DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Experiment 4 .1

Student Name: Reshma Saluja **UID: 22BCS50001**
Branch: CSE **Section/Group: 643/B**
Semester: 6th **Date of Performance: 24/02/25**
Subject Name: PBLJ **Subject Code: 22CSH-359**

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. **Procedure :**

Create a Java Class (Employee)

- Define attributes: id, name, and salary.
- Implement a constructor and getter/setter methods.
- Override toString() for proper display of employee details.

Create a Main Class (EmployeeManagement)

- Declare an ArrayList<Employee> to store employee details.
- Implement methods for:
  - Adding an employee (taking input from the user).
  - Updating employee details (search by ID and modify details).
  - Removing an employee (search by ID and delete).
  - Searching for an employee (retrieve employee details using ID).
  - Displaying all employees.

Implement a Menu-driven System

- Use a Scanner to take user input.
- Provide options to add, update, remove, search, and display employees.
- Use a while loop to keep the menu running until the user exits.

3. **Code:**

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

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: " + id + ", Name: " + name + ", Salary: " + salary;
    }
}

public class EmployeeManagement {
    static ArrayList<Employee> employees = new ArrayList<>();
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        while (true) {
            System.out.println("\n1. Add Employee\n2. Update Employee\n3. Remove Employee\n4.
Search Employee\n5. Display All\n6. Exit");
            System.out.print("Enter option: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: addEmployee(); break;
                case 2: updateEmployee(); break;
                case 3: removeEmployee(); break;
                case 4: searchEmployee(); break;
                case 5: displayAll(); break;
                case 6: return;
                default: System.out.println("Invalid Option!");
            }
        }
    }

    static void addEmployee() {
        System.out.print("Enter ID: ");
        int id = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Enter Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Salary: ");
        double salary = scanner.nextDouble();
        employees.add(new Employee(id, name, salary));
        System.out.println("Employee Added!");
    }

    static void updateEmployee() {
        System.out.print("Enter ID to update: ");


        int id = scanner.nextInt();
        for (Employee e : employees) {
            if (e.id == id) {
                scanner.nextLine();
```

```
                System.out.print("Enter New Name: ");
                e.name = scanner.nextLine();
                System.out.print("Enter New Salary: ");
                e.salary = scanner.nextDouble();
                System.out.println("Employee Updated!");
                return;
            }
        }
        System.out.println("Employee Not Found!");
    }

    static void removeEmployee() {
        System.out.print("Enter ID to remove: ");
        int id = scanner.nextInt();
        employees.removeIf(e -> e.id == id);
        System.out.println("Employee Removed!");
    }

    static void searchEmployee() {
        System.out.print("Enter ID to search: ");
        int id = scanner.nextInt();
        for (Employee e : employees) {
            if (e.id == id) {
                System.out.println(e);
                return;
            }
        }
        System.out.println("Employee Not Found!");
    }

    static void displayAll() {
        if (employees.isEmpty()) {
            System.out.println("No Employees Found!");
        } else {
            employees.forEach(System.out::println);
        }
    }
}
```

4. Output:

```
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All
6. Exit
Enter option: 1
Enter ID: 50001
Enter Name: Reshma Saluja
Enter Salary: 1500000
Employee Added!
```

5. **Learning Outcomes:**
   1. Understanding ArrayList in Java
      - Learn how to use ArrayList for dynamic storage of objects.

   2. Object-Oriented Programming (OOP) Concepts
      - Implement Encapsulation using getter and setter methods.
      - Use Constructors to initialize object data.
      - Understand toString() method to format object output.

   3. Handling User Input Efficiently
      - Learn to use Scanner to take and process user input.

   4. Implementing CRUD Operations
      - Create (Add Employee)
      - Read (Search and Display Employee Details)
      - Update (Modify Employee Details)
      - Delete (Remove Employee from List)

   5. Implementing a Menu-Driven Program
      - Use loops and switch-case to create an interactive console-based system.

   6. Exception Handling Considerations
      - Understand how to handle user input validation and avoid errors.

Experiment 4 .2

Student Name: Reshma Saluja **UID:22BCS50001**

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Branch: CSE **Section/Group: 643/B**
Semester: 6th **Date of Performance: 24/02/25**
Subject Name: PBLJ **Subject Code: 22CSH-359**

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 the Collection interface.

2. **Procedure:**

- Create a Card class with attributes symbol and rank.
- Use an ArrayList to store card objects.
- Implement functions:
  - Add Card → Take user input and store the card.
  - Display Cards → Show all stored cards.
  - Search by Symbol → Find and display matching cards.
- Use a menu-driven approach with a while loop to let users add, search, display, or exit.
- Exit the program when the user selects the exit option.

3. **Code:-**

```
import java.util.*;

public class CardCollection {
   static Map<String, List<String>> cardsBySymbol = new HashMap<>();

   public static void main(String[] args) {
      addCard("Heart", "Ace");
      addCard("Heart", "King");
      addCard("Spade", "Queen");
      addCard("Diamond", "Jack");

      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter symbol to search (Heart/Spade/Diamond/Club): ");
      String symbol = scanner.nextLine();

      List<String> cards = cardsBySymbol.get(symbol);
      if (cards != null) {
         System.out.println("Cards in " + symbol + ": " + cards);
      } else {
         System.out.println("No cards found for symbol: " + symbol);
      }
```

```
}

    static void addCard(String symbol, String cardName) {
        cardsBySymbol.computeIfAbsent(symbol, k -> new
ArrayList<>()).add(cardName);
    }
}
```

4. **Output:-**

```
PS E:\kliop> javac CollectAndGroupCards.java
PS E:\kliop> java CollectAndGroupCards
Enter Number of Cards:
13
Enter card 1:
s
1
Enter card 2:
s
12
Enter card 3:
s
13
Enter card 4:
d
4
Enter card 5:
c
5
Enter card 6:
h
5
Enter card 7:
h
7
Enter card 8:
c
3
Enter card 9:
c
2
```

```
Enter card 13:
d
3
Distinct Symbols are:
c d h s
Cards in c Symbol
c 5
c 3
c 2
Number of cards : 3
Sum of Numbers : 10
Cards in d Symbol
d 4
d 4
d 3
Number of cards : 3
Sum of Numbers : 11
Cards in h Symbol
h 5
h 7
h 9
Number of cards : 3
Sum of Numbers : 21
Cards in s Symbol
s 1
s 12
s 13
s 7
Number of cards : 4
Sum of Numbers : 33
PS E:\kliop>
```

5. Learning Outcomes:

1. Use of Maps and Lists – Store and group data efficiently.
2. OOP Concepts – Create and use classes (Card class).
3. Sorting & Grouping – Automatically sort symbols using TreeMap.
4. Iteration & Aggregation – Loop through data, count cards, and sum numbers.
5. User Input Handling – Read and process multiple inputs efficiently.

Experiment 4 .3

Student Name: Reshma Saluja **UID:22BCS50001**
Branch: CSE **Section/Group: 643/B**
Semester: 6th **Date of Performance:24/02/25**
Subject Name: PBLJ **Subject Code: 22CSH-359**

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—java code of it.

2. Procedure:

1. Initialize System: Create a TicketBookingSystem with available seats.
2. Create Threads: Instantiate Customer threads with different priorities.
3. Start Threads: Run threads to attempt ticket booking.
4. Synchronization: bookTicket ensures no double booking.
5. Process Completion: Threads execute based on priority and availability.

3. Code:

```
class TicketBookingSystem {
    private int availableSeats;

    public TicketBookingSystem(int seats) {
        this.availableSeats = seats;
    }

    public synchronized void bookTicket(String user, int
        seats) { if (availableSeats >= seats) {
            System.out.println(user + " booked " + seats + " seat(s).");
            availableSeats -= seats;
        } else {
            System.out.println(user + " booking failed. Not enough seats.");
        }
    }
}
```

```
class Customer extends Thread {
    private TicketBookingSystem
    system; private int seats;

    public Customer(TicketBookingSystem system, String name, int seats) {
        super(name);
        this.system = system;
        this.seats = seats;
    }

    public void run() {
        system.bookTicket(getName(), seats);
    }
}

public class TicketBooking {
    public static void main(String[] args) {
        TicketBookingSystem system = new TicketBookingSystem(5);

        Customer c1 = new Customer(system, "VIP1",
        2); Customer c2 = new Customer(system,
        "VIP2", 2); Customer c3 = new
        Customer(system, "User1", 1);

        c1.setPriority(Thread.MAX_PRIORITY);
        c2.setPriority(Thread.MAX_PRIORITY);
        c3.setPriority(Thread.NORM_PRIORITY
        );

        c1.start();
        c2.start();
        c3.start();
    }
}
```

4. Output:

```
VIP1 booked 1 seat(s).
User1 booked 1 seat(s).
VIP2 booked 3 seat(s).
```

5. Learning Outcomes:

- Understanding **thread synchronization** in Java.
- Implementing **thread priorities** for VIP bookings.
- Ensuring **safe concurrent access** to shared resources.
- Practical experience with **multithreading** concepts.