# Experiment 4,1

**Student Name: Sagar Chandel UID: 22BCS16655**

**Branch: CSE Section: 22BCS\_IOT-642-A**

**Semester: 6th DOP:25/02/25**

**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:** The objective of this Java program is to implement an **ArrayList** to manage employee details, including ID**,** Name, and Salary.

# Code:

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

class Employee { private int id; private String name;

private double salary;

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

this.name = name; this.salary = salary;

}

public int getId() { return id;

}

public String getName() { return name;

}

public double getSalary() { return salary;

}

public void setName(String name) { this.name = name;

}

public void setSalary(double salary) { this.salary = salary;

}

public String getDetails() {

return "Employee ID: " + id + ", Name: " + name + ", Salary: " + salary;

}

}

public class EmployeeManagement {

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

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

System.out.println("\nEmployee Management System"); System.out.println("1. Add Employee data"); System.out.println("2. Update Employee data"); System.out.println("3. Remove Employee data"); System.out.println("4. Search Employee data"); System.out.println("5. Display All Employees data"); System.out.println("6. Exit");

System.out.print("Enter your choice: "); int choice = scanner.nextInt(); scanner.nextLine();

switch (choice) { case 1:

addEmployee(); break;

case 2:

updateEmployee(); break;

case 3:

removeEmployee(); break;

case 4:

searchEmployee(); break;

case 5:

displayEmployees(); break;

case 6:

System.out.println("Exiting..."); return;

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

private static void addEmployee() { 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();

employees.add(new Employee(id, name, salary)); System.out.println("Employee added successfully!");

}

private static void updateEmployee() { System.out.print("Enter Employee ID to update: "); int id = scanner.nextInt();

scanner.nextLine();

for (Employee emp : employees) { if (emp.getId() == id) {

System.out.print("Enter new Name: "); String name = scanner.nextLine(); System.out.print("Enter new Salary: "); double salary = scanner.nextDouble(); emp.setName(name); emp.setSalary(salary);

System.out.println("Employee updated successfully!"); return;

}

}

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

}

private static void removeEmployee() { System.out.print("Enter Employee ID to remove: "); int id = scanner.nextInt();

scanner.nextLine();

employees.removeIf(emp -> emp.getId() == id); System.out.println("Employee removed successfully!");

}

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

scanner.nextLine();

for (Employee emp : employees) { if (emp.getId() == id) {

System.out.println(emp.getDetails()); return;

}

}

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

}

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

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

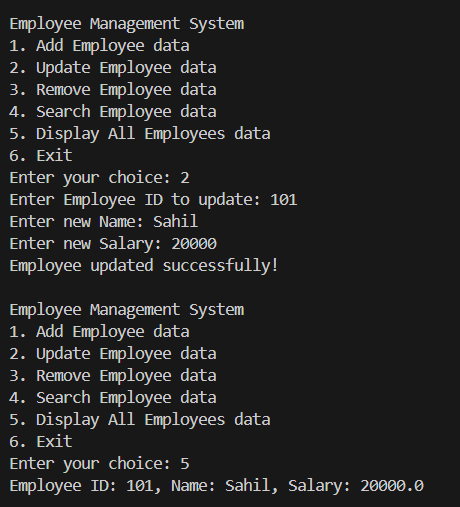
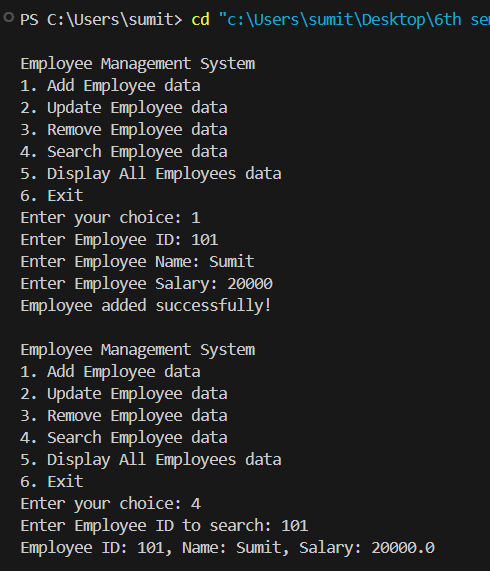
}

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

}

}}

# Output:



**Learning Outcomes:**

* Adding, updating, removing, and searching elements dynamically.
* Creating and managing objects using a class.
* Implementing switch-case and loops for menu-driven execution.
* Implementing switch-case and loops for menu-driven execution.
* Storing and retrieving structured employee data efficiently.

# 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:** The objective is to serialize and deserialize a Student object, store and retrieve its id, name, and GPA from a file, and handle exceptions like FileNotFoundException, IOException, and ClassNotFoundException.
3. **Implementation Code:**

import java.util.\*; class Card {

String symbol;

String name;

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

this.name = name;

}

public String getSymbol() { return symbol;

}

public String getName() { return name;

}

}

public class CardCollectionManager {

static Collection<Card> cards = new ArrayList<>(); static Scanner scanner = new Scanner(System.in);

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

System.out.println("\nCard Collection Manager"); System.out.println("1. Add Card"); System.out.println("2. Find Cards by Symbol"); System.out.println("3. Display All Cards"); System.out.println("4. Remove Card"); System.out.println("5. Count Cards by Symbol"); System.out.println("6. Check if a Card Exists"); System.out.println("7. Exit"); System.out.print("Enter your choice: ");

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

switch (choice) { case 1:

addCard(); break;

case 2:

findCardsBySymbol(); break;

case 3:

displayAllCards(); break;

case 4:

removeCard(); break;

case 5:

countCardsBySymbol(); break;

case 6:

checkCardExists(); break;

case 7:

System.out.println("Exiting..."); return;

default:

System.out.println("Invalid choice. Try again.");

}

}

}

static void addCard() { System.out.print("Enter Card Symbol: "); String symbol = scanner.nextLine(); System.out.print("Enter Card Number: "); String name = scanner.nextLine(); cards.add(new Card(symbol, name));

System.out.println("Card added successfully!");

}

static void findCardsBySymbol() { System.out.print("Enter symbol to search: "); String symbol = scanner.nextLine();

boolean found = false; for (Card card : cards) {

if (card.getSymbol().equalsIgnoreCase(symbol)) { System.out.println("Card Name: " + card.getName()); found = true;

}

}

if (!found) {

System.out.println("No cards found for the given symbol.");

}

}

static void displayAllCards() { if (cards.isEmpty()) {

System.out.println("No cards in the collection.");

return;

}

for (Card card : cards) {

System.out.println("Symbol: " + card.getSymbol() + ", Name: " + card.getName());

}

}

static void removeCard() {

System.out.print("Enter Card Name to remove: "); String name = scanner.nextLine();

boolean removed = cards.removeIf(card -> card.getName().equalsIgnoreCase(name)); if (removed) {

System.out.println("Card removed successfully!");

} else {

System.out.println("Card not found!");

}

}

static void countCardsBySymbol() { System.out.print("Enter symbol to count: "); String symbol = scanner.nextLine();

long count = cards.stream().filter(card -> card.getSymbol().equalsIgnoreCase(symbol)).count();

System.out.println("Total cards with symbol '" + symbol + "': " + count);

}

static void checkCardExists() { System.out.print("Enter Card Name to check: "); String name = scanner.nextLine();

boolean exists = cards.stream().anyMatch(card -> card.getName().equalsIgnoreCase(name)); if (exists) {

System.out.println("Card exists in the collection.");

} else {

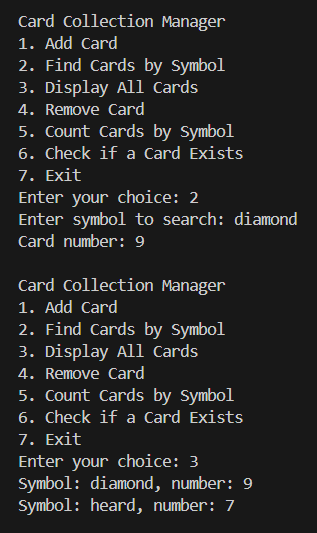
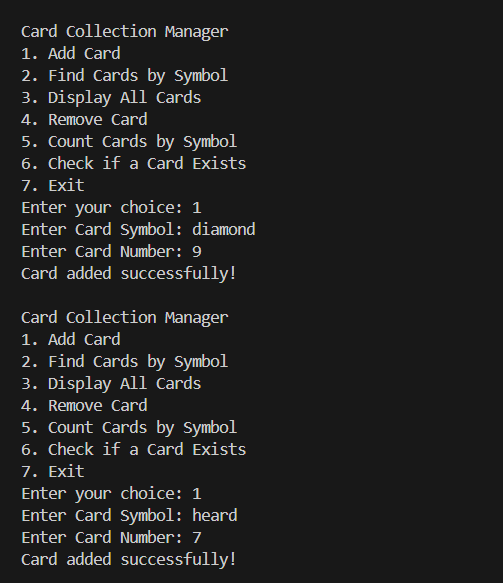
System.out.println("Card not found.");

}

}

}

1. **Output**



1. **Learning Outcomes:**
   * Adding, updating, removing, and searching elements dynamically.
   * Creating and managing objects using a class.
   * Implementing switch-case and loops for menu-driven execution.
   * Implementing switch-case and loops for menu-driven execution.
   * Storing and retrieving structured employee data efficiently.

# 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:**
       - Use synchronized methods to prevent race conditions.
       - Use thread priorities to handle VIP bookings before general bookings.
       - Ensure fairness in seat allocation using Thread.sleep() to simulate real-world delays.
    3. **Implementation Code:**

import java.util.\*;

class TicketBookingSystem { int totalSeats;

boolean[] seats;

TicketBookingSystem(int totalSeats) { this.totalSeats = totalSeats; this.seats = new boolean[totalSeats];

}

synchronized boolean bookSeat(int seatNumber, String customerName) { if (seatNumber < 0 || seatNumber >= totalSeats) {

System.out.println(customerName + " - Invalid seat number!"); return false;

}

if (!seats[seatNumber]) { seats[seatNumber] = true;

System.out.println(customerName + " successfully booked seat " + seatNumber); return true;

} else {

System.out.println(customerName + " - Seat " + seatNumber + " is already booked!"); return false;

}

}

}

class TicketBookingThread extends Thread { TicketBookingSystem system;

int seatNumber; String customerName;

TicketBookingThread(TicketBookingSystem system, int seatNumber, String customerName, int priority) {

this.system = system; this.seatNumber = seatNumber; this.customerName = customerName; setPriority(priority);

}

public void run() {

try {

Thread.sleep(100); system.bookSeat(seatNumber, customerName);

} catch (InterruptedException e) { e.printStackTrace();

}

}

}

public class TicketBookingApp {

public static void main(String[] args) {

TicketBookingSystem system = new TicketBookingSystem(10); TicketBookingThread vip1 = new TicketBookingThread(system, 3, "VIP-1",

Thread.MAX\_PRIORITY);

TicketBookingThread vip2 = new TicketBookingThread(system, 2, "VIP-2", Thread.MAX\_PRIORITY);

TicketBookingThread normal1 = new TicketBookingThread(system, 3, "User-1", Thread.MIN\_PRIORITY);

TicketBookingThread normal2 = new TicketBookingThread(system, 2, "User-2", Thread.MIN\_PRIORITY);

TicketBookingThread normal3 = new TicketBookingThread(system, 5, "User-3", Thread.NORM\_PRIORITY);

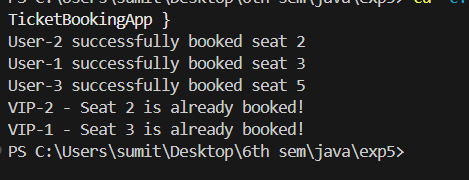
vip1.start();

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

}

}

* + 1. **Output:**



* + 1. **Learning Outcomes:**
       - Understanding Multi-threading: Implemented parallel execution using the Thread class to handle multiple booking requests simultaneously.
       - Synchronization for Data Safety: Used synchronized methods to ensure that no two threads can book the same seat at the same time, preventing race conditions.
       - Thread Priorities & Scheduling: Assigned Thread.MAX\_PRIORITY to VIP bookings to ensure they are processed before normal users.
       - Concurrency Management: Demonstrated how multiple users can attempt bookings simultaneously without causing data inconsistencies.
       - Seat Availability Control: Checked and updated seat booking status in a thread-safe manner to prevent double bookings.