**Experiment 4.1**

|  |  |
| --- | --- |
| **Student Name: Mayank sharma** | **UID: 22BCS16224** |
| **Branch: CSE** | **Section/Group:22BCS\_IOT-640/A** |
| **Semester: 6th** | **Date of Performance:17/02/25** |
| **Subject Name: 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:

The objective of this Java program is to implement an **ArrayList** to store and manage employee details, including **Employee ID, Name, and Salary**. The program will provide functionalities for users to:

* 1. **Add** new employees.
  2. **Update** existing employee details.
  3. **Remove** an employee by ID.
  4. **Search** for an employee by ID or Name.
  5. **Display** all employees.

This program demonstrates the use of **ArrayList**, **OOP principles (Encapsulation and Classes)**, and **basic CRUD operations** in Java.

1. **Implementation/Code:** import java.util.ArrayList; import java.util.Scanner;

public class EmployeeManagement {

static ArrayList<String> employees = new ArrayList<>(); static Scanner scanner = new Scanner(System.in);

static void addEmployee() {

System.out.print("Enter Employee Details (ID Name Salary): "); employees.add(scanner.nextLine());

}

static void displayEmployees() { if (employees.isEmpty()) {

System.out.println("No employees found.");

} else {

for (String emp : employees) { System.out.println(emp);

}

}

}

static void removeEmployee() { System.out.print("Enter Employee ID to remove: ");

String id = scanner.next();

employees.removeIf(emp -> emp.startsWith(id));

}

static void searchEmployee() { System.out.print("Enter Employee ID to search: "); String id = scanner.next();

for (String emp : employees) {

if (emp.startsWith(id)) { System.out.println(emp); return;

}

}

System.out.println("Employee not found.");

}

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

System.out.print("\n1.Add 2.Remove 3.Search 4.Display 0.Exit: "); int choice = scanner.nextInt(); scanner.nextLine(); // Consume newline

if (choice == 1) addEmployee(); else if (choice == 2) removeEmployee(); else if (choice == 3) searchEmployee(); else if (choice == 4) displayEmployees(); else if (choice == 0) break;

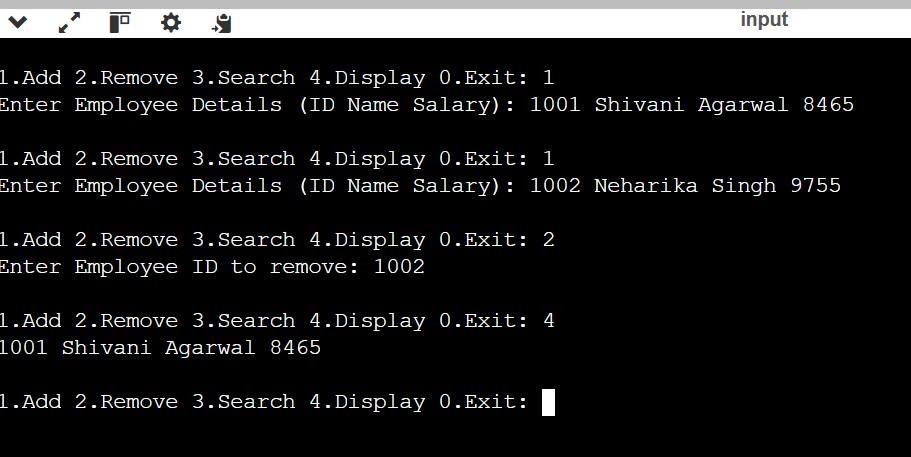
else System.out.println("Invalid choice.");

}

}

}

# Output:

****

**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.

# Objective :

The objective of this program is to design a system that collects and stores all cards using the

**Collection interface** in Java. The program will allow users to efficiently search and retrieve all cards associated with a given symbol.

By implementing this, users can:

* + 1. **Store** different types of cards in a collection.
    2. **Retrieve** all cards corresponding to a given symbol.
    3. **Utilize** Java’s Collection framework for efficient storage and retrieval.
    4. **Enhance** usability by providing a structured way to manage and search cards.
  1. **Implementation /Code :** import java.util.\*;

// Card class to represent a playing card class Card { private String symbol;

private String value;

public Card(String symbol, String value) { this.symbol = symbol; this.value = value;

}

public String getSymbol() { return symbol;

}

public String getValue() { return value;

}

@Override public String toString() {

return value + " of " + symbol;

}

}

// CardCollection class to manage and store cards class CardCollection {

private Map<String, Collection<Card>> cardMap;

public CardCollection() { cardMap = new HashMap<>();

}

public void addCard(Card card) { cardMap.putIfAbsent(card.getSymbol(), new ArrayList<>());

cardMap.get(card.getSymbol()).add(card);

}

public Collection<Card> getCardsBySymbol(String symbol) { return cardMap.getOrDefault(symbol, Collections.emptyList());

}

public void displayAllCards() {

for (Map.Entry<String, Collection<Card>> entry : cardMap.entrySet()) { System.out.println("Symbol: " + entry.getKey());

for (Card card : entry.getValue()) { System.out.println(" " + card);

}

}

}

}

public class CardManager {

public static void main(String[] args) { CardCollection collection = new CardCollection();

collection.addCard(new Card("Hearts", "Ace")); collection.addCard(new Card("Hearts", "King")); collection.addCard(new Card("Spades", "Queen")); collection.addCard(new Card("Diamonds", "Jack")); collection.addCard(new Card("Clubs", "10"));

Scanner scanner = new Scanner(System.in);

System.out.print("Enter symbol to find cards (e.g., Hearts, Spades): "); String symbol = scanner.nextLine();

Collection<Card> foundCards = collection.getCardsBySymbol(symbol); if (foundCards.isEmpty()) {

System.out.println("No cards found for symbol: " + symbol);

} else {

System.out.println("Cards found for " + symbol + ":"); for (Card card : foundCards) { System.out.println(" " + card);

}

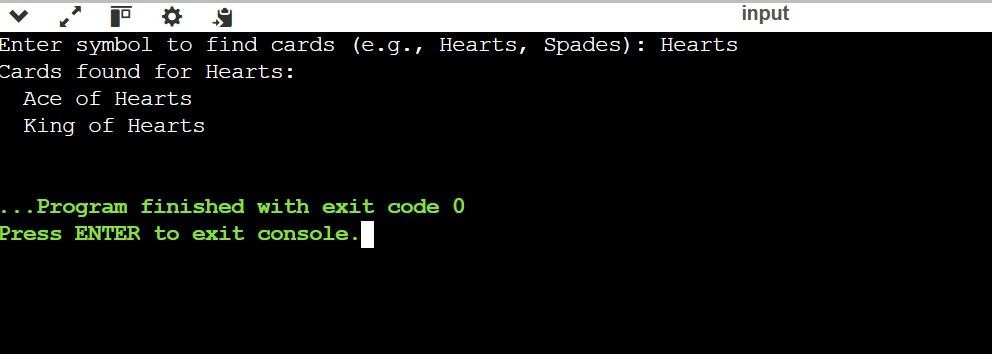
}

scanner.close();

}

}

# Output :

****

**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.

# Objective :

Prevent Double Booking – Use thread synchronization to avoid multiple bookings for the same seat.

 VIP Priority – Assign higher thread priority to VIP users for faster processing.  Efficient Seat Allocation – Dynamically update seat availability in real time.

 Concurrency Handling – Manage multiple booking requests safely using locks.  Transaction Integrity – Ensure accurate booking records and handle failures gracefully.

 Scalability – Optimize for high-volume concurrent bookings.

 Logging & Monitoring – Track transactions and detect system issues.

# Implementation /Code :

class TicketBookingSystem {

private int availableSeats = 5; // Total seats available

public synchronized boolean bookSeat(String name) { if (availableSeats > 0) {

System.out.println(name + " booked seat " + availableSeats); availableSeats--; return true; } else {

System.out.println(name + " booking failed. No seats available."); return false;

}

}

}

class BookingThread extends Thread { private TicketBookingSystem system; private String customerName;

public BookingThread(TicketBookingSystem system, String customerName, int priority) {

this.system = system; this.customerName = customerName;

setPriority(priority);

}

@Override public void run() {

system.bookSeat(customerName);

}

}

public class TicketBookingApp { public static void main(String[] args) {

TicketBookingSystem system = new TicketBookingSystem();

BookingThread vip1 = new BookingThread(system, "VIP1", Thread.MAX\_PRIORITY);

BookingThread vip2 = new BookingThread(system, "VIP2", Thread.MAX\_PRIORITY);

BookingThread normal1 = new BookingThread(system, "User1", Thread.NORM\_PRIORITY);

BookingThread normal2 = new BookingThread(system, "User2", Thread.NORM\_PRIORITY);

BookingThread normal3 = new BookingThread(system, "User3", Thread.NORM\_PRIORITY);

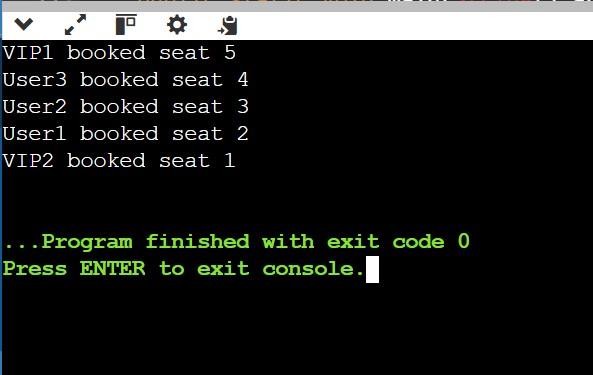
vip1.start();

vip2.start(); normal1.start(); normal2.start(); normal3.start();

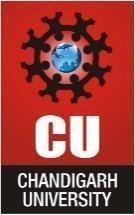
}

}

# Output :



1. **Learning Outcomes :**
   * Learn about Thread Synchronization.
   * Learn about Arraylist.
   * Implementation of Arraylist.



DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING