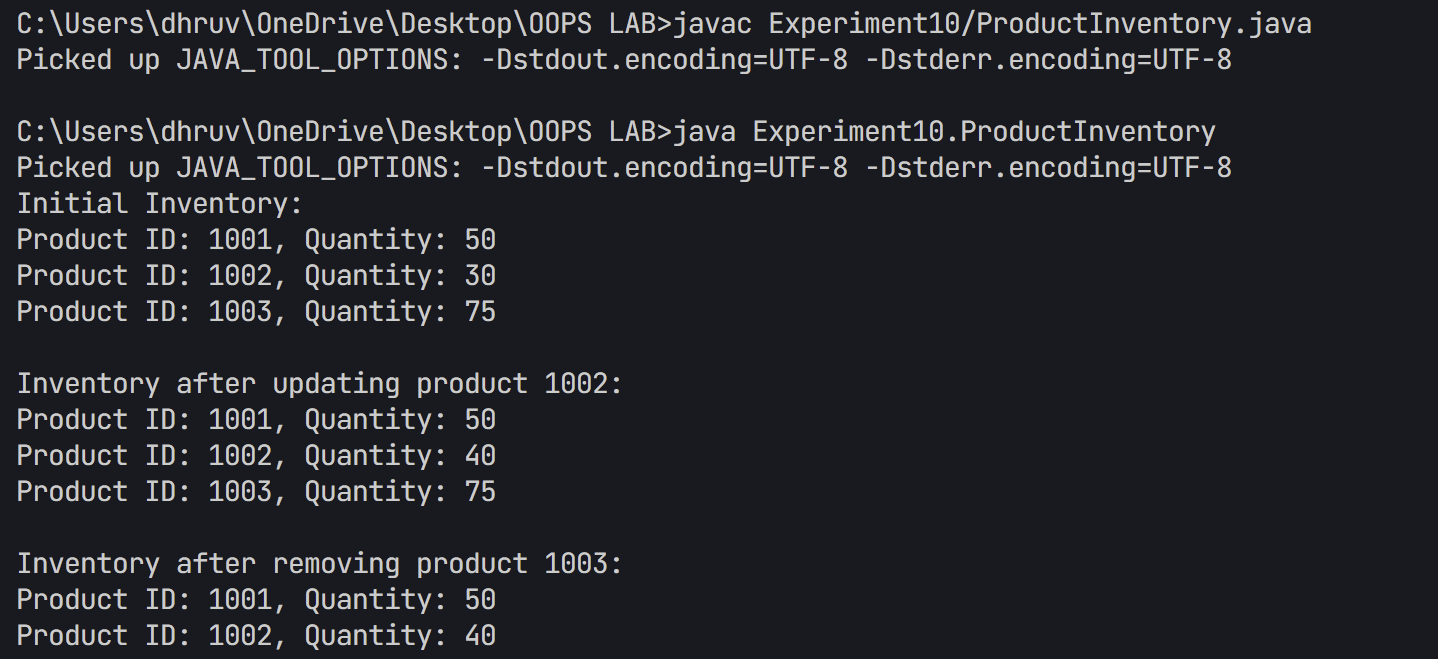
}

Output



1. Use a HashMap to manage a product inventory where keys are productId (Integer) and values are quantity (Integer). Add three products, update the quantity of one product, remove another, and display the final inventory.
2. package Experiment10;
3. /\*\*
4. \* Problem Statement:
5. \* Write a Java program that implements a simple Product Inventory System using HashMap to:
6. \* 1. Store product IDs and their corresponding quantities
7. \* 2. Perform basic inventory operations (add, update, remove)
8. \* 3. Display the current state of inventory
9. \*
10. \* Requirements:
11. \* - Use HashMap to store product information (Key: ProductID, Value: Quantity)
12. \* - Implement basic inventory operations:
13. \*   - Adding new products
14. \*   - Updating product quantities
15. \*   - Removing products
16. \* - Create a method to display inventory status
17. \*
18. \* Expected Output:
19. \* - Show initial inventory state
20. \* - Display updated inventory after modifications
21. \* - Demonstrate proper use of HashMap methods
22. \*/
23. import java.util.HashMap;
24. public class ProductInventory {
25. public static void main(String[] args) {
26. // Create HashMap to store product inventory
27. HashMap<Integer, Integer> inventory = new HashMap<>();
28. // Add three products (productId -> quantity)
29. inventory.put(1001, 50);  // Product 1001: 50 units
30. inventory.put(1002, 30);  // Product 1002: 30 units
31. inventory.put(1003, 75);  // Product 1003: 75 units
32. // Display initial inventory
33. System.out.println("Initial Inventory:");
34. displayInventory(inventory);
35. // Update quantity of product 1002
36. inventory.put(1002, 40);  // Update product 1002 to 40 units
37. System.out.println("\nInventory after updating product 1002:");
38. displayInventory(inventory);
39. // Remove product 1003
40. inventory.remove(1003);
41. System.out.println("\nInventory after removing product 1003:");
42. displayInventory(inventory);
43. }
44. private static void displayInventory(HashMap<Integer, Integer> inventory) {
45. inventory.forEach((productId, quantity) ->
46. System.out.printf("Product ID: %d, Quantity: %d\n", productId, quantity));
47. }
48. }

Output



4. Given an array of names with duplicates (e.g., ["Aman", "Varchasv", "Divyansh", "Varchasv", ”Aman”]), store them in a HashSet to eliminate duplicates. Check if a specific name exists in the set and print the unique names.

package Experiment10;

/\*\*

 \* Problem Statement:

 \* Write a Java program that demonstrates the use of HashSet to remove duplicate names from an array.

 \* The program should:

 \* 1. Create an array of strings containing duplicate names

 \* 2. Use HashSet to eliminate duplicate entries

 \* 3. Display both original and unique names

 \* 4. Demonstrate membership testing using contains() method

 \* 5. Show the total count of unique names

 \*

 \* Requirements:

 \* - Use String array to store initial names with duplicates

 \* - Convert array to HashSet using Arrays.asList()

 \* - Use forEach() method to display unique names

 \* - Demonstrate contains() method for membership testing

 \* - Show size() method usage for counting unique elements

 \*/

import java.util.HashSet;

import java.util.Arrays;

public class NameDeduplication {

    public static void main(String[] args) {

        // Create array of names with duplicates

        String[] names = {"Aman", "Varchasv", "Divyansh", "Varchasv", "Aman"};

        // Create HashSet and add all names

        HashSet<String> uniqueNames = new HashSet<>(Arrays.asList(names));

        // Print original array

        System.out.println("Original array of names:");

        System.out.println(Arrays.toString(names));

        // Print unique names

        System.out.println("\nUnique names in HashSet:");

        uniqueNames.forEach(System.out::println);

        // Check if specific names exist

        String nameToCheck = "Varchasv";

        System.out.printf("\nIs '%s' in the set? %b\n",

                         nameToCheck, uniqueNames.contains(nameToCheck));

        nameToCheck = "John";

        System.out.printf("Is '%s' in the set? %b\n",

                         nameToCheck, uniqueNames.contains(nameToCheck));

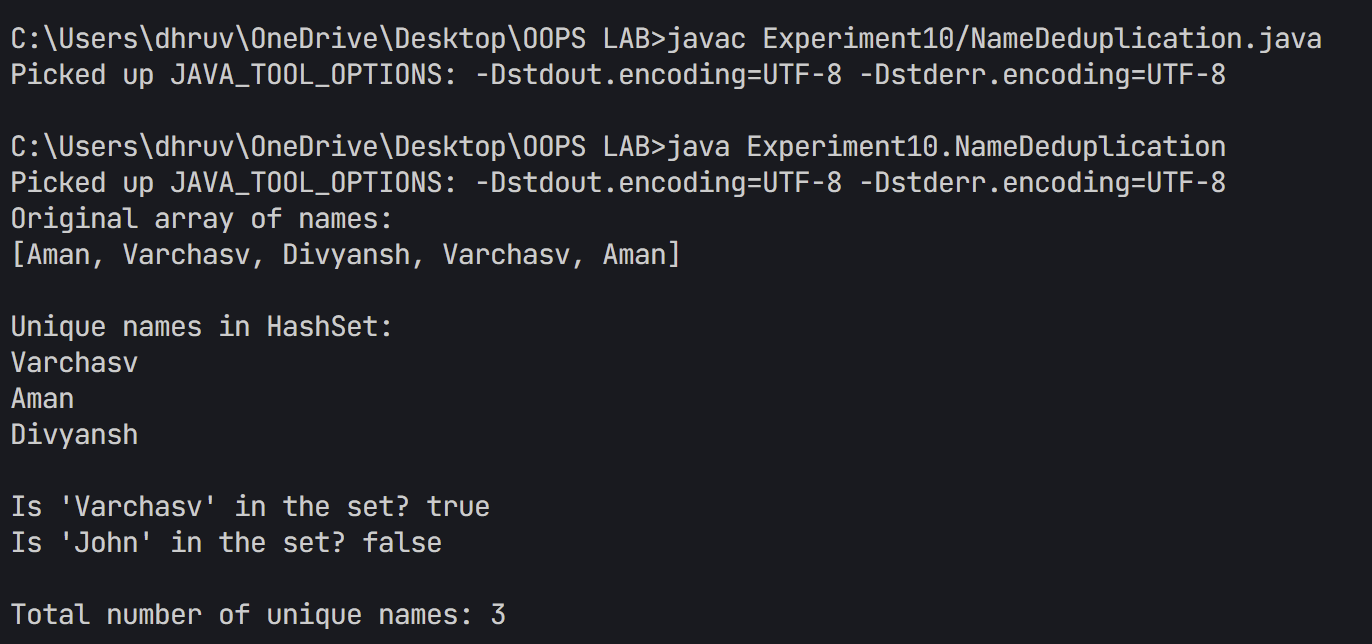
        // Print total count of unique names

        System.out.printf("\nTotal number of unique names: %d\n", uniqueNames.size());

    }

}

Output



5. Given an ArrayList of integers (possibly with duplicates), find the sum of its unique values. [Hint: Convert it to HashSet]

Example:

Input: [3, 5, 3, 8, 2, 5]

Unique values: [3, 5, 8, 2]

Sum: 18

package Experiment10;

/\*\*

 \* Problem Statement:

 \* Write a Java program that demonstrates the use of ArrayList and HashSet to:

 \* 1. Create an ArrayList with duplicate integer values

 \* 2. Convert the ArrayList to HashSet to remove duplicates

 \* 3. Calculate and display the sum of unique values

 \*

 \* Requirements:

 \* - Use ArrayList to store initial numbers with duplicates

 \* - Use HashSet to eliminate duplicate values

 \* - Demonstrate the difference between original and unique values

 \* - Calculate sum of unique values using enhanced for loop

 \*

 \* Expected Output:

 \* - Display original ArrayList with duplicates

 \* - Show unique values after conversion to HashSet

 \* - Print the sum of unique values

 \*/

import java.util.ArrayList;

import java.util.HashSet;

import java.util.Arrays;

public class UniqueSum {

    public static void main(String[] args) {

        // Create ArrayList with duplicate values

        ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(3, 5, 3, 8, 2, 5));

        // Print original list

        System.out.println("Original ArrayList: " + numbers);

        // Convert to HashSet to get unique values

        HashSet<Integer> uniqueNumbers = new HashSet<>(numbers);

        System.out.println("Unique values: " + uniqueNumbers);

        // Calculate sum of unique values

        int sum = 0;

        for (Integer num : uniqueNumbers) {

            sum += num;

        }

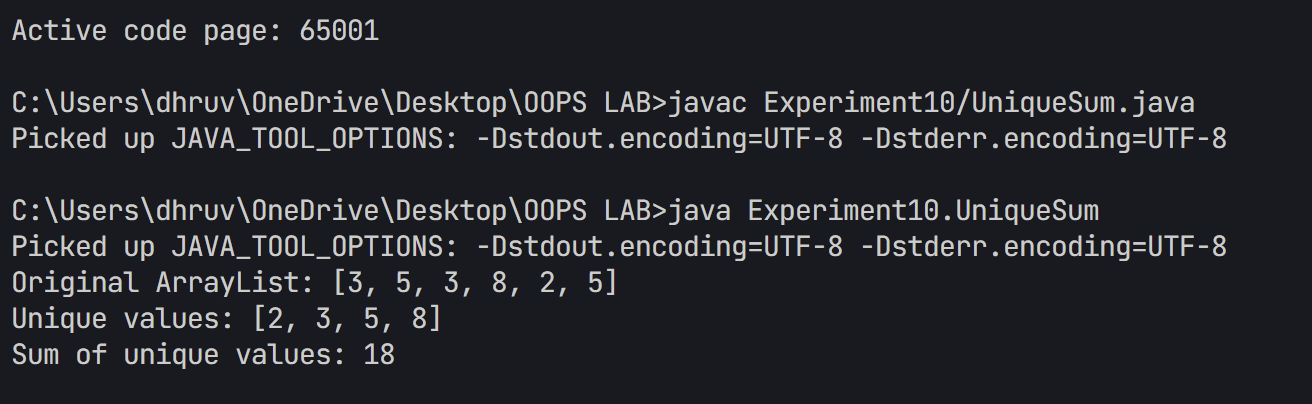
        // Print the sum

        System.out.println("Sum of unique values: " + sum);

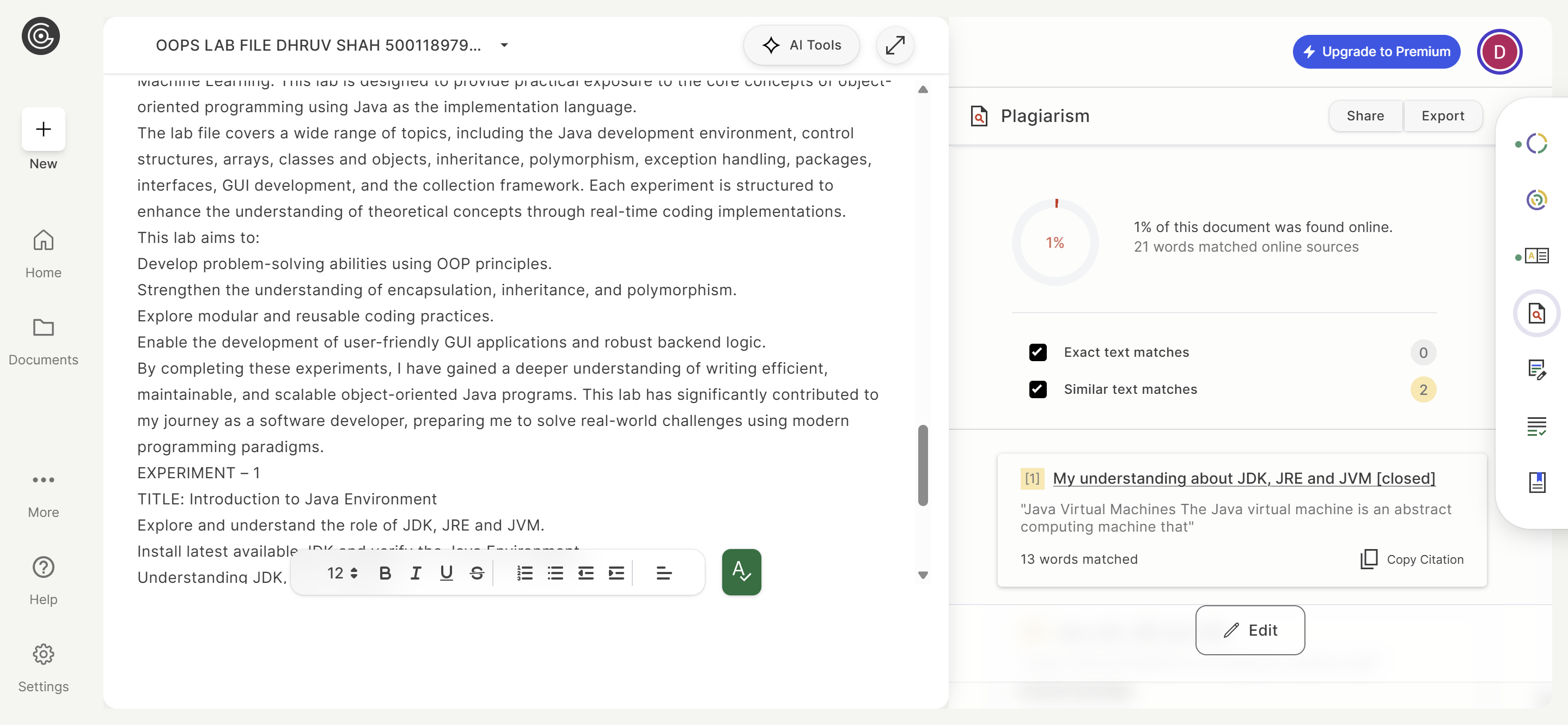
    }

}

Output



**PLAIGARISM REPORT-**

****