ADVANCED JAVA PROGRAMMING LAB

(LAB MANUAL)

B.E. II Year III Semester

Department of Computer Science and Engineering

Chennai Institute of Technology

An AICTE Recognized- Accredited by NBA, NAAC - Affiliated to Anna University

LIST OF PROGRAMS

1. Write a program to demonstrate the use of multidimensional arrays and looping constructs.
2. Write a program to demonstrate the applications of string handling functions.
3. Write a program to demonstrate the use of inheritance.
4. Write a program to demonstrate the applications of user defined packages and sub packages.
5. Write a program to demonstrate the use of Java exception Handling methods.
6. Write a program to demonstrate the use of threads in java.
7. Demonstrate with a program the use of file handling methods in Java.
8. Demonstrate the use of Java collection frameworks in reducing application development time.
9. Write a program to register students data using JDBC with MySQL database.
10. Develop applications to demonstrate the features of generics classes.

1. Theatre Seat Book arrangement

Problem Statement:

Write a Java program that demonstrates the use of multidimensional arrays and looping constructs to create, manipulate, and print a 2D array representing a simple seating arrangement in a theatre.

Expected Learning Outcomes:

1. Understanding the concept of multidimensional arrays in Java.

2. Practicing the use of nested loops for array manipulation.

3. Developing problem-solving skills by representing and working with real-world data.

Problem Analysis:

You are tasked with creating a program that models a seating arrangement in a theatre. The theatre has a fixed number of rows and columns, and you need to create a 2D array to represent the seats. Each seat can be either empty (0) or occupied (1). The program should allow you to mark seats as occupied and print the current seating arrangement.

Input:

1. Number of rows and columns in the theatre.

2. Commands to mark seats as occupied. Each command consists of a row and column number.

Output:

1. The current seating arrangement after each command.

Algorithm:

1. Create a 2D array to represent the theatre with the specified number of rows and columns, initialized with all zeros (empty seats).

2. Enter a loop to process commands until the user chooses to exit.

   a. Prompt the user to enter a command (row and column number).

   b. Check if the input is valid (within the theatre boundaries).

   c. If valid, mark the seat as occupied (set the corresponding element in the array to 1).

   d. Print the current seating arrangement.

   e. Repeat the loop until the user chooses to exit.

Java Code:

```
import java.util.Scanner;
public class TheaterSeatingArrangement {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input: Number of rows and columns in the theater
        System.out.print("Enter the number of rows in the theater: ");
        int numRows = scanner.nextInt();
        System.out.print("Enter the number of columns in the theater: ");
        int numCols = scanner.nextInt();
        // Create a 2D array to represent the theater
        int[][] theater = new int[numRows][numCols];
        // Main loop to process commands
        while (true) {
            // Print the current seating arrangement
            System.out.println("Current Seating Arrangement:");
            for (int i = 0; i < numRows; i++) {
                for (int j = 0; j < numCols; j++) {
                    System.out.print(theater[i][j] + " ");
                }
                System.out.println();
            }
            // Prompt for a command (row and column)
            System.out.print("Enter a command (row and column) or 'exit' to quit: ");
            String input = scanner.next();
            if (input.equals("exit")) {
                break; // Exit the program
            }
            // Parse row and column from the input
            int row = Integer.parseInt(input);
            int col = scanner.nextInt();
            // Check if the input is valid
            if (row >= 0 && row < numRows && col >= 0 && col < numCols) {
                // Mark the seat as occupied
                theater[row][col] = 1;
                System.out.println("Seat at row " + row + ", column " + col + " is now occupied.");
            } else {
                System.out.println("Invalid input. Please enter valid row and column numbers.");
            }
        }
        System.out.println("Thank you for using the Theater Seating Arrangement program!");
    }
}
```

This Java program allows the user to interactively mark seats as occupied in a theatre's seating arrangement and displays the updated arrangement after each command. The program continues until the user chooses to exit.

Test case :

Expected Output	
	```
	Enter the number of rows in the theater: 3
	Enter the number of columns in the theater: 4

	Current Seating Arrangement:
	0 0 0 0
	0 0 0 0
	0 0 0 0

	Enter a command (row and column) or 'exit' to quit: 1 2
	Seat at row 1, column 2 is now occupied.

	Current Seating Arrangement:
	0 0 0 0
	0 0 1 0
	0 0 0 0

	Enter a command (row and column) or 'exit' to quit: 0 3
	Seat at row 0, column 3 is now occupied.

	Current Seating Arrangement:
	0 0 0 1
	0 0 1 0
	0 0 0 0

	Enter a command (row and column) or 'exit' to quit: 2 2
	```

	Seat at row 2, column 2 is now occupied. Current Seating Arrangement: 0 0 0 1 0 0 1 0 0 0 0 0 Enter a command (row and column) or 'exit' to quit: 4 4 Invalid input. Please enter valid row and column numbers. Enter a command (row and column) or 'exit' to quit: exit Thank you for using the Theater Seating Arrangement program!
Actual Output	

Viva :

2. String Handling Functions

Problem Statement:

Write a Java program that demonstrates the applications of string handling functions by processing and manipulating a text-based user profile.

Expected Learning Outcomes:

1. Understanding the basics of string handling in Java.

2. Practicing the use of string manipulation functions.

3. Developing problem-solving skills for processing textual data.

Problem Analysis:

You are given a text-based user profile containing the user's name, email address, and a short bio. The program should allow users to perform the following tasks:

1. Display the user's name.

2. Display the user's email address.

3. Display the user's bio.

4. Update the user's email address.

5. Update the user's bio.

6. Exit the program.

Input:

1. The user's name.

2. The user's initial email address.

3. The user's initial bio.

4. User's choice of action (1-6) to perform on the user profile.

Output:

Depending on the user's choice, the program will display the corresponding information, update the user's profile, or exit.

Algorithm:

1. Initialize variables for the user's name, email address, and bio.

2. Enter a loop to display a menu of options and process the user's choice until the user chooses to exit.

   a. Display the menu of options.

   b. Read the user's choice.

   c. Use a switch statement to perform the selected action:

    1. Display the user's name.
    2. Display the user's email address.
    3. Display the user's bio.
    4. Update the user's email address.
    5. Update the user's bio.
    6. Exit the program.
  d. Repeat the loop until the user chooses to exit.

Java Code:

```
import java.util.Scanner;
public class UserProfileManager {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input: User's initial profile
        System.out.print("Enter user's name: ");
        String name = scanner.nextLine();
        System.out.print("Enter user's email address: ");
        String email = scanner.nextLine();
        System.out.print("Enter user's bio: ");
        String bio = scanner.nextLine();
        // Main loop to process user actions
        while (true) {
            // Display menu of options
            System.out.println("\nUser Profile Options:");
            System.out.println("1. Display name");
            System.out.println("2. Display email address");
            System.out.println("3. Display bio");
            System.out.println("4. Update email address");
            System.out.println("5. Update bio");
            System.out.println("6. Exit");

            // Read user's choice
```

```java
System.out.print("Enter your choice (1-6): ");
int choice = scanner.nextInt();
scanner.nextLine(); // Consume the newline character
switch (choice) {
  case 1:
    // Display the user's name
    System.out.println("Name: " + name);
    break;
  case 2:
    // Display the user's email address
    System.out.println("Email Address: " + email);
    break;
  case 3:
    // Display the user's bio
    System.out.println("Bio: " + bio);
    break;
  case 4:
    // Update the user's email address
    System.out.print("Enter new email address: ");
    email = scanner.nextLine();
    System.out.println("Email address updated successfully.");
    break;
  case 5:
    // Update the user's bio
    System.out.print("Enter new bio: ");
    bio = scanner.nextLine();
    System.out.println("Bio updated successfully.");
    break;
  case 6:
    // Exit the program
    System.out.println("Goodbye!");
```

```
                    System.exit(0);
                default:
                    System.out.println("Invalid choice. Please select a valid option (1-6).");
            }
        }
    }
}
```

This Java program allows users to manage a text-based user profile by displaying information, updating email addresses and bios, and exiting the program. It demonstrates the use of string handling functions for input, output, and manipulation of textual data.

Test case:

Expected Output	Enter user's name: John Doe Enter user's email address: john.doe@example.com Enter user's bio: A software developer User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 1 Name: John Doe User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 4 Enter new email address: john.new@example.com Email address updated successfully. User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 2 Email Address: john.new@example.com

	User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 5 Enter new bio: Experienced software developer Bio updated successfully. User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 3 Bio: Experienced software developer User Profile Options: 1. Display name 2. Display email address 3. Display bio 4. Update email address 5. Update bio 6. Exit Enter your choice (1-6): 6 Goodbye!
Actual Output	

Viva :

3. Inheritance model for vehicles – cars, Bicycle

Problem Statement:

Write a Java program that demonstrates the use of inheritance by modelling a simple hierarchy of vehicles, including cars and bicycles, with shared and specialized attributes and behaviours.

Expected Learning Outcomes:

1. Understanding the concept of inheritance in Java.

2. Practicing the creation of classes and subclasses.

3. Demonstrating the use of super classes and subclasses to model relationships.

Problem Analysis:

You are tasked with creating a program that models vehicles. There are two types of vehicles: cars and bicycles. Both vehicles have common attributes such as brand and speed, but they also have specialized attributes and behaviours. The program should demonstrate the use of inheritance by creating classes for vehicles, cars, and bicycles, and show how inherited attributes and methods can be used.

Input:

No user input is required for this program.

Output:

The program will demonstrate the attributes and behaviors of vehicles, cars, and bicycles.

Algorithm:

1. Create a superclass named "Vehicle" with common attributes such as "brand" and "speed" and methods like "accelerate" and "brake."

2. Create a subclass named "Car" that inherits from "Vehicle" and adds specialized attributes like "numDoors" and "fuelType."

3. Create another subclass named "Bicycle" that also inherits from "Vehicle" and adds specialized attributes like "numGears" and "bikeType."

4. In the main method, create instances of "Car" and "Bicycle," set their attributes, and demonstrate their behaviors.

Java Code:

```java
// Superclass: Vehicle

class Vehicle {

    private String brand;

    private double speed;

    public Vehicle(String brand) {
        this.brand = brand;
        this.speed = 0.0;
    }
    public void accelerate(double amount) {
        speed += amount;
        System.out.println(brand + " is accelerating. Current speed: " + speed + " km/h");
    }
    public void brake(double amount) {
        speed -= amount;
        System.out.println(brand + " is braking. Current speed: " + speed + " km/h");
    }
}
// Subclass: Car
class Car extends Vehicle {
    private int numDoors;
    private String fuelType;
    public Car(String brand, int numDoors, String fuelType) {
        super(brand);
        this.numDoors = numDoors;
        this.fuelType = fuelType;
    }
    public void honk() {
        System.out.println(getBrand() + " is honking!");
    }
    public String getBrand() {
        return super.brand;
    }
}
// Subclass: Bicycle
class Bicycle extends Vehicle {
    private int numGears;
    private String bikeType;
    public Bicycle(String brand, int numGears, String bikeType) {
        super(brand);
        this.numGears = numGears;
        this.bikeType = bikeType;
    }
    public void ringBell() {
        System.out.println(getBrand() + " is ringing the bell!");
    }
    public String getBrand() {
        return super.brand;
    }
}
public class VehicleInheritanceDemo {
    public static void main(String[] args) {
        // Create a car and a bicycle
        Car myCar = new Car("Toyota", 4, "Gasoline");
        Bicycle myBicycle = new Bicycle("Schwinn", 21, "Mountain Bike");
        // Demonstrate car's and bicycle's attributes and behaviors
        System.out.println("Car Brand: " + myCar.getBrand());
        myCar.accelerate(40);
        myCar.brake(10);
        myCar.honk();
        System.out.println("\nBicycle Brand: " + myBicycle.getBrand());
        myBicycle.accelerate(20);
        myBicycle.brake(5);
        myBicycle.ringBell();
    }
}
```

This Java program demonstrates the use of inheritance by modelling a hierarchy of vehicles, including cars and bicycles, with shared and specialized attributes and behaviours. It shows how attributes and methods from the superclass ("Vehicle") are inherited by the subclasses ("Car" and "Bicycle").

Test case :

Expected Output	Car Brand: Toyota
	Toyota is accelerating. Current speed: 40.0 km/h
	Toyota is braking. Current speed: 30.0 km/h
	Toyota is honking!
	Bicycle Brand: Schwinn
	Schwinn is accelerating. Current speed: 20.0 km/h
	Schwinn is braking. Current speed: 15.0 km/h
	Schwinn is ringing the bell!
Actual Output	

Viva :

4. User defined packages and sub-packages for student management system

Problem Statement:

Write a Java program that demonstrates the creation and use of user-defined packages and sub-packages by organizing classes for a simple student management system.

Expected Learning Outcomes:

1. Understanding the concept of packages and sub-packages in Java.

2. Practicing the creation and organization of classes into packages and sub-packages.

3. Demonstrating how to import and use classes from different packages and sub-packages.

Problem Analysis:

You are tasked with creating a program that models a simple student management system. The program should use user-defined packages and sub-packages to organize classes related to students, courses, and student enrollment. This demonstrates the concept of packages and sub-packages in Java and how to import and use classes from different packages and sub-packages.

Input:

No user input is required for this program.

Output:

The program will demonstrate the organization of classes into packages and perform student management operations.

Algorithm:

1. Create a package named "university" to contain classes related to the university management system.

2. Create sub-packages within the "university" package for "students" and "courses."

3. Inside the "students" sub-package, create a class for managing students (e.g., "StudentManager").

4. Inside the "courses" sub-package, create a class for managing courses (e.g., "CourseManager").

5. Create a class outside the packages (e.g., "UniversityDemo") to demonstrate the use of classes from the "university" package and its sub-packages.

6. In the "UniversityDemo" class, import and use classes from the "students" and "courses" sub-packages to perform student management operations.

Java Code:

```
// Create a package "university"

package university;

// Create a sub-package "students" inside "university"

package university.students;

public class StudentManager {

    public void enrollStudent(String studentName, String courseName) {

        System.out.println("Enrolled student " + studentName + " in course " + courseName);

    }

    public void graduateStudent(String studentName) {

        System.out.println("Graduated student " + studentName);

    }

}

// Create a sub-package "courses" inside "university"

package university.courses;

public class CourseManager {

    public void createCourse(String courseName) {

        System.out.println("Created course " + courseName);

    }

    public void deleteCourse(String courseName) {

        System.out.println("Deleted course " + courseName);

    }

}

// Create a class "UniversityDemo" to demonstrate the use of packages and sub-packages

import university.students.StudentManager;

import university.courses.CourseManager;

public class UniversityDemo {

    public static void main(String[] args) {

        // Create instances of StudentManager and CourseManager from the respective sub-
packages

        StudentManager studentManager = new StudentManager();

        CourseManager courseManager = new CourseManager();
        // Demonstrate student management operations
        studentManager.enrollStudent("Alice", "Math");
        courseManager.createCourse("Physics");
    }
}
```

In this Java program, user-defined packages and sub-packages are created to organize classes for a simple student management system. The "UniversityDemo" class demonstrates the use of classes from the "students" and "courses" sub-packages to perform student management operations.

Test case :

Expected Output	Enrolled student Alice in course Math Created course Physics
Actual Output	

Viva :

5. Java Exception Handling for File Operations

Problem Statement:

Write a Java program that demonstrates the use of Java exception handling methods to handle various types of exceptions that may occur during file operations.

Expected Learning Outcomes:

1. Understanding the concept of exception handling in Java.
2. Learning how to use try-catch blocks to handle exceptions.
3. Practicing the use of specific exception classes for different error scenarios.

Problem Analysis:

You are tasked with creating a program that performs file operations such as reading from and writing to files. Your program should demonstrate the use of exception handling methods to gracefully handle errors that may occur during file operations, including file not found, permission denied, and input/output errors.

Input:

No user input is required for this program.

Output:

The program will demonstrate the handling of exceptions and display error messages when specific errors occur during file operations.

Algorithm:

```
1. Create a method named "performFileOperations" to simulate file operations.
2. Inside the method:
   a. Attempt to read from a non-existent file.
   b. Attempt to write to a read-only file.
   c. Attempt to read from a file with invalid content.
   d. Catch and handle specific exceptions that may occur during these operations.
3. In the main method, call the "performFileOperations" method to demonstrate exception handling.
```

Java Code:

```java
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;
```

```java
public class ExceptionHandlingDemo {
    public static void main(String[] args) {
        performFileOperations();
    }
    public static void performFileOperations() {
        try {
            // Attempt to read from a non-existent file
            FileReader fileReader = new FileReader("non_existent_file.txt");
            BufferedReader reader = new BufferedReader(fileReader);
            String line = reader.readLine();
            reader.close();
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
            System.out.println("File not found or unable to read.");
        }
        try {
            // Attempt to write to a read-only file
            FileWriter fileWriter = new FileWriter("read_only_file.txt");
            fileWriter.write("This is a write operation.");
            fileWriter.close();
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
            System.out.println("Permission denied for writing.");
        }
        try {
            // Attempt to read from a file with invalid content
            FileReader fileReader = new FileReader("invalid_content.txt");
            BufferedReader reader = new BufferedReader(fileReader);
            String line = reader.readLine();
            int number = Integer.parseInt(line); // This line will throw a NumberFormatException
            reader.close();
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
            System.out.println("Error reading file.");
        } catch (NumberFormatException e) {
            System.out.println("NumberFormatException: " + e.getMessage());
            System.out.println("Invalid content in the file.");
        }
    }
}
```

In this Java program, we demonstrate the use of exception handling methods to handle various types of exceptions that may occur during file operations. The program attempts to perform file operations that result in exceptions, catches those exceptions, and displays corresponding error messages.

Test case :

Expected Output	IOException: non_existent_file.txt (No such file or directory) File not found or unable to read. IOException: read_only_file.txt (Permission denied) Permission denied for writing. IOException: invalid_content.txt (Is a directory) Error reading file. NumberFormatException: null Invalid content in the file.
Actual Output	

Viva :

6. Multi-Thread scenario to print 1-10 by each thread

Problem Statement:

Write a Java program that demonstrates the use of threads to simulate a simple multi-threaded scenario where multiple threads perform tasks concurrently.

Expected Learning Outcomes:

1. Understanding the concept of threads and multi-threading in Java.

2. Learning how to create and manage threads using the Java Thread class.

3. Demonstrating the benefits of multi-threading in handling concurrent tasks.

Problem Analysis:

You are tasked with creating a program that simulates a scenario where multiple threads perform tasks concurrently. In this program, we'll create three threads, each representing a worker, and each worker will perform a simple task of counting from 1 to 10. The program should demonstrate the concurrent execution of these threads and their coordination.

Input:

No user input is required for this program.

Output:

The program will demonstrate the concurrent execution of three threads, each counting from 1 to 10.

Algorithm:

```
1. Create a class "Worker" that extends the Thread class.

2. Override the "run" method in the "Worker" class to define the task to be performed by each worker (counting from 1 to 10).

3. In the main method:

   a. Create three instances of the "Worker" class, representing three workers.

   b. Start each worker using the "start" method (which internally calls the "run" method).

   c. Use the "join" method to wait for each worker to complete its task before proceeding.

   d. Print a message indicating that all workers have finished their tasks.
```

Java Code:

```
class Worker extends Thread {

    private String name;

    public Worker(String name) {

        this.name = name;
    }
    @Override
    public void run() {
        System.out.println(name + " is starting.");
        for (int i = 1; i <= 10; i++) {
            System.out.println(name + " - Count: " + i);
            try {
                // Sleep for a random amount of time (simulating work)
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(name + " is done.");
    }
}
public class ThreadDemo {
    public static void main(String[] args) {
        System.out.println("Main thread is starting.");
        // Create three worker threads
        Worker worker1 = new Worker("Worker 1");
        Worker worker2 = new Worker("Worker 2");
        Worker worker3 = new Worker("Worker 3");
        // Start the worker threads
        worker1.start();
        worker2.start();
        worker3.start();
        try {
            // Wait for all worker threads to finish
            worker1.join();
            worker2.join();
            worker3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Main thread is done.");
    }
}
```

In this Java program, we create three worker threads ("Worker 1," "Worker 2," and "Worker 3") using the Thread class. Each worker counts from 1 to 10 with random delays between counts, simulating concurrent tasks. The program demonstrates the use of threads and shows how they execute concurrently, with the main thread waiting for all worker threads to finish before completing.

Test case :

Expected Output	Main thread is starting. Worker 1 is starting. Worker 2 is starting. Worker 3 is starting. Worker 1 - Count: 1 Worker 2 - Count: 1 Worker 3 - Count: 1 Worker 1 - Count: 2 Worker 2 - Count: 2 Worker 3 - Count: 2 Worker 3 - Count: 3 Worker 2 - Count: 3 Worker 1 - Count: 3 Worker 3 - Count: 4 Worker 2 - Count: 4 Worker 1 - Count: 4 Worker 1 - Count: 5 Worker 3 - Count: 5 Worker 2 - Count: 5 Worker 3 - Count: 6 Worker 2 - Count: 6 Worker 1 - Count: 6 Worker 3 - Count: 7 Worker 2 - Count: 7 Worker 1 - Count: 7 Worker 1 - Count: 8 Worker 3 - Count: 8 Worker 2 - Count: 8 Worker 3 - Count: 9 Worker 2 - Count: 9 Worker 1 - Count: 9 Worker 2 - Count: 10 Worker 1 - Count: 10

	Worker 3 - Count: 10 Worker 1 is done. Worker 2 is done. Worker 3 is done. Main thread is done.
Actual Output	

Viva :

7. File Handling Methods

Problem Statement:

Write a Java program that demonstrates the use of file handling methods to read data from a text file, manipulate the data, and write the modified data back to the file.

Expected Learning Outcomes:

1. Understanding the concept of file handling in Java.

2. Learning how to read data from a text file.

3. Learning how to manipulate data and write it back to a file.

Problem Analysis:

You are tasked with creating a program that reads data from a text file, performs some manipulation on the data (e.g., converting text to uppercase), and then writes the modified data back to the file. In this program, we will read a text file, convert its contents to uppercase, and save the modified content back to the same file.

Input:

The input for this program is a text file named "input.txt" containing some text data.

Output:

The program will read the data from "input.txt," convert it to uppercase, and write the modified data back to the same file.

Algorithm:

1. Open the input file ("input.txt") for reading.

2. Create a StringBuilder to store the modified data.

3. Read each line from the input file.

4. For each line, convert it to uppercase and append it to the StringBuilder.

5. Close the input file.

6. Open the same file for writing (this will overwrite the existing content).

7. Write the modified data from the StringBuilder to the file.

8. Close the output file.

Java Code:

```
import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;
import java.io.IOException;
public class FileHandlingDemo {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        try {
            // Step 1: Open the input file for reading
            FileReader fileReader = new FileReader(inputFile);
            BufferedReader reader = new BufferedReader(fileReader);
            // Step 2: Create a StringBuilder to store the modified data
            StringBuilder modifiedData = new StringBuilder();
            // Step 3: Read each line from the input file
            String line;
            while ((line = reader.readLine()) != null) {
                // Step 4: Convert each line to uppercase and append it to the StringBuilder
                modifiedData.append(line.toUpperCase()).append("\n");
            }
            // Step 5: Close the input file
            reader.close();
            // Step 6: Open the same file for writing (this will overwrite the existing content)
            FileWriter fileWriter = new FileWriter(inputFile);
            BufferedWriter writer = new BufferedWriter(fileWriter);
            // Step 7: Write the modified data from the StringBuilder to the file
            writer.write(modifiedData.toString());
            // Step 8: Close the output file
            writer.close();
            System.out.println("File processing complete.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this Java program, we demonstrate file handling methods to read data from a text file, convert it to uppercase, and write the modified data back to the same file. The program handles file reading and writing using BufferedReader and BufferedWriter, respectively.

Test case :

Expected Output	File processing complete.
Actual Output	

Viva :

8. Java Collection Framework for data manipulation using List, Hash, Map

Problem Statement:

Write a Java program that demonstrates the use of Java collection frameworks to efficiently perform common data manipulation tasks, such as searching, sorting, and filtering a list of student records.

Expected Learning Outcomes:

1. Understanding the importance of Java collection frameworks in application development.

2. Learning how to use collections like ArrayList and HashMap for common data manipulation tasks.

3. Demonstrating how collections can reduce development time and improve code readability.

Problem Analysis:

You are tasked with creating a program that manages student records using Java collection frameworks. The program should use an ArrayList to store student objects, and a HashMap to store student IDs as keys and corresponding student objects as values. It should demonstrate how collections can simplify common tasks such as searching for a student, sorting students by criteria, and filtering students based on specific attributes.

Input:

No user input is required for this program. Student records are predefined in the program.

Output:

The program will demonstrate various operations on student records, including searching, sorting, and filtering, and display the results.

Algorithm:

```
1. Create a Student class with attributes like ID, name, age, and GPA.
2. Create an ArrayList to store student objects.
3. Create a HashMap to store student IDs as keys and student objects as values.
4. Add several student records to the ArrayList and HashMap.
5. Perform the following operations:
   a. Search for a student by ID and display their details.
   b. Sort the students by GPA and display the sorted list.
   c. Filter students who are older than a certain age and display the filtered list.
6. Implement each operation using Java collection frameworks (ArrayList and HashMap).
```

Java Code:

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
class Student {
    private int id;
    private String name;
    private int age;
    private double gpa;
    public Student(int id, String name, int age, double gpa) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.gpa = gpa;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public double getGpa() {
        return gpa;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
            ", name='" + name + '\'' +
            ", age=" + age +
            ", gpa=" + gpa +
            '}';
    }
}
public class CollectionFrameworkDemo {
    public static void main(String[] args) {
        // Create an ArrayList to store student records
        List<Student> students = new ArrayList<>();

        // Create a HashMap to store student IDs as keys and student objects as values
        Map<Integer, Student> studentMap = new HashMap<>();

        // Add student records to the ArrayList and HashMap
        students.add(new Student(101, "Alice", 20, 3.8));
        students.add(new Student(102, "Bob", 22, 3.6));
        students.add(new Student(103, "Charlie", 21, 3.9));
        students.add(new Student(104, "David", 19, 3.7));
        for (Student student : students) {
            studentMap.put(student.getId(), student);
        }
        // Search for a student by ID and display their details
        int searchId = 103;
        Student searchedStudent = studentMap.get(searchId);
        System.out.println("Searched Student: " + searchedStudent);
        // Sort the students by GPA and display the sorted list
        students.sort((s1, s2) -> Double.compare(s2.getGpa(), s1.getGpa()));
        System.out.println("Sorted Students by GPA:");
        for (Student student : students) {
            System.out.println(student);
        }
        // Filter students who are older than a certain age and display the filtered list
        int filterAge = 20;
        List<Student> filteredStudents = new ArrayList<>();
        for (Student student : students) {
            if (student.getAge() > filterAge) {
                filteredStudents.add(student);
            }
        }
        System.out.println("Students older than " + filterAge + " years:");
        for (Student student : filteredStudents) {
            System.out.println(student);
        }
    }
}
```

In this Java program, we demonstrate the use of Java collection frameworks (ArrayList and HashMap) to efficiently manage and manipulate student records. The program performs operations like searching, sorting, and filtering on the student records, showcasing the benefits of using collections in reducing application development time and improving code readability.

Test case :

Expected Output	Searched Student: Student{id=103, name='Charlie', age=21, gpa=3.9}
	Sorted Students by GPA:
	Student{id=103, name='Charlie', age=21, gpa=3.9}
	Student{id=101, name='Alice', age=20, gpa=3.8}
	Student{id=102, name='Bob', age=22, gpa=3.6}
	Student{id=104, name='David', age=19, gpa=3.7}
	Students older than 20 years:
	Student{id=103, name='Charlie', age=21, gpa=3.9}
	Student{id=102, name='Bob', age=22, gpa=3.6}

Actual Output	

Viva :

9. Java JDBC connectivity using MySQL for Student Registration Portal

Problem Statement:

Write a Java program to register student data using JDBC (Java Database Connectivity) with a MySQL database. The program should allow users to input student details, validate the data, and insert the data into a MySQL database table.

Expected Learning Outcomes:

1. Understanding the basics of JDBC for database connectivity.

2. Learning how to establish a connection to a MySQL database from Java.

3. Practicing data validation and insertion into a database table.

Problem Analysis:

You are tasked with creating a program that allows users to register student data and store it in a MySQL database. The program should collect student details such as name, age, and course, validate the data to ensure it meets certain criteria, and insert the data into a database table.

Input:

The user is required to input the following student details:

1. Student name (String)

2. Student age (Integer)

3. Student course (String)

Output:

The program will display messages indicating the registration status, including success or failure.

Algorithm:

1. Set up the MySQL database connection parameters (URL, username, password).

2. Import the necessary JDBC libraries.

3. Create a method to collect student details (name, age, course) from the user.

4. Validate the collected data:

   a. Ensure that the name and course are not empty.

   b. Ensure that the age is a positive integer.

5. Establish a database connection using JDBC.

6. Create an SQL INSERT statement to insert the student data into a database table.

7. Execute the SQL INSERT statement to insert the data.

8. Display a success message if the data is successfully inserted, or an error message if there is a problem.

Java Code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;
public class StudentRegistration {
  // Database connection parameters
  private static final String DB_URL = "jdbc:mysql://localhost:3306/studentdb";
  private static final String DB_USER = "root";
  private static final String DB_PASSWORD = "your_password";
  public static void main(String[] args) {
    try {
      // Load the MySQL JDBC driver
      Class.forName("com.mysql.cj.jdbc.Driver");
      // Collect student details from the user
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter student name: ");
      String name = scanner.nextLine();
      System.out.print("Enter student age: ");
      int age = scanner.nextInt();
      scanner.nextLine(); // Consume the newline character
      System.out.print("Enter student course: ");
      String course = scanner.nextLine();
      // Validate the collected data
      if (name.isEmpty() || course.isEmpty() || age <= 0) {
        System.out.println("Invalid input. Please provide valid student details.");
        return;
      }
      // Establish a database connection
```

```
            Connection connection = DriverManager.getConnection(DB_URL, DB_USER,
DB_PASSWORD);
            // Create an SQL INSERT statement
            String insertSQL = "INSERT INTO students (name, age, course) VALUES (?, ?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(insertSQL);
            preparedStatement.setString(1, name);
            preparedStatement.setInt(2, age);
            preparedStatement.setString(3, course);
            // Execute the SQL INSERT statement
            int rowsAffected = preparedStatement.executeUpdate();
            if (rowsAffected > 0) {
                System.out.println("Student registration successful.");
            } else {
                System.out.println("Student registration failed.");
            }
            // Close the database connection
            connection.close();
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

In this Java program, we use JDBC to register student data in a MySQL database. The program collects student details, validates the input, establishes a database connection, and inserts the data into a database table. It demonstrates the basics of JDBC and database connectivity with MySQL. Make sure to replace `"your_password"` with your actual MySQL password in the `DB_PASSWORD` variable.

Test case :

Expected Output	Enter student name: Alice Enter student age: 20 Enter student course: Computer Science Student registration successful

Actual Output	

Viva :

10. Stack operations using Generic Classes using Collection Framework ArrayList

Problem Statement:

Develop a Java application that demonstrates the features of generic classes by implementing a generic stack data structure. The program should allow users to push and pop elements of different data types onto/from the stack and display the stack contents.

Expected Learning Outcomes:

1. Understanding the concept of generic classes in Java.

2. Learning how to create and use generic classes to work with various data types.

3. Demonstrating the flexibility and type safety provided by generics.

Problem Analysis:

You are tasked with creating a program that implements a generic stack data structure. The program should allow users to push elements onto the stack and pop elements from the stack, irrespective of the data type of the elements. This demonstrates the use of generic classes in Java to create reusable and type-safe data structures.

Input:

The user can input elements of various data types to be pushed onto the stack.

Output:

The program will display the stack contents after each push or pop operation.

Algorithm:

1. Create a generic class named "GenericStack" that represents a stack data structure.

2. Use a generic type parameter to make the class generic.

3. Implement the stack operations, including push and pop.

4. Create an instance of the "GenericStack" class to demonstrate its usage with different data types.

Java Code:

```
import java.util.ArrayList;

import java.util.List;

class GenericStack<T> {

    private List<T> stack;

    public GenericStack() {

        stack = new ArrayList<>();

    }
```

```java
    public void push(T element) {
        stack.add(element);
        System.out.println("Pushed: " + element);
    }
    public T pop() {
        if (!isEmpty()) {
            T element = stack.remove(stack.size() - 1);
            System.out.println("Popped: " + element);
            return element;
        } else {
            System.out.println("Stack is empty.");
            return null;
        }
    }
    public boolean isEmpty() {
        return stack.isEmpty();
    }
    public void displayStack() {
        System.out.println("Stack Contents: " + stack);
    }
}
public class GenericStackDemo {
    public static void main(String[] args) {
        // Create a generic stack for integers
        GenericStack<Integer> intStack = new GenericStack<>();
        intStack.push(10);
        intStack.push(20);
        intStack.displayStack();
        intStack.pop();
        intStack.displayStack();
        // Create a generic stack for strings
        GenericStack<String> stringStack = new GenericStack<>();
        stringStack.push("Hello");
        stringStack.push("World");
        stringStack.displayStack();
        stringStack.pop();
        stringStack.displayStack();
        // Create a generic stack for doubles
        GenericStack<Double> doubleStack = new GenericStack<>();
        doubleStack.push(3.14);
        doubleStack.push(2.71);
        doubleStack.displayStack();
        doubleStack.pop();
        doubleStack.displayStack();
    }
}
```

In this Java program, we demonstrate the features of generic classes by implementing a generic stack data structure. The program allows users to push and pop elements of different data types onto/from the stack and displays the stack contents. The use of generic classes makes it possible to create a type-safe and reusable stack for various data types.

Test case :

Expected Output	Pushed: 10 Pushed: 20 Stack Contents: [10, 20] Popped: 20 Stack Contents: [10] Pushed: Hello Pushed: World Stack Contents: [Hello, World] Popped: World Stack Contents: [Hello] Pushed: 3.14 Pushed: 2.71 Stack Contents: [3.14, 2.71] Popped: 2.71 Stack Contents: [3.14]
Actual Output	

Viva :