Experiment 4

StudentName:Disha Singh **UID:22BCS13898**
Branch: CSE **Section: 22BCS_IOT-637/A**
Semester: 6th **DOP:14/02/25**
Subject: PBLJ **Subject Code:22CSH-359**

Aim: Write a Java program to implement an ArrayList that stores employee details (ID, Name, and Salary). Allow users to add, update, remove, and search employees.

Objective: To implement an ArrayList in Java for managing employee details (ID, Name, Salary) with functionalities to **add, update, remove, search, and display** employee records.

Algorithm:

Algorithm for Employee Management Using ArrayList in Java

Step 1: Initialize the Program

1. Start the program.
2. Import ArrayList and Scanner classes.
3. Define a class Employee with attributes:
   - id (int)
   - name (String)
   - salary (double)
4. Create an ArrayList<Employee> to store employee records.

Step 2: Implement Employee Management Functionalities

1. **Add Employee**
   1. Prompt the user for ID, Name, and Salary.
   2. Create an Employee object.
   3. Add it to the ArrayList.
2. **Update Employee**
   1. Prompt the user for ID of the employee to update.
   2. Search the ArrayList for the matching ID.
   3. If found, prompt for new Name and Salary, then update.
   4. If not found, display "Employee not found."
3. **Remove Employee**
   1. Prompt the user for ID of the employee to remove.
   2. Search the ArrayList for the matching ID.
   3. If found, remove the employee.
   4. If not found, display "Employee not found."
4. **Search Employee**
   1. Prompt the user for ID or Name.
   2. Search the ArrayList for a match.
   3. If found, display employee details.
   4. If not found, display "Employee not found."
5. **Display Employees**
   1. Iterate through the ArrayList and display all employees.

2. If the list is empty, display "No employees available."

Step 3: Display Menu Options

1. Display options:
    - 1. Add Employee
    - 2. Update Employee
    - 3. Remove Employee
    - 4. Search Employee
    - 5. Display All Employees
    - 6. Exit
2. Prompt the user for a choice.

Step 4: Handle User Input

1. Loop until the user chooses to exit:
    - Read the user's input.
    - Call the appropriate function:
        - 1 → Add Employee
        - 2 → Update Employee
        - 3 → Remove Employee
        - 4 → Search Employee
        - 5 → Display All Employees
        - 6 → Exit
2. Handle invalid inputs:
    - If input is non-numeric, display "Invalid input, try again."
    - If an invalid option is chosen, display "Invalid choice, please try again."

Step 5: Terminate the Program

1. When the user selects Exit (6), terminate the loop.
2. Display "Program exited. Thank you!" and end execution.

Code:

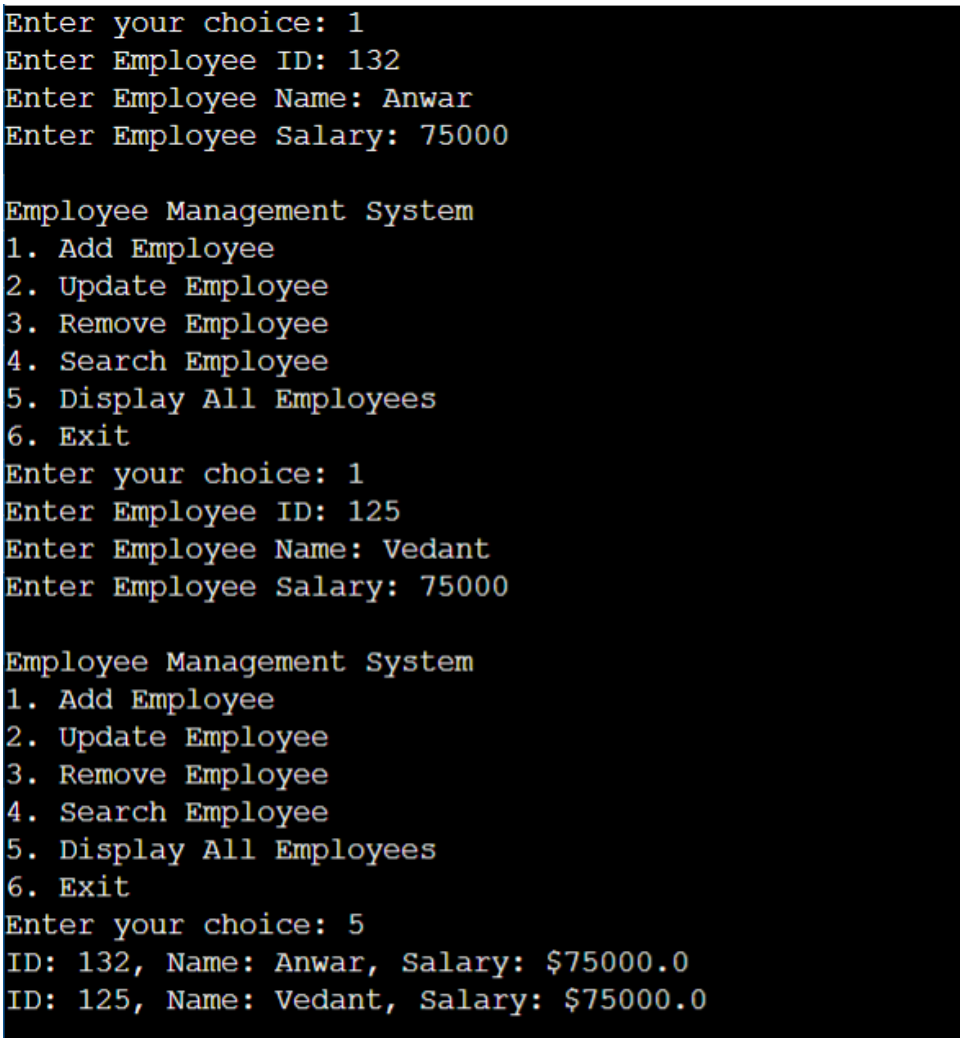
```
import java.util.ArrayList;
import java.util.Scanner;

class Employee {
    int id;
    String name;
    double salary;

    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public void displayDetails() {
        System.out.println("ID: " + id + ", Name: " + name + ", Salary: $" + salary);
    }
}
```

```
public class EmployeeManagementSystem {
    private static ArrayList<Employee> employeeList = new ArrayList<>();
    private static Scanner scanner = new Scanner(System.in);

    public static void addEmployee(int id, String name, double salary) {
        employeeList.add(new Employee(id, name, salary));
    }

    public static void updateEmployee(int id, String newName, double newSalary) {
        for (Employee employee : employeeList) {
            if (employee.id == id) {
                employee.name = newName;
                employee.salary = newSalary;
                System.out.println("Employee updated successfully!");
                return;
            }
        }
        System.out.println("Employee with ID " + id + " not found.");
    }

    public static void removeEmployee(int id) {
        for (Employee employee : employeeList) {
            if (employee.id == id) {
                employeeList.remove(employee);
                System.out.println("Employee removed successfully.");
                return;
            }
        }
        System.out.println("Employee with ID " + id + " not found.");
    }

    public static void searchEmployee(int id) {
        for (Employee employee : employeeList) {
            if (employee.id == id) {
                employee.displayDetails();
                return;
            }
        }
        System.out.println("Employee with ID " + id + " not found.");
    }

    public static void displayAllEmployees() {
        if (employeeList.isEmpty()) {
            System.out.println("No employees in the system.");
        } else {
            for (Employee employee : employeeList) {
                employee.displayDetails();
            }
        }
    }

    public static void main(String[] args) {
        while (true) {
```

```
System.out.println("\nEmployee Management System");
System.out.println("1. Add Employee");
System.out.println("2. Update Employee");
System.out.println("3. Remove Employee");
System.out.println("4. Search Employee");
System.out.println("5. Display All Employees");
System.out.println("6. Exit");
System.out.print("Enter your choice: ");

int choice = scanner.nextInt();
scanner.nextLine();

switch (choice) {
    case 1:
        System.out.print("Enter Employee ID: ");
        int id = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Enter Employee Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Employee Salary: ");
        double salary = scanner.nextDouble();
        addEmployee(id, name, salary);
        break;

    case 2:
        System.out.print("Enter Employee ID to update: ");
        int updateId = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Enter New Name: ");
        String newName = scanner.nextLine();
        System.out.print("Enter New Salary: ");
        double newSalary = scanner.nextDouble();
        updateEmployee(updateId, newName, newSalary);
        break;

    case 3:
        System.out.print("Enter Employee ID to remove: ");
        int removeId = scanner.nextInt();
        removeEmployee(removeId);
        break;

    case 4:
        System.out.print("Enter Employee ID to search: ");
        int searchId = scanner.nextInt();
        searchEmployee(searchId);
        break;

    case 5:
        displayAllEmployees();
        break;

    case 6:
        System.out.println("Exiting...");
```

```
                scanner.close();
                return;

            default:
                System.out.println("Invalid choice, please try again.");
        }
    }
}
}
```

Output:



Learning Outcomes:

- Understand how to use **ArrayList** in Java to store and manage dynamic data collections.
- Implement **CRUD (Create, Read, Update, Delete)** operations efficiently using Java.
- Develop a **menu-driven console application** for user interaction.
- Apply **object-oriented programming (OOP) principles** like encapsulation and abstraction.
- Handle **user input and exceptions** effectively to ensure program robustness.

Experiment 4.2

1.Aim: Create a program to collect and store all the cards to assist the users in finding all the cards in a given symbol using Collection interface.

2.Objective: To develop a Java program using the **Collection interface** to store and manage a set of playing cards. The program will allow users to **add, retrieve, and search** for all cards of a given symbol efficiently. This ensures organized card management and easy lookup based on user input.

3.Algorithm:

Step 1: Initialize the Program

1. Start the program.
2. Import java.util.* for using collections and Scanner.
3. Define a class Card with attributes:
    - rank (String)
    - suit (String)
4. Implement a constructor to initialize Card objects.
5. Implement displayCard() method to print the card details.

Step 2: Define CardCollection Class

1. Create a class CardCollection.
2. Define a List<Card> named deck to store all card objects.

Step 3: Initialize the Deck

1. Define initializeDeck() method:
    - Create arrays for ranks (2–10, J, Q, K, A) and suits (Hearts, Diamonds, Clubs, Spades).
    - Loop through each suit and rank to create a Card object.
    - Add each Card to the deck list.

Step 4: Implement Search Functionality

1. Define findCardsBySuit(String suit):
    - Iterate through the deck.
    - If the card's suit matches the user input, display it.
    - If no match is found, print "No cards found for the suit."

Step 5: Implement Display Functionality

1. Define displayAllCards() method:
    - If the deck is empty, print "The deck is empty."
    - Otherwise, iterate and display all cards in the deck.

Step 6: Display Menu Options

1. In the main() method, create a Scanner object for user input.
2. Call initializeDeck() to populate the deck.
3. Display options:
    - 1. Display All Cards
    - 2. Find Cards by Suit
    - 3. Exit
4. Prompt the user for a choice.

Step 7: Handle User Input

1. Loop until the user chooses to exit:
    - Read the user's input.

- Call the appropriate function:
  - 1 → displayAllCards()
  - 2 → findCardsBySuit() (Prompt user for suit input)
  - 3 → Exit

2. Handle invalid inputs:
   - If input is invalid, print "Invalid choice. Please try again."

Step 8: Terminate the Program

1. If the user selects Exit (3), close the Scanner and end execution.
2. Print "Exiting... Goodbye!" before terminating.

4. Implementation Code:

```
import java.util.*;

class Card {
    String rank;
    String suit;

    public Card(String rank, String suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public void displayCard() {
        System.out.println(rank + " of " + suit);
    }
}

public class CardCollection {

    private static List<Card> deck = new ArrayList<>();

    public static void initializeDeck() {
        String[] ranks = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
        String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};

        for (String suit : suits) {
            for (String rank : ranks) {
                deck.add(new Card(rank, suit));
            }
        }
    }

    public static void findCardsBySuit(String suit) {
        boolean found = false;
        for (Card card : deck) {
            if (card.suit.equalsIgnoreCase(suit)) {
                card.displayCard();
                found = true;
            }
        }
        if (!found) {
            System.out.println("No cards found for the suit: " + suit);
        }
```

```
    }

    public static void displayAllCards() {
        if (deck.isEmpty()) {
            System.out.println("The deck is empty.");
        } else {
            for (Card card : deck) {
                card.displayCard();
            }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        initializeDeck();

        while (true) {
            System.out.println("\nCard Collection System");
            System.out.println("1. Display All Cards");
            System.out.println("2. Find Cards by Suit");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");

            int choice = scanner.nextInt();
            scanner.nextLine();

            switch (choice) {
                case 1:
                    displayAllCards();
                    break;

                case 2:
                    System.out.print("Enter the suit (Hearts, Diamonds, Clubs, Spades) to find: ");
                    String suit = scanner.nextLine();
                    findCardsBySuit(suit);
                    break;

                case 3:
                    System.out.println("Exiting... Goodbye!");
                    scanner.close();
                    return;

                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

5.Output

```
Card Collection System
1. Display All Cards
2. Find Cards by Suit
3. Exit
Enter your choice: 2
Enter the suit (Hearts, Diamonds, Clubs, Spades) to find: Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
J of Hearts
Q of Hearts
K of Hearts
A of Hearts
```

6.Learning Outcomes:

- Understand how to use the Collection framework (ArrayList) to store and manage objects dynamically.
- Implement search functionality to filter and retrieve specific data from a collection.
- Develop a menu-driven Java program to interact with users efficiently.
- Apply object-oriented programming (OOP) principles such as encapsulation and class design.
- Enhance problem-solving skills by handling user input validation and iterative operations.

Experiment 4.3

1. **Aim:** Develop a ticket booking system with synchronized threads to ensure no double booking of seats. Use thread priorities to simulate VIP bookings being processed first.

2. **Objective:** To develop a **multi-threaded Ticket Booking System** in Java that ensures **synchronized seat booking**, preventing double bookings. The system will use **thread priorities** to simulate **VIP bookings being processed first**, demonstrating effective thread management, synchronization, and priority handling in concurrent programming.

3. **Algorithm:**

Step 1: Initialize the Program

1. Start the program.
2. Import java.util.* and java.util.concurrent.* for thread handling.
3. Define a class TicketBookingSystem with:
   - A List<Boolean> representing seat availability (true for available, false for booked).
   - A synchronized method bookSeat(int seatNumber, String passengerName) to ensure thread safety.

Step 2: Implement Seat Booking Logic

1. Define bookSeat(int seatNumber, String passengerName):
   - If the seat is available (true), mark it as booked (false).
   - Print confirmation: "Seat X booked successfully by Y".
   - If already booked, print: "Seat X is already booked."

Step 3: Define Booking Threads

1. Create a class PassengerThread extending Thread:
   - Store passenger name, seat number, and booking system reference.
   - Implement run() method to call bookSeat().

Step 4: Assign Thread Priorities

1. Create VIP and Regular passenger threads.
2. Set higher priority for VIP passengers using setPriority(Thread.MAX_PRIORITY).
3. Set default priority for regular passengers.

Step 5: Handle User Input & Simulate Booking

1. In main(), create an instance of TicketBookingSystem.
2. Accept number of seats and bookings from the user.
3. Create multiple PassengerThread instances for VIP and regular passengers.
4. Start all threads using start().

Step 6: Synchronization & Preventing Double Booking

1. Use the synchronized keyword in bookSeat() to ensure only one thread accesses it at a time.
2. Ensure thread execution order by assigning higher priority to VIP threads.

Step 7: Display Final Booking Status

1. After all threads finish execution, display the list of booked seats.
2. End the program with a message: "All bookings completed successfully."

4.Implementation Code:

```
class TicketBookingSystem {
    private int totalSeats;
    private int bookedSeats;

    public TicketBookingSystem(int totalSeats) {
```

```
        this.totalSeats = totalSeats;
        this.bookedSeats = 0;
    }

    public synchronized boolean bookTicket(String customerName, boolean isVIP) {
        if (bookedSeats < totalSeats) {
            bookedSeats++;
            System.out.println(customerName + " successfully booked a seat.");
            return true;
        } else {
            System.out.println("Sorry " + customerName + ", no seats available.");
            return false;
        }
    }

    public int getAvailableSeats() {
        return totalSeats - bookedSeats;
    }
}

class Customer extends Thread {
    private String customerName;
    private TicketBookingSystem bookingSystem;
    private boolean isVIP;

    public Customer(String customerName, TicketBookingSystem bookingSystem, boolean
isVIP) {
        this.customerName = customerName;
        this.bookingSystem = bookingSystem;
        this.isVIP = isVIP;
    }

    @Override
    public void run() {
        if (isVIP) {
            System.out.println(customerName + " is a VIP. Processing VIP
booking...");
        } else {
            System.out.println(customerName + " is a regular customer. Processing
booking...");
        }

        boolean booked = bookingSystem.bookTicket(customerName, isVIP);

        if (booked) {
            System.out.println(customerName + " has successfully booked the ticket.");
        } else {
            System.out.println(customerName + " failed to book a ticket due to no
available seats.");
        }
    }
}
```

```
public class TicketBookingApp {
    public static void main(String[] args) {
            TicketBookingSystem bookingSystem = new TicketBookingSystem(5);

            Customer customer1 = new Customer("John", bookingSystem, true);
            Customer customer2 = new Customer("Jane", bookingSystem, false);
            Customer customer3 = new Customer("Sam", bookingSystem, false);
            Customer customer4 = new Customer("Mia", bookingSystem, true);
            Customer customer5 = new Customer("Alex", bookingSystem, false);
            Customer customer6 = new Customer("Oliver", bookingSystem, true);

            customer1.setPriority(Thread.MAX_PRIORITY);
            customer2.setPriority(Thread.NORM_PRIORITY);
            customer3.setPriority(Thread.NORM_PRIORITY);
            customer4.setPriority(Thread.MAX_PRIORITY);
            customer5.setPriority(Thread.NORM_PRIORITY);
            customer6.setPriority(Thread.MAX_PRIORITY);

            customer1.start();
            customer2.start();
            customer3.start();
            customer4.start();
            customer5.start();
            customer6.start();
    }
}
```

5. **Output:**

```
Anwar is a VIP. Processing VIP booking...
Vedant is a regular customer. Processing booking...
Ambuj is a VIP. Processing VIP booking...
Shivi is a VIP. Processing VIP booking...
Aditya is a regular customer. Processing booking...
Sahil is a regular customer. Processing booking...
Anwar successfully booked a seat.
Anwar has successfully booked the ticket.
Ambuj successfully booked a seat.
Ambuj has successfully booked the ticket.
Sahil successfully booked a seat.
Shivi successfully booked a seat.
Aditya successfully booked a seat.
Aditya has successfully booked the ticket.
Shivi has successfully booked the ticket.
Sahil has successfully booked the ticket.
Sorry Vedant, no seats available.
Vedant failed to book a ticket due to no available seats.
```

6. Learning Outcomes:

- Understand thread synchronization to prevent race conditions and ensure safe seat booking.
- Learn how to use thread priorities to manage VIP and regular bookings efficiently.
- Implement multi-threading concepts in Java for concurrent execution.
- Gain experience in handling shared resources using synchronization techniques.
- Develop a real-world ticket booking system demonstrating practical thread management.