# Experiment 4

**Student Name: Rajvardhan Singh** **UID: 22BCS11638**
**Branch: BE-CSE** **Section/Group: 903-B**
**Semester: 6th** **Date of Performance:**
**Subject Name: Project based Learning Java** **Subject Code: 22CSH-359**

## Problem :- 1(Easy-Level)

**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:**

- To create a Java program using ArrayList to store and manage employee details.
- To implement operations such as adding, updating, removing, and searching employees based on their ID.
- To enhance understanding of collections in Java, particularly ArrayList.

## 3. Algorithm:

- Define an Employee class with attributes: id, name, and salary.
- Create an ArrayList<Employee> to store employee records.
- Implement functions for:
- **Adding** an employee.
- **Updating** employee details based on the ID.
- **Removing** an employee by ID.
- **Searching** for an employee by ID.
- Provide a menu-driven interface to perform operations

## 4. Implementation :

```
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;
    }

    public String toString() {
        return "ID: " + id + ", Name: " + name + ", Salary: " + salary;
    }
}
public class EmployeeManagement {
    static ArrayList<Employee> employees = new ArrayList<>();
    static Scanner sc = new Scanner(System.in);
    public static void addEmployee() {
        System.out.print("Enter ID: ");
        int id = sc.nextInt();
```

```
        sc.nextLine();  // Consume newline
        System.out.print("Enter Name: ");
        String name = sc.nextLine();
        System.out.print("Enter Salary: ");
        double salary = sc.nextDouble();
        employees.add(new Employee(id, name, salary));
        System.out.println("Employee added successfully!");
    }
    public static void updateEmployee() {
        System.out.print("Enter Employee ID to update: ");
        int id = sc.nextInt();
        for (Employee emp : employees) {
            if (emp.id == id) {
                sc.nextLine();  // Consume newline
                System.out.print("Enter new Name: ");
                emp.name = sc.nextLine();
                System.out.print("Enter new Salary: ");
                emp.salary = sc.nextDouble();
                System.out.println("Employee details updated.");
                return;
            }
        }
```

```
        System.out.println("Employee not found.");
    }
    public static void removeEmployee() {
        System.out.print("Enter Employee ID to remove: ");
        int id = sc.nextInt();
        employees.removeIf(emp -> emp.id == id);
        System.out.println("Employee removed successfully!");
    }
    public static void searchEmployee() {
        System.out.print("Enter Employee ID to search: ");
        int id = sc.nextInt();
        for (Employee emp : employees) {
            if (emp.id == id) {
                System.out.println(emp);
                return;
            }
        }
        System.out.println("Employee not found.");
    }
    public static void main(String[] args) {
        while (true) {
```

```
            System.out.println("\n1. Add Employee\n2. Update Employee\n3. Remove
Employee\n4. Search Employee\n5. Exit");

            System.out.print("Choose an option: ");

            int choice = sc.nextInt();

            switch (choice) {

                case 1: addEmployee(); break;

                case 2: updateEmployee(); break;

                case 3: removeEmployee(); break;

                case 4: searchEmployee(); break;

                case 5: System.out.println("Exiting..."); return;

                default: System.out.println("Invalid choice!");

            }

        }

    }

}
```

**5. Output:**

```
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Exit
Choose an option: 1
Enter ID: 101
Enter Name: John Doe
Enter Salary: 50000
Employee added successfully!

Choose an option: 4
Enter Employee ID to search: 101
ID: 101, Name: John Doe, Salary: 50000.0
```

## Problem:- 2(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:

- To create a system for managing and searching for playing cards.
- To use Java Collections (ArrayList) to store and retrieve cards efficiently.
- To implement a search function that returns all cards of a given symbol.

## 3. Algorithm:

- **Define a** Card **class** with attributes: symbol and value.
- **Use an** ArrayList<Card> to store all cards.
- **Provide methods for:**
  - **Adding a card** to the collection.
  - **Finding all cards** of a given symbol.
- **Display results** in a user-friendly format.

## 4. Implementation/Code:

```
import java.util.*;
class Card {
    private String symbol;
    private int value;

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

```
    public String getSymbol() {
        return symbol;
    }

    public int getValue() {
        return value;
    }

    public String toString() {
        return "Card{Symbol='" + symbol + "', Value=" + value + "}";
    }
}

public class CardCollection {
    private Collection<Card> cards;

    public CardCollection() {
        cards = new ArrayList<>();
    }

    public void addCard(String symbol, int value) {
        cards.add(new Card(symbol, value));
    }

    public List<Card> findCardsBySymbol(String symbol) {
        List<Card> result = new ArrayList<>();
        for (Card card : cards) {
            if (card.getSymbol().equalsIgnoreCase(symbol)) {
```

```
            result.add(card);
        }
    }
    return result;
}

public static void main(String[] args) {
    CardCollection collection = new CardCollection();
    Scanner sc = new Scanner(System.in);

    // Adding some cards
    collection.addCard("Hearts", 10);
    collection.addCard("Spades", 7);
    collection.addCard("Hearts", 2);
    collection.addCard("Diamonds", 5);
    collection.addCard("Spades", 3);

    // User input to search
    System.out.print("Enter a card symbol to search: ");
    String symbol = sc.nextLine();
    List<Card> foundCards = collection.findCardsBySymbol(symbol);

    if (foundCards.isEmpty()) {
        System.out.println("No cards found for symbol: " + symbol);
    } else {
        System.out.println("Cards found:");
        for (Card card : foundCards) {
            System.out.println(card);
```

```
        }
      }
      sc.close();
   }
}
```

### 5. Output:

```
Enter a card symbol to search: Hearts
Cards found:
Card{Symbol='Hearts', Value=10}
Card{Symbol='Hearts', Value=2}
```

## Problem:- 3(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:

- Implement **thread synchronization** to avoid race conditions.
- Use **thread priorities** to ensure VIP customers are served first.
- Maintain a list of available seats and track bookings.

### 3. Algorithm:

- **Create a** TicketBookingSystem **class** to manage available seats.
- **Use synchronization** in the bookSeat() method to prevent double booking.
- **Define a** Customer **thread** where VIP customers get higher priority.

- **Start multiple threads** for both VIP and regular customers.
- **Ensure VIP bookings are processed first** using thread priorities.

## 4.Implementation/Code:

```
import java.util.*;
class TicketBookingSystem {
    private Set<Integer> bookedSeats = new HashSet<>();
    private int totalSeats;

    public TicketBookingSystem(int totalSeats) {
        this.totalSeats = totalSeats;
    }

    public synchronized boolean bookSeat(int seatNumber, String customerName) {
        if (seatNumber < 1 || seatNumber > totalSeats) {
            System.out.println(customerName + " tried to book an invalid seat: " +
seatNumber);
            return false;
        }
        if (bookedSeats.contains(seatNumber)) {
            System.out.println(customerName + " tried to book an already booked seat: " +
seatNumber);
            return false;
        }
        bookedSeats.add(seatNumber);
        System.out.println(customerName + " successfully booked seat: " + seatNumber);
        return true;
    }
```

```
}

class Customer extends Thread {
    private TicketBookingSystem system;
    private int seatNumber;
    private String customerName;

    public Customer(TicketBookingSystem system, int seatNumber, String
customerName, int priority) {
        this.system = system;
        this.seatNumber = seatNumber;
        this.customerName = customerName;
        setPriority(priority); // Set thread priority
    }

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

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

        // Creating customer threads
        Customer vip1 = new Customer(system, 1, "VIP John", Thread.MAX_PRIORITY);
        Customer vip2 = new Customer(system, 2, "VIP Alice", Thread.MAX_PRIORITY);
```

```
        Customer regular1 = new Customer(system, 3, "Regular Bob",
Thread.NORM_PRIORITY);
        Customer regular2 = new Customer(system, 4, "Regular Eve",
Thread.NORM_PRIORITY);
        Customer regular3 = new Customer(system, 2, "Regular Charlie",
Thread.NORM_PRIORITY);


        // Start threads
        vip1.start();
        vip2.start();
        regular1.start();
        regular2.start();
        regular3.start();


        try {
            vip1.join();
            vip2.join();
            regular1.join();
            regular2.join();
            regular3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }


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

## 6. Output:

```
VIP John successfully booked seat: 1
VIP Alice successfully booked seat: 2
Regular Bob successfully booked seat: 3
Regular Eve successfully booked seat: 4
Regular Charlie tried to book an already booked seat: 2
All bookings processed.
```

## 7. Learning Outcomes:

- Password validation using character type checks and length constraints.
- Efficient data management using Java Collections (ArrayList).
- Multi-threading with synchronized methods to prevent race conditions.
- Thread prioritization to handle VIP bookings first.
- Searching and filtering data using structured collections.
- Real-world application of Java concurrency for booking systems.