Experiment 4

Student Name: Astha	**UID: 22BCS10262**
Branch: CSE	**Section/Group:IOT-641-B**
Semester: 6th	**DOP:11/02/2025**
Subject: Java Lab	**Subject Code: 22CSH-359**

Aim: To develop 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: To develop a Java program that uses an ArrayList to store and manage employee details (ID, Name, and Salary). The program allows users to: Add a new employee.
Update an existing employee's details.
Remove an employee from the list.
Search for an employee by ID.
Display all employees in the list.

Algorithm:

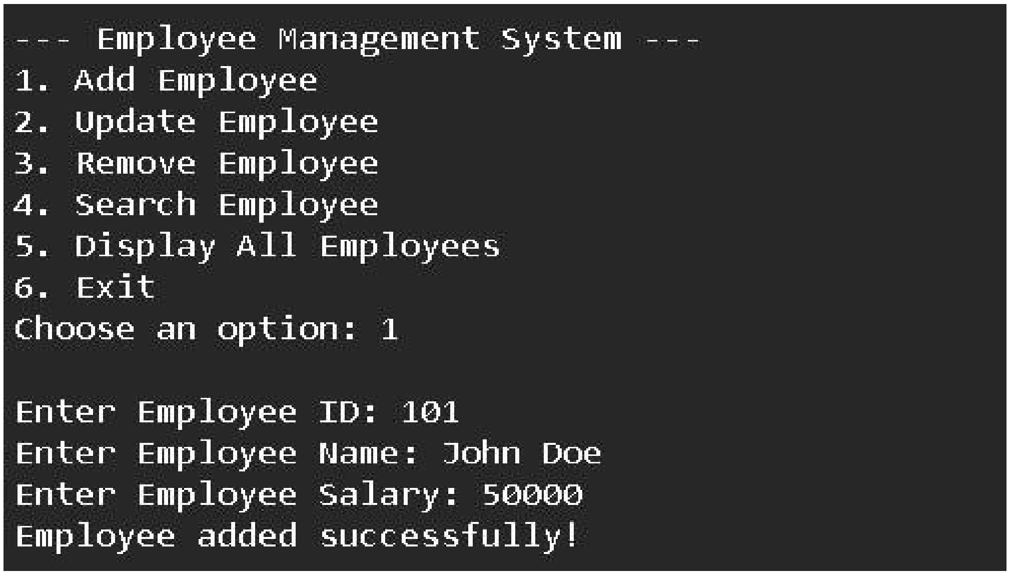
1.Start
2.Create an Employee class with attributes:
3.ID (int), Name (String), Salary (double).
4.Use an ArrayList to store multiple employees.
5.Display a menu with options:
6.Add an Employee
7.Update Employee Details
8.Remove an Employee
9.Search for an Employee
10.Display All Employees
11.Exit
12.Based on user input, perform the respective operation.
13.If updating or removing, search for the employee by ID.
14.Display confirmation messages after each operation.
5.Loop the menu until the user chooses to exit.
16.End

Code:

```
import java.util.ArrayList;
import java.util.Scanner;
// Employee class to store details class
Employee {
int id;
String name;
double salary;
// Constructor
public Employee(int id, String name, double salary) {
this.id = id;
this.name = name;
this.salary = salary;
}// Display Employee details
@Override   public
String toString() {
 return "ID: " + id + ", Name: " + name + ", Salary: " + salary;
}
}
public class EmployeeManagementSystem {
static ArrayList<Employee> employees = new ArrayList<>();
static Scanner scanner = new Scanner(System.in);
public static void main(String[] args) {
    while (true)
{
System.out.println("\n--- Employee 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("Choose an option: ");
int choice = scanner.nextInt();
scanner.nextLine(); // Consume newline
switch (choice) {
case 1:
   addEmployee();
break;
case 2:
    updateEmployee();
break;
case 3:
    removeEmployee();
break;
case 4:
    searchEmployee();
break;
case 5:
    displayAllEmployees();
break;
case 6:
System.out.println("Exiting Employee Management System.");
scanner.close();
return;
default:
System.out.println("Invalid choice! Please try again.");
}
}
}
// Add Employee
public static void addEmployee() { System.out.print("Enter Employee ID: ");
int id = scanner.nextInt();
scanner.nextLine(); // Consume newline
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!");
}
// Update Employee
public static void updateEmployee() {
    System.out.print("Enter Employee ID to update: ");
int id = scanner.nextInt();
scanner.nextLine(); // Consume newline
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();
System.out.println("Employee details updated successfully!");
return;
}
}
System.out.println("Employee not found!");
}
// Remove Employee
public static void removeEmployee() {
    System.out.print("Enter Employee ID to remove: ");
int id = scanner.nextInt();

for (Employee emp : employees) {
    if (emp.id == id) {
        employees.remove(emp);
System.out.println("Employee removed successfully!");
return;
}
}
System.out.println("Employee not found!");
}
// Search Employee
public static void searchEmployee() {
System.out.print("Enter Employee ID to search: ");
int id = scanner.nextInt();
for (Employee emp : employees) {
    if (emp.id == id) {
        System.out.println("Employee Found: " + emp);
        return;
}
}
System.out.println("Employee not found!");
}
// Display All Employees
public static void displayAllEmployees() {
    if (employees.isEmpty()) {
System.out.println("No employees found!");
} else {
System.out.println("\nEmployee List:");
for (Employee emp : employees) {
System.out.println(emp);
}
}
}
}
```

Output:



```
--- Employee Management System ---
1. Add Employee
2. Update Employee
3. Remove Employee
4. Search Employee
5. Display All Employees
6. Exit
Choose an option: 1

Enter Employee ID: 101
Enter Employee Name: John Doe
Enter Employee Salary: 50000
Employee added successfully!
```

Question 2

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: To develop a Java program using the Collection interface to store and manage playing cards. The program will help users:

Store cards in a collection.

Search for cards by a given symbol (e.g., Hearts, Spades).

Display all available cards in the collection.

Algorithm:

- Start
- Create a Card class with attributes:
- Symbol (String), Number (String).
- Use a Collection (ArrayList) to store multiple card objects.
- Display a menu with options:
- Add a card.
- Find all cards by symbol.
- Display all stored cards.
- Exit the program.

- Based on user input, perform the respective operation.
- If searching, iterate through the list and find all matching symbols.
- Display confirmation messages after each operation.
- Loop the menu until the user chooses to exit.
- End

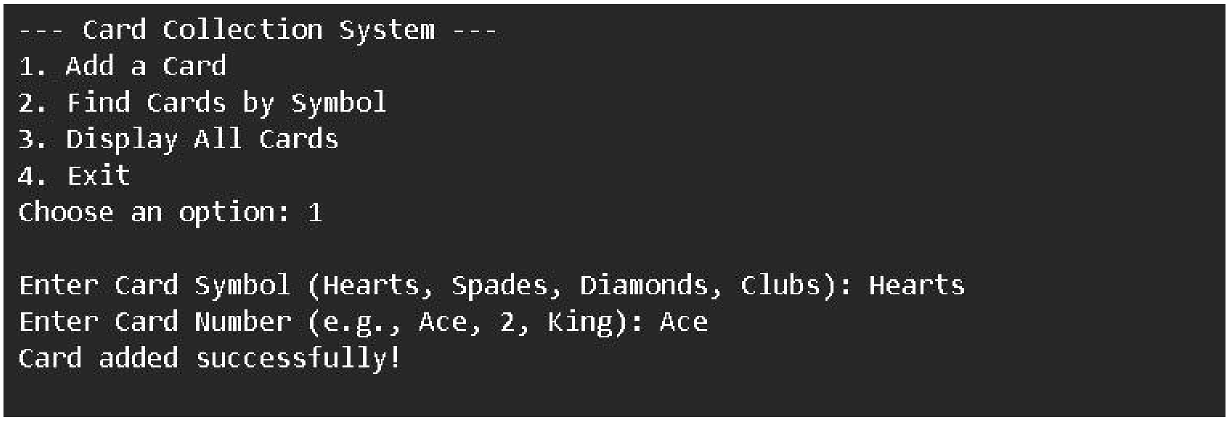
CODE:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
// Card class to store symbol and number
class Card {
    private String symbol;
    private String number;
// Constructor
public Card(String symbol, String number) {
    this.symbol = symbol;
    this.number = number;
}
public String getSymbol() {
    return symbol;
}
@Override
public String toString() {
+ number + " of " + symbol;
}
}
 return "Card: "
public class CardCollectionSystem {
    static List<Card>
cardCollection = new ArrayList<>();
```

```
static Scanner
scanner = new Scanner(System.in);
public static void main(String[] args) {
while (true) {
System.out.println("\n--- Card Collection System ---");
System.out.println("1. Add a Card");
System.out.println("2. Find Cards by Symbol");
System.out.println("3. Display All Cards");
System.out.println("4. Exit");
System.out.print("Choose an option: ");
int choice = scanner.nextInt();
scanner.nextLine(); // Consume newline
switch (choice)
{
case 1:
   addCard();
break;
case 2:
   findCardsBySymbol();
break;
case 3:
   displayAllCards();
break;
case 4:
   System.out.println("Exiting Card Collection System.");
   scanner.close();
return;
   default:
stem.out.println("Invalid choice! Please try again.");
}
}
}
// Add a new card     public
static void addCard() {
System.out.print("Enter Card Symbol (Hearts, Spades, Diamonds, Clubs): ");
String symbol = scanner.nextLine();
System.out.print("Enter Card Number (e.g., Ace, 2, King): ");
String number = scanner.nextLine();
cardCollection.add(new Card(symbol, number));
System.out.println("Card added successfully!");
}
// Find and display all cards of a given symbol
public static void findCardsBySymbol() {
System.out.print("Enter Symbol to search for (Hearts, Spades, Diamonds, Clubs):");
 String symbol = scanner.nextLine();
boolean found = false;
System.out.println("\nCards in " + symbol + ":");
for (Card card : cardCollection) {
if (card.getSymbol().equalsIgnoreCase(symbol))
{
true;
}
 System.out.println(card);
}
}
if (!found) {
 found =
System.out.println("No cards found with the symbol " + symbol);
} // Display all stored cards
static void displayAllCards() {
```

```
(cardCollection.isEmpty()) {
 public
if
System.out.println("No cards stored!");
} else {
System.out.println("\nAll Cards:");
for (Card card : cardCollection) {
System.out.println(card);
}
}
}
```

Output:



```
--- Card Collection System ---
1. Add a Card
2. Find Cards by Symbol
3. Display All Cards
4. Exit
Choose an option: 1

Enter Card Symbol (Hearts, Spades, Diamonds, Clubs): Hearts
Enter Card Number (e.g., Ace, 2, King): Ace
Card added successfully!
```

Ques3.

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:

□ Thread Synchronization:

- Ensure that multiple threads do not book the same seat simultaneously.
- Use synchronization mechanisms like synchronized, Lock, or Atomic variables in Java.

□ Thread Priorities:

- Assign higher priorities to VIP customers so their booking requests are processed before others.

- Utilize Thread.setPriority() in Java or implement a priority queue.

□ Seat Allocation Logic:

- Maintain a shared data structure (e.g., a Set or Map) to track booked seats.
- Prevent race conditions using proper concurrency controls.

□ Multi-threading Implementation:

- Create multiple threads simulating different users trying to book tickets.
- Use ExecutorService or ThreadPoolExecutor for better thread management.

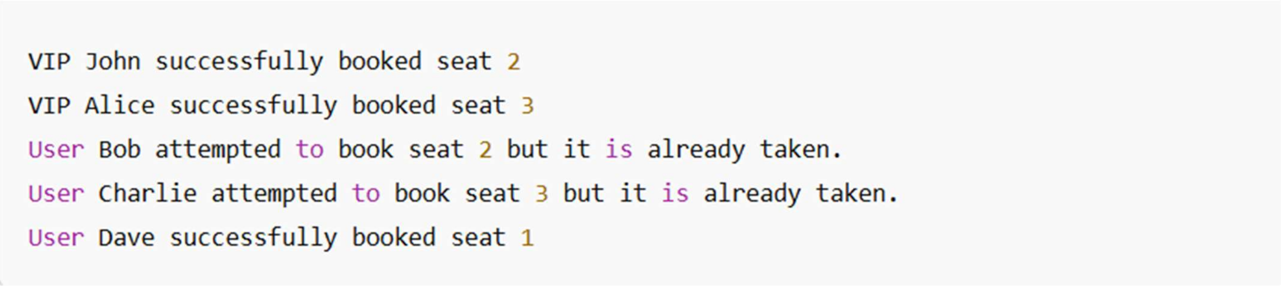
□ Fairness and Performance:

- Ensure non-VIP users still get a chance to book if VIP requests are not continuous.
- Optimize thread execution for minimal delay.

CODE:

```
import java.util.concurrent.locks.*;
class TicketBookingSystem {
    private final boolean[] seats;
    private final Lock lock;
    public TicketBookingSystem(int totalSeats) {
        this.seats = new boolean[totalSeats]; // false means available
        this.lock = new ReentrantLock();
    }
public void bookSeat(int seatNumber, String customerName) {
        lock.lock();
        try {
            if (seatNumber < 0 || seatNumber >= seats.length) {
                System.out.println(customerName + " attempted to book an invalid seat.");
                return;
            }
            if (!seats[seatNumber]) {
                seats[seatNumber] = true;
                System.out.println(customerName + " successfully booked seat " +
seatNumber);
            } else {
System.out.println(customerName + " attempted to book seat " + seatNumber + " but it is already taken.");
            }
        } 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 TicketBookingApp {
    public static void main(String[] args) {
        TicketBookingSystem system = new TicketBookingSystem(5);
// Creating threads with different priorities (VIP customers get higher priority)
        BookingThread vip1 = new BookingThread(system, 2, "VIP John",
Thread.MAX_PRIORITY);
        BookingThread vip2 = new BookingThread(system, 3, "VIP Alice",
Thread.MAX_PRIORITY);
        BookingThread user1 = new BookingThread(system, 2, "User Bob",
Thread.MIN_PRIORITY);
        BookingThread user2 = new BookingThread(system, 3, "User Charlie",
Thread.MIN_PRIORITY);
        BookingThread user3 = new BookingThread(system, 1, "User Dave",
Thread.NORM_PRIORITY);
// Start threads
        vip1.start();
        vip2.start();
        user1.start();
        user2.start();
        user3.start();
    }
}
```

Output:



```
VIP John successfully booked seat 2
VIP Alice successfully booked seat 3
User Bob attempted to book seat 2 but it is already taken.
User Charlie attempted to book seat 3 but it is already taken.
User Dave successfully booked seat 1
```

Learning Outcomes:

- **Inheritance**: Use of base and derived classes for shared attributes and methods.
- **Method Overriding**: Custom implementation of methods in subclasses.
- **Constructor**: Initializing object attributes using constructors.
- **Encapsulation**: Storing and manipulating data within objects.
- **Polymorphism**: Different behavior of calculateInterest() based on object type.
- **Interest Calculation**: Implementing FD and RD interest formulas.
- **Class Interaction**: Creating objects and calling methods to display details.