Experiment 4.3

Student Name: Divyanshi **UID:** 22BCS12482
Branch: BE-CSE **Section/Group:** 22BCS-IOT-637(A)
Semester: 6th **Date of Performance:** 20/02/25
Subject Name: JAVA (PBLJ) **Subject Code:** 22CSH-359

A) EASY:

1) 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.

2) Objective:

The objective of this Java program is to implement an **Employee Management System** using an `ArrayList` to store employee details such as **ID, Name, and Salary**. The program should allow users to:

1. **Add** new employees.
2. **Update** existing employee details.
3. **Remove** employees from the list.
4. **Search** for employees based on their ID or Name.
5. **Display** the complete list of employees.

This program demonstrates the use of **ArrayList**, **OOP concepts**, and **basic CRUD (Create, Read, Update, Delete) operations** in Java. It provides a simple yet functional system for managing employee records efficiently.

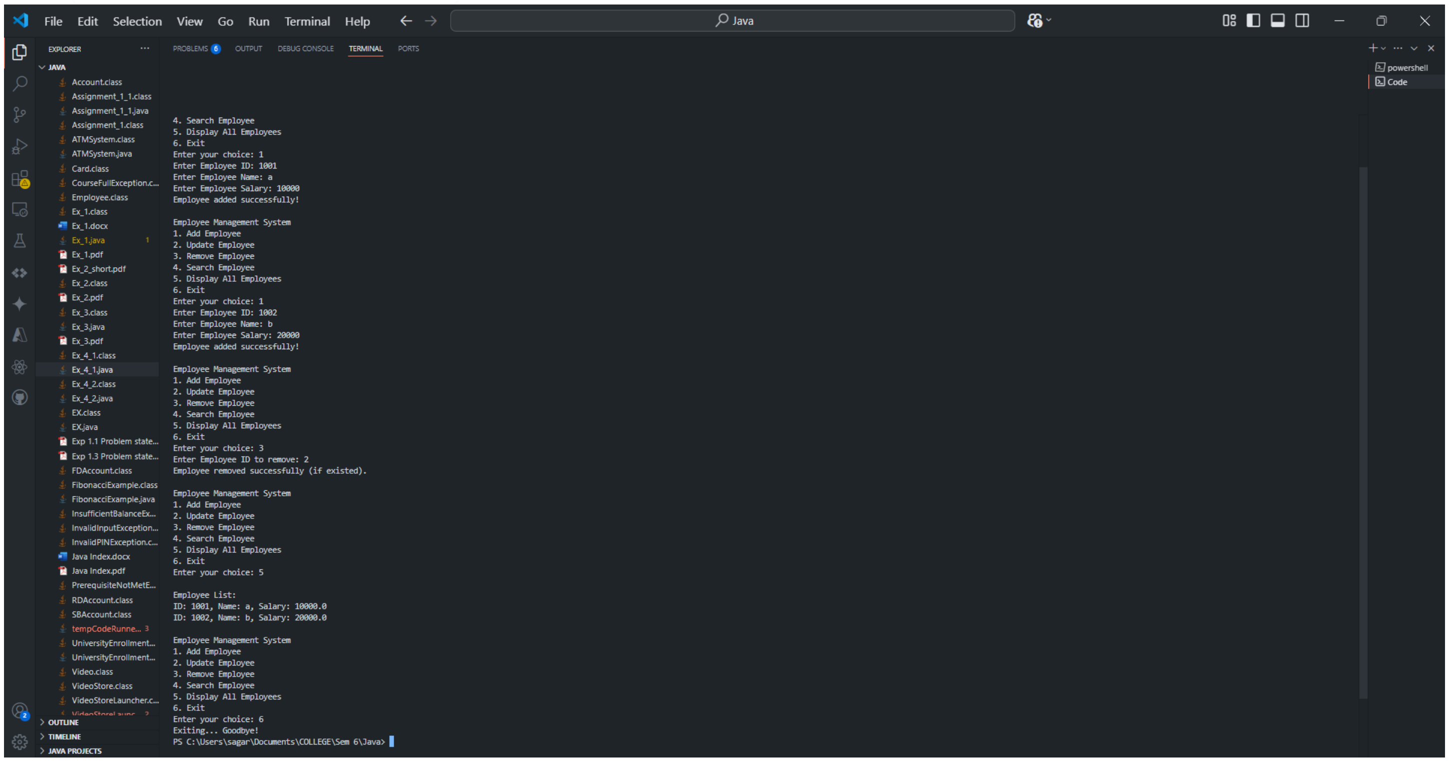
3) Implementation/Code:

```
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;
}

@Override
public String toString() {
    return "ID: " + id + ", Name: " + name + ", Salary: " + salary;
}
```

```
}

public class Ex_4_1 { private static final ArrayList employees = new ArrayList<>();

private static final 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");
        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 -> addEmployee();
            case 2 -> updateEmployee();
            case 3 -> removeEmployee();
            case 4 -> searchEmployee();
            case 5 -> displayEmployees();
            case 6 -> {
                System.out.println("Exiting... Goodbye!");
                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();
    scanner.nextLine();

    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.id == id) {
            System.out.print("Enter new Name: ");
```

```
                emp.name = scanner.nextLine();
                System.out.print("Enter new Salary: ");
                emp.salary = scanner.nextDouble();
                scanner.nextLine();
                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.id == id);
        System.out.println("Employee removed successfully (if existed). ");
    }

    private static void searchEmployee() {
        System.out.print("Enter Employee ID to search: ");
        int id = scanner.nextInt();
        scanner.nextLine();
        for (Employee emp : employees) {
            if (emp.id == id) {
                System.out.println("Employee Found: " + emp);
                return;
            }
        }
        System.out.println("Employee not found.");
    }

    private static void displayEmployees() {
        if (employees.isEmpty()) {
            System.out.println("No employees to display.");
        } else {
            System.out.println("\nEmployee List:");
            for (Employee emp : employees) {
                System.out.println(emp);
            }
        }
    }


    }
```

4) Output:



```
4. Search Employee
5. Display All Employees
6. Exit
Enter your choice: 1
Enter Employee ID: 1001
Enter Employee Name: a
Enter Employee Salary: 10000
Employee added successfully!

Employee Management System
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All Employees
6. Exit
Enter your choice: 1
Enter Employee ID: 1002
Enter Employee Name: b
Enter Employee Salary: 20000
Employee added successfully!

Employee Management System
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All Employees
6. Exit
Enter your choice: 3
Enter Employee ID to remove: 2
Employee removed successfully (if existed).

Employee Management System
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All Employees
6. Exit
Enter your choice: 5

Employee List:
ID: 1001, Name: a, Salary: 10000.0
ID: 1002, Name: b, Salary: 20000.0

Employee Management System
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All Employees
6. Exit
Enter your choice: 6
Exiting... Goodbye!
PS C:\Users\sagar\Documents\COLLEGE\Sem 6\Java>
```

5) Learning Outcomes:

- **Understanding ArrayList** – Learn how to store and manage dynamic collections of objects in Java.
- **Implementing CRUD Operations** – Gain hands-on experience with adding, updating, removing, and searching records.
- **Object-Oriented Programming (OOP)** – Apply encapsulation and object manipulation using classes and objects.
- **User Input Handling** – Improve skills in taking and processing user inputs using `Scanner`.
- **Practical Java Application** – Develop a real-world console-based employee management system.

B) MEDIUM

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

2. Objective:

The objective of this Java program is to implement a **Card Collection System** using the **Collection interface** to store and manage different playing cards. The program should allow users to:

1. **Add** new cards to the collection.
2. **Search** for all cards belonging to a specific symbol (e.g., Hearts, Diamonds).
3. **Display** the complete list of stored cards.
4. **Efficiently manage** card storage and retrieval using Java Collections.

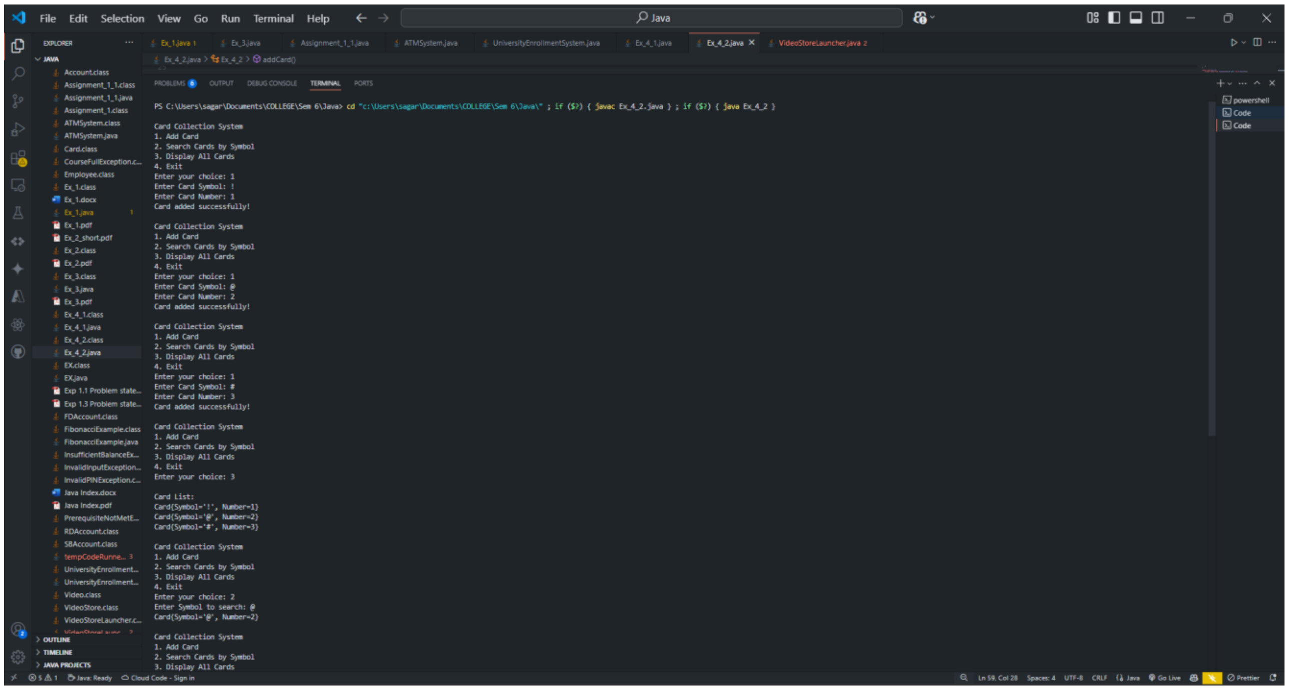
This program demonstrates the use of **Collection interface and its implementations**, **dynamic data handling**, and **efficient searching techniques** to assist users in organizing and retrieving cards easily.

3. Implementation:

```
import java.util.*;

class Card { private String symbol; private int number;

public Card(String symbol, int number) {
    this.symbol = symbol;
    this.number = number;
}

public String getSymbol() {
    return symbol;
}

public int getNumber() {
    return number;
}

@Override
public String toString() {
    return "Card{Symbol='" + symbol + "', Number=" + number + "}";
}


}

public class Ex_4_2 { private static final Collection cards = new ArrayList<>();

private static final Scanner scanner = new Scanner(System.in);

public static void main(String[] args) {
    while (true) {
        System.out.println("\nCard Collection System");
```

```
            System.out.println("1. Add Card");
            System.out.println("2. Search Cards by Symbol");
            System.out.println("3. Display All Cards");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            switch (choice) {
                case 1 -> addCard();
                case 2 -> searchCardsBySymbol();
                case 3 -> displayCards();
                case 4 -> {
                    System.out.println("Exiting... Goodbye!");
                    return;
                }
                default -> System.out.println("Invalid choice. Please try again.");
            }
        }
    }

    private static void addCard() {
        System.out.print("Enter Card Symbol: ");
        String symbol = scanner.nextLine();
        System.out.print("Enter Card Number: ");
        int number = scanner.nextInt();
        scanner.nextLine();

        cards.add(new Card(symbol, number));
        System.out.println("Card added successfully!");
    }

    private static void searchCardsBySymbol() {
        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);
                found = true;
            }
        }
        if (!found) {
            System.out.println("No cards found with symbol: " + symbol);
        }
    }

    private static void displayCards() {
        if (cards.isEmpty()) {
            System.out.println("No cards to display.");
        } else {
            System.out.println("\nCard List:");
            for (Card card : cards) {
                System.out.println(card);
            }
    }
}


}
```

4. Output:



```
PS C:\Users\sagar\Documents\COLLEGE\Sem 6\Java> cd "c:\Users\sagar\Documents\COLLEGE\Sem 6\Java\" ; if ($?) { javac Ex_4_2.java } ; if ($?) { java Ex_4_2 }

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
4. Exit
Enter your choice: 1
Enter Card Symbol: !
Enter Card Number: 1
Card added successfully!

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
4. Exit
Enter your choice: 1
Enter Card Symbol: @
Enter Card Number: 2
Card added successfully!

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
4. Exit
Enter your choice: 1
Enter Card Symbol: #
Enter Card Number: 3
Card added successfully!

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
4. Exit
Enter your choice: 3

Card List:
Card{Symbol='!', Number=1}
Card{Symbol='@', Number=2}
Card{Symbol='#', Number=3}

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
4. Exit
Enter your choice: 2
Enter Symbol to search: @
Card{Symbol='@', Number=2}

Card Collection System
1. Add Card
2. Search Cards by Symbol
3. Display All Cards
```

5. Learning Outcome:

- **Understanding Java Collections** – Learn how to use the `Collection` interface for dynamic data storage.
- **Efficient Data Management** – Implement a system to store, retrieve, and search for cards based on symbols.
- **Practical Use of Interfaces** – Gain experience in using the `Collection` interface and its implementations.
- **Implementing Searching Mechanisms** – Develop skills to filter and retrieve specific data from a collection.
- **Hands-on OOP Concepts** – Apply encapsulation and object manipulation while managing card data.

C) HARD

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 develop a **multithreaded Ticket Booking System** that ensures:

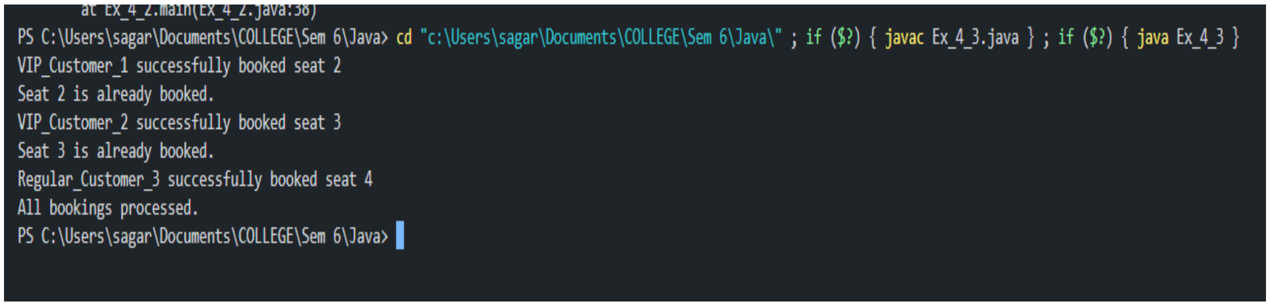
1. **Thread Synchronization** – Prevents double booking of seats by using synchronized methods.
2. **VIP Booking Priority** – Utilizes thread priorities to process VIP bookings before regular users.
3. **Concurrent Seat Allocation** – Allows multiple users to book tickets simultaneously in a controlled manner.
4. **Efficient Resource Management** – Ensures fair seat distribution while avoiding conflicts.
5. **Real-World Simulation** – Demonstrates the use of multithreading in ticket booking applications.

3. **Implementation:**

```
import java.util.concurrent.locks.;

import java.util.;

class TicketBookingSystem { private static final int TOTAL_SEATS = 10; private final boolean[] seats = new
boolean[TOTAL_SEATS]; private final Lock lock = new ReentrantLock();

public void bookSeat(int seatNumber, String customerName) {
    lock.lock();
    try {
        if (seatNumber < 0 || seatNumber >= TOTAL_SEATS) {
            System.out.println("Invalid seat number");
            return;
        }
        if (!seats[seatNumber]) {
            seats[seatNumber] = true;
            System.out.println(customerName + " successfully booked seat " + seatNumber);
        } else {
            System.out.println("Seat " + seatNumber + " is already booked.");
        }
    } finally {
        lock.unlock();
    }
}


}

class BookingThread extends Thread { private final TicketBookingSystem system;

private final int seatNumber; private final String customerName;

public BookingThread(TicketBookingSystem system, int seatNumber, String customerName, int priority) {
    this.system = system;
    this.seatNumber = seatNumber;
    this.customerName = customerName;
```

```
        setPriority(priority);
    }

    @Override
    public void run() {
        system.bookSeat(seatNumber, customerName);
    }


    }

    public class Ex_4_3 { public static void main(String[] args) { TicketBookingSystem system = new
    TicketBookingSystem(); List threads = new ArrayList<>();

       threads.add(new BookingThread(system, 2, "VIP_Customer_1", Thread.MAX_PRIORITY));
        threads.add(new BookingThread(system, 2, "Regular_Customer_1", Thread.MIN_PRIORITY));
        threads.add(new BookingThread(system, 3, "VIP_Customer_2", Thread.MAX_PRIORITY));
        threads.add(new BookingThread(system, 3, "Regular_Customer_2", Thread.NORM_PRIORITY));
        threads.add(new BookingThread(system, 4, "Regular_Customer_3", Thread.NORM_PRIORITY));

        for (Thread thread : threads) {
            thread.start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("All bookings processed.");
    }


    }
```

4. Output:



```
        at Ex_4_2.main(Ex_4_2.java:38)
PS C:\Users\sagar\Documents\COLLEGE\Sem 6\Java> cd "c:\Users\sagar\Documents\COLLEGE\Sem 6\Java\" ; if ($?) { javac Ex_4_3.java } ; if ($?) { java Ex_4_3 }
VIP_Customer_1 successfully booked seat 2
Seat 2 is already booked.
VIP_Customer_2 successfully booked seat 3
Seat 3 is already booked.
Regular_Customer_3 successfully booked seat 4
All bookings processed.
PS C:\Users\sagar\Documents\COLLEGE\Sem 6\Java>
```

5. Learning Outcome:

1. **Thread Synchronization** – Learn how to prevent race conditions using synchronized methods in Java.
2. **Thread Prioritization** – Understand how to set and manage thread priorities for VIP and regular bookings.
3. **Concurrency Handling** – Gain experience in managing multiple threads accessing shared resources.
4. **Efficient Seat Allocation** – Develop skills in designing a fair and conflict-free ticket booking system.
5. **Real-World Multithreading** – Apply Java multithreading concepts to solve practical scheduling problems.