Experiment 4

Student Name: Akshita Sharma **UID: 22BCS15804**

Branch: BE/CSE **Section/Group: IOT_618/B**

Semester: 6th **Date of Performance: 21/02/25**

Subject Name: Project based learning in Java

Subject Code: 22CSH- 359

Easy Level

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

2. **Objective:** To implement an Employee Management System using Array List in java that allows users to add, update, remove, and search employee details (ID, Name, and Salary).

3. **Implementation/Code:**

```
import java.util.*;
class Employee {
    int id;
    String name;
    double salary;

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

```
        this.name = name;
        this.salary = salary;
    }

    public String toString() {
        return id + " - " + name + " - " + salary;
    }
}

public class EmployeeManager {
    public static void main(String[] args) {
        ArrayList<Employee> employees = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        // System.out.println("AKSHITA SHARMA 22BCS15804");

        while (true) {
            System.out.println("1. Add 2. Update 3. Remove 4. Search 5. Display 6.
Exit");
            int choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter ID, Name, Salary: ");
                    employees.add(new Employee(sc.nextInt(), sc.next(), sc.nextDouble()));
                    break;
                case 2:
                    System.out.print("Enter ID to update: ");
                    int uid = sc.nextInt();
```

```
                for (Employee emp : employees) {
                    if (emp.id == uid) {
                        System.out.print("Enter new Name & Salary: ");
                        emp.name = sc.next();
                        emp.salary = sc.nextDouble();
                    }
                }
                break;
            case 3:
                System.out.print("Enter ID to remove: ");
                int rid = sc.nextInt();
                employees.removeIf(emp -> emp.id == rid);
                break;
            case 4:
                System.out.print("Enter ID to search: ");
                int sid = sc.nextInt();
                employees.stream().filter(emp -> emp.id ==
sid).forEach(System.out::println);
                break;
            case 5:
                employees.forEach(System.out::println);
                break;
            case 6:
                sc.close();
                return;
        }
    }
```

```
        }
    }
```

4. Output:

Add Employee:

```
PS C:\Users\harsh\OneDrive\Documents\Java Sem 6> cd "c:\Users
er }
1. Add 2. Update 3. Remove 4. Search 5. Display 6. Exit
1
Enter ID, Name, Salary: 15804
Akshita
1200000
1. Add 2. Update 3. Remove 4. Search 5. Display 6. Exit
5
15804 - Akshita - 1200000.0
1. Add 2. Update 3. Remove 4. Search 5. Display 6. Exit
```

Update Employee:

```
Enter ID to update: 15805
1. Add 2. Update 3. Remove 4. Search 5. Display 6. Exit
5
```

5. Learning Outcomes:

- Learn how to handle the concept of Multithreading.
- Implementing CRUD operations.
- Learn to handle user input from the command line.
- Understand how to store and manage Employee information using arrays.
- Understanding Java Collections.

Medium Level

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:** The objective of this program is to store and retrieve playing cards based on their symbols (e.g., Hearts, Spades, Diamonds, Clubs) using the Collection framework in Java.

3. **Implementation/Code:**

```
import java.util.*;
public class Card {
    public static void main(String[] args) {
        Map<String, List<String>> cards = new HashMap<>();
        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.println("1. Add Card 2. Find by Symbol 3. Display All 4. Exit");
            int choice = sc.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter Symbol and Card Name: ");
                    String symbol = sc.next();
                    String name = sc.next();
```

```
                cards.putIfAbsent(symbol, new ArrayList<>());
                cards.get(symbol).add(name);
                break;
            case 2:
                System.out.print("Enter Symbol to search: ");
             System.out.println(cards.getOrDefault(sc.next(), new ArrayList<>()));
                break;
            case 3:
             cards.forEach((key, value) -> System.out.println(key + " -> " + value));
                break;
            case 4:
                sc.close();
                return;
        }
    }
}}
```

4. Output:

```
1. Add Card 2. Find by Symbol 3. Display All 4. Exit
1
Enter Symbol and Card Name: Heart
Ace
1. Add Card 2. Find by Symbol 3. Display All 4. Exit
3
Heart -> [Ace]
1. Add Card 2. Find by Symbol 3. Display All 4. Exit
Ready
```

5. Learning Outcomes:

- Understanding Java collections.
- Implement key-value storage for categorizing playing cards.
- Add and retrieve elements dynamically without predefined limits.
- Use Scanner to take user input and process it efficiently.

Hard Level

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:** The objective of this Java program is to simulate a ticket booking system where multiple users (threads) attempt to book seats concurrently.

3. **Implementation/Code:**

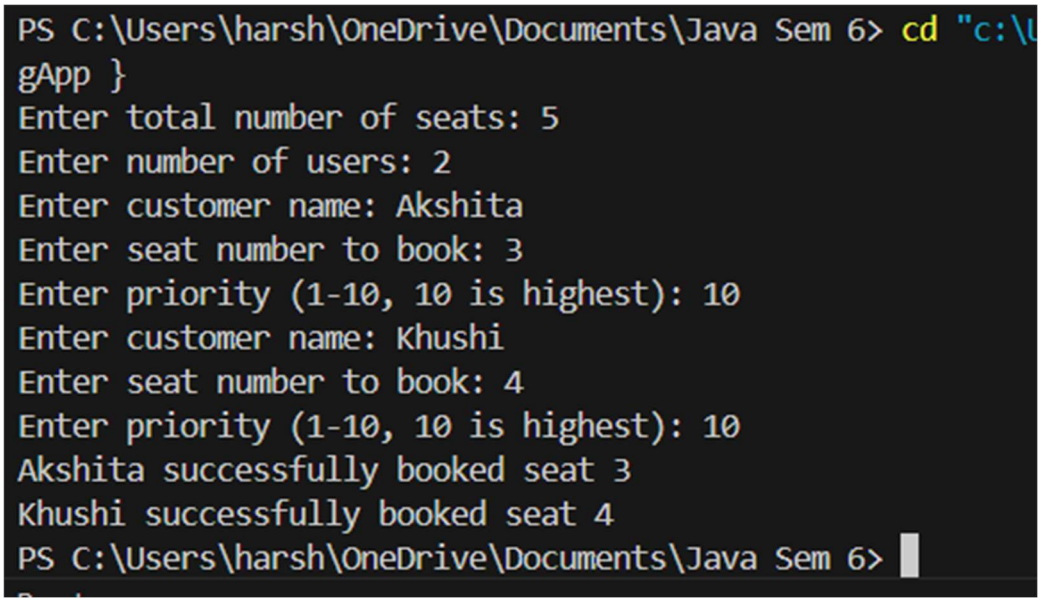
```
import java.util.Scanner;
import java.util.concurrent.locks.*;
class TicketBookingSystem {
    private final boolean[] seats;
    private final Lock lock = new ReentrantLock();
    public TicketBookingSystem(int totalSeats) {
        seats = new boolean[totalSeats];
    }
    public void bookSeat(int seatNumber, String customer) {
        lock.lock();
        try {
```

```
            if (!seats[seatNumber]) {
                seats[seatNumber] = true;
                System.out.println(customer + " successfully booked seat " + seatNumber);
            } else {
                System.out.println(customer + " failed to book seat " + seatNumber + " (Already booked)");
            }
        } finally {
            lock.unlock();
        }
    }
}
class BookingThread extends Thread {
    private final TicketBookingSystem system;
    private final int seatNumber;
    private final String customer;
    public BookingThread(TicketBookingSystem system, int seatNumber, String customer, int priority) {
        this.system = system;
```

```
        this.seatNumber = seatNumber;
        this.customer = customer;
        setPriority(priority);
    }
    @Override
    public void run() {
        system.bookSeat(seatNumber, customer);
    }
}
public class TicketBookingApp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter total number of seats: ");
        int totalSeats = scanner.nextInt();
        TicketBookingSystem system = new TicketBookingSystem(totalSeats);
        System.out.print("Enter number of users: ");
        int userCount = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        Thread[] users = new Thread[userCount];
```

```
        for (int i = 0; i < userCount; i++) {
            System.out.print("Enter customer name: ");
            String customer = scanner.nextLine();
            System.out.print("Enter seat number to book: ");
            int seatNumber = scanner.nextInt();
            System.out.print("Enter priority (1-10, 10 is highest): ");
            int priority = scanner.nextInt();
            scanner.nextLine(); // Consume newline
            users[i] = new BookingThread(system, seatNumber, customer, priority);
        }
        for (Thread user : users) {
            user.start();
        }
        scanner.close();
    }
}
```

4. Output:



5. Learning Outcomes:

- How multiple threads interact when accessing shared data.
- Handling race conditions in a multi-threaded environment.
- Ensuring that only one thread modifies the shared resource (seats) at a time.
- Taking user input for dynamic seat selection and priority assignment.