ULAB
UNIVERSITY OF LIBERAL ARTS BANGLADESH

OOP LAB FINAL ASSIGNMENT

SUBMITTED TO: SHAKIB MAHMUD DIPTO
PROGRAM:CSE 2104
SEC:04

Submitted By:

MAHBUB BILLAH NAFIS
ID: (223014051)

Assignment Title

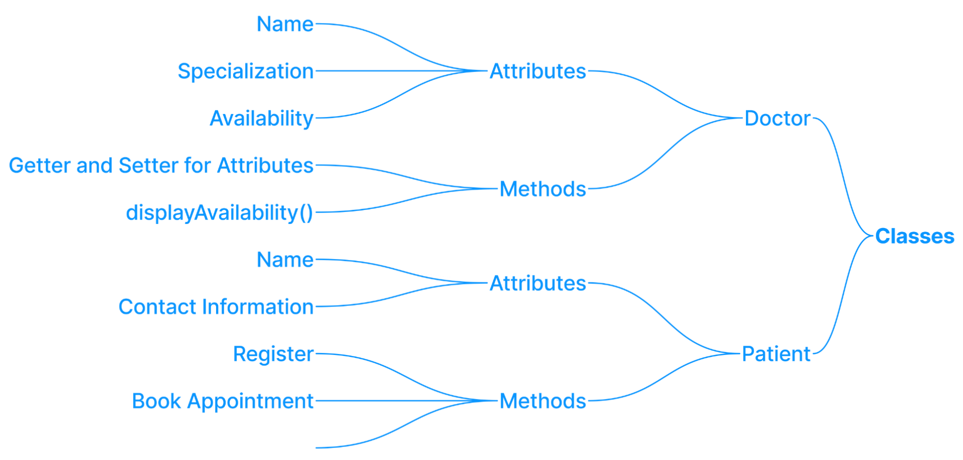
Clinic Management System

Problem Statement:
Task developed a Doctor's Appointment Management System for a local clinic. The clinic has multiple doctors, each specializing in different areas, and patients can book appointments based on doctor availability. The system should allow patients to register, view available doctors, and book an appointment, while doctors can view their appointment schedules. The system should store and retrieve all data through file handling to maintain persistence across sessions.

System Design

The Clinic Management System designed with Java language and implementing Object-C

- How the system w



Classes
Inheritance
Polymorphism
Encapsulation

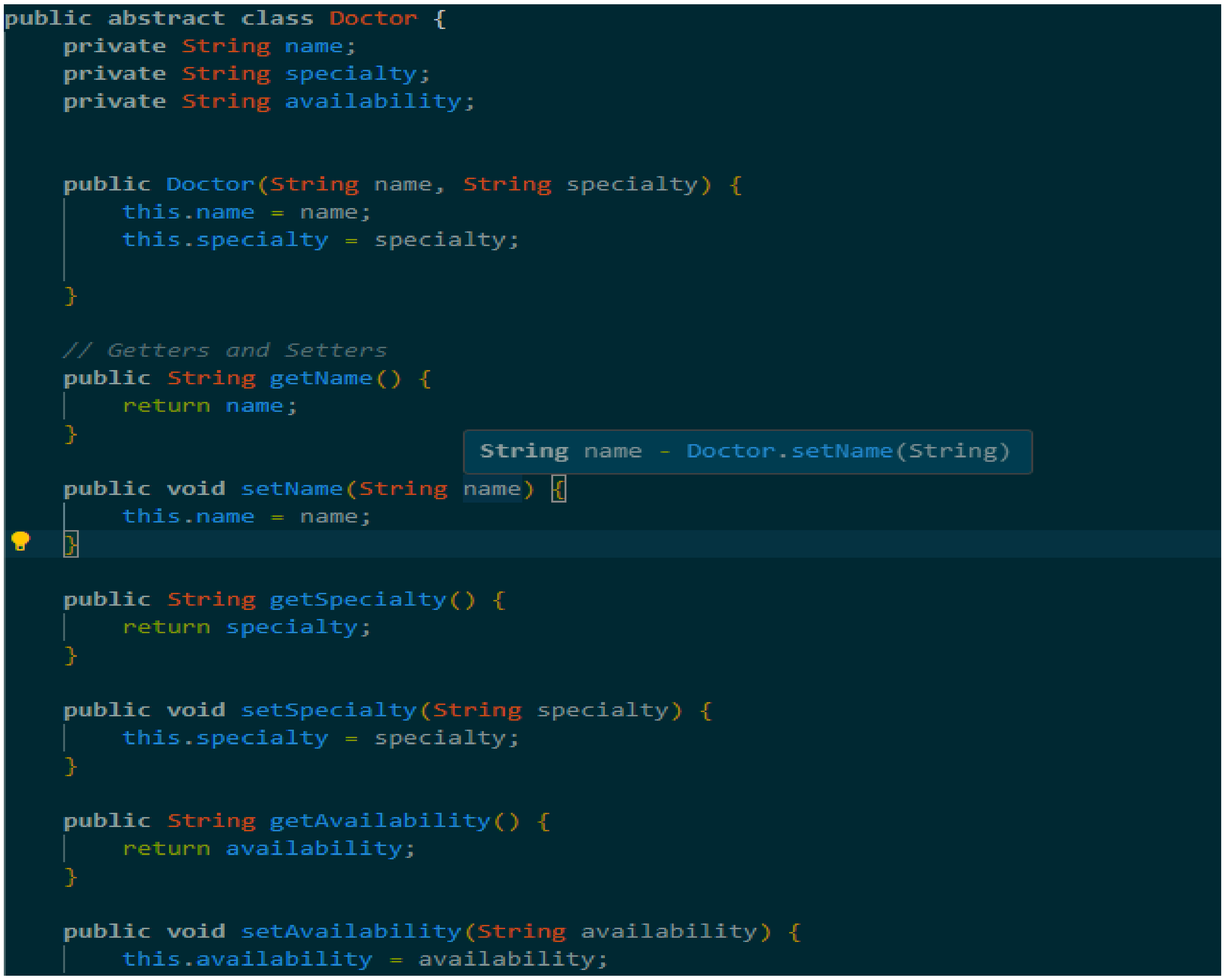
Classes: There are 5 pu

- ☐ **ClinicManagementSystem**
- ☐ **Doctor**
- ☐ **DoctorsDetails**
- ☐ **Interface**
- ☐ **Patient**

Inheritance : 2 Subclass

- ☐ **Specialist**
- ☐ **GeneralPractitioner**

Code Explanation



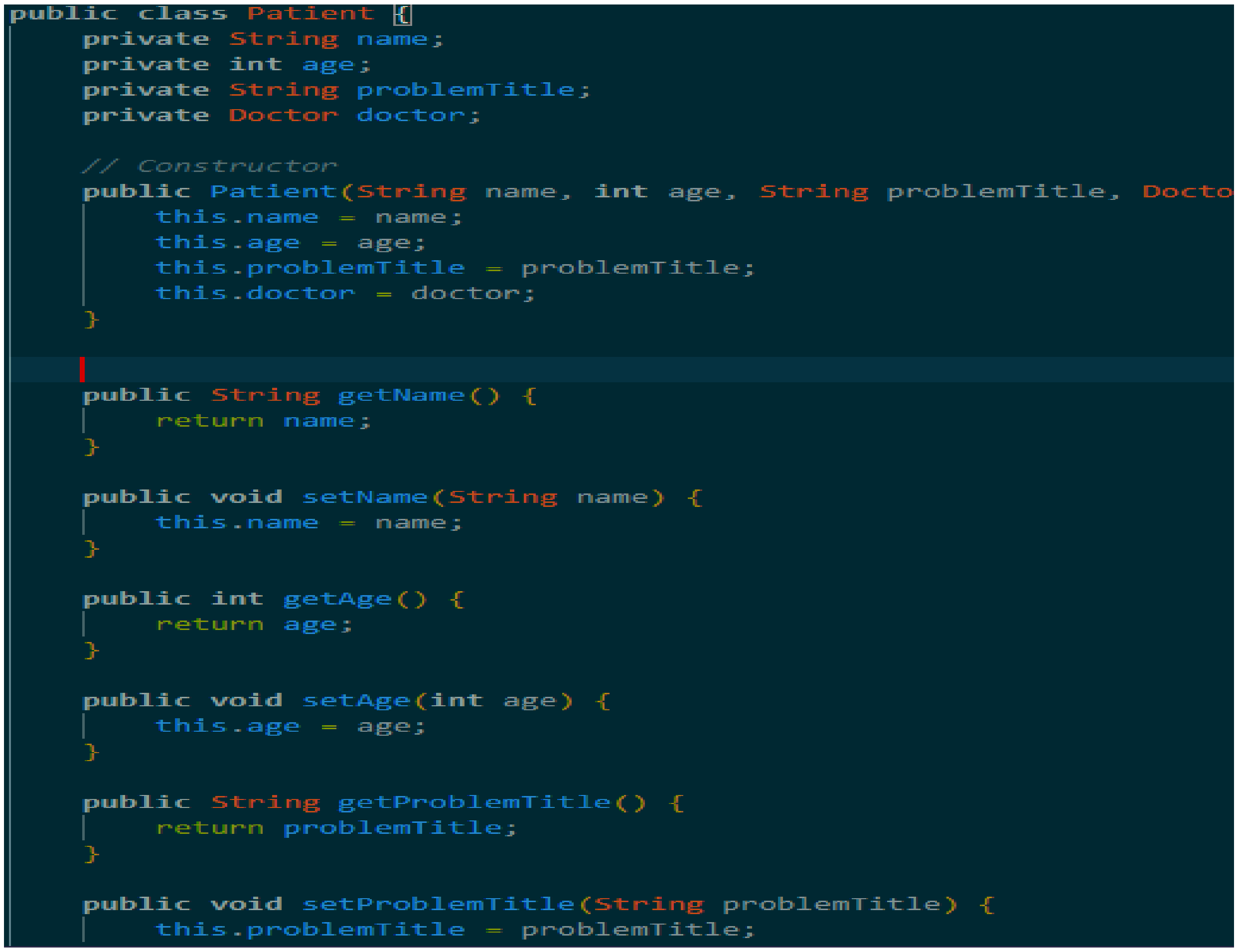
```
public abstract class Doctor {
    private String name;
    private String specialty;
    private String availability;


    public Doctor(String name, String specialty) {
        this.name = name;
        this.specialty = specialty;

    }

    // Getters and Setters
    public String getName() {
        return name;
    }
                                  String name - Doctor.setName(String)
    public void setName(String name) {
        this.name = name;
    }

    public String getSpecialty() {
        return specialty;
    }

    public void setSpecialty(String specialty) {
        this.specialty = specialty;
    }

    public String getAvailability() {
        return availability;
    }

    public void setAvailability(String availability) {
        this.availability = availability;
```

The Doctor class is as follows Attributes: name: A String representing the doctor's name. specialty: A String indicating the doctor's area of expertise.

Constructor: The constructor initializes the name and specialty attributes while setting a default value for availability.

Getters and Setters: These methods provide controlled access to the private attributes, allowing for retrieval and modification of their values.

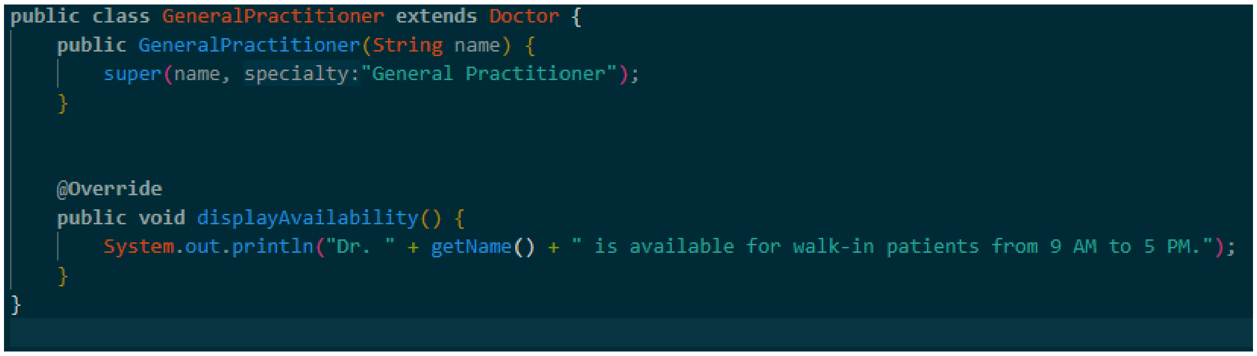
Abstract Method: displayAvailability() This method is declared abstract.



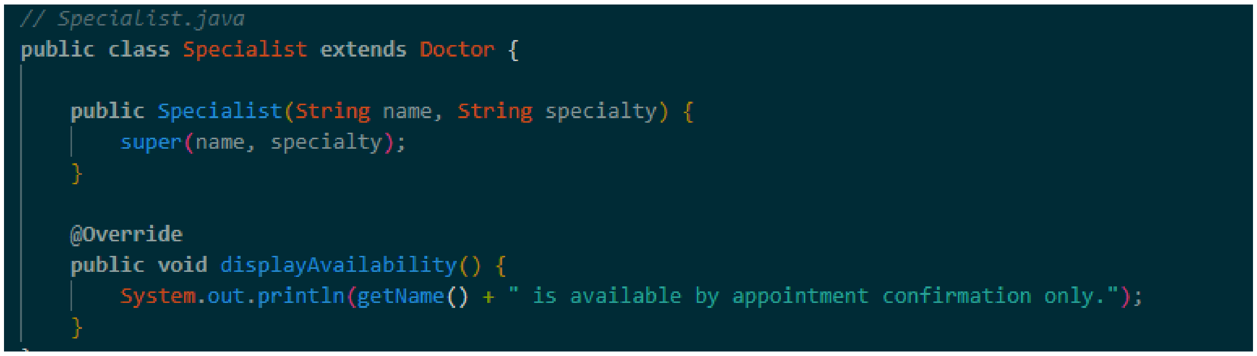
```java
public class Patient {
    private String name;
    private int age;
    private String problemTitle;
    private Doctor doctor;

    // Constructor
    public Patient(String name, int age, String problemTitle, Docto
        this.name = name;
        this.age = age;
        this.problemTitle = problemTitle;
        this.doctor = doctor;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getProblemTitle() {
        return problemTitle;
    }

    public void setProblemTitle(String problemTitle) {
        this.problemTitle = problemTitle;
```

The Patient class follows: save : The class contains four private fields: name, age, problemTitle, and doctor. These fields store the essential information about a patient.
Constructor: A constructor is defined to initialize the fields when a new Patient object is created.
Getters and Setters: Public methods are provided to retrieve and modify the values of the private fields,
principles of encapsulation.

Encapsulation: This principle is illustrated through the use of private fields, which restrict direct access to the class's data. Instead, public getter and setter methods are provided to manipulate these fields, ensuring controlled access. **Abstraction:** The class abstracts the details of a patient, allowing users to interact with the patient object without needing to understand the underlying implementation. **The Patient class contains a reference to a Doctor object, showcasing the relationship between different classes and how they can work together.**

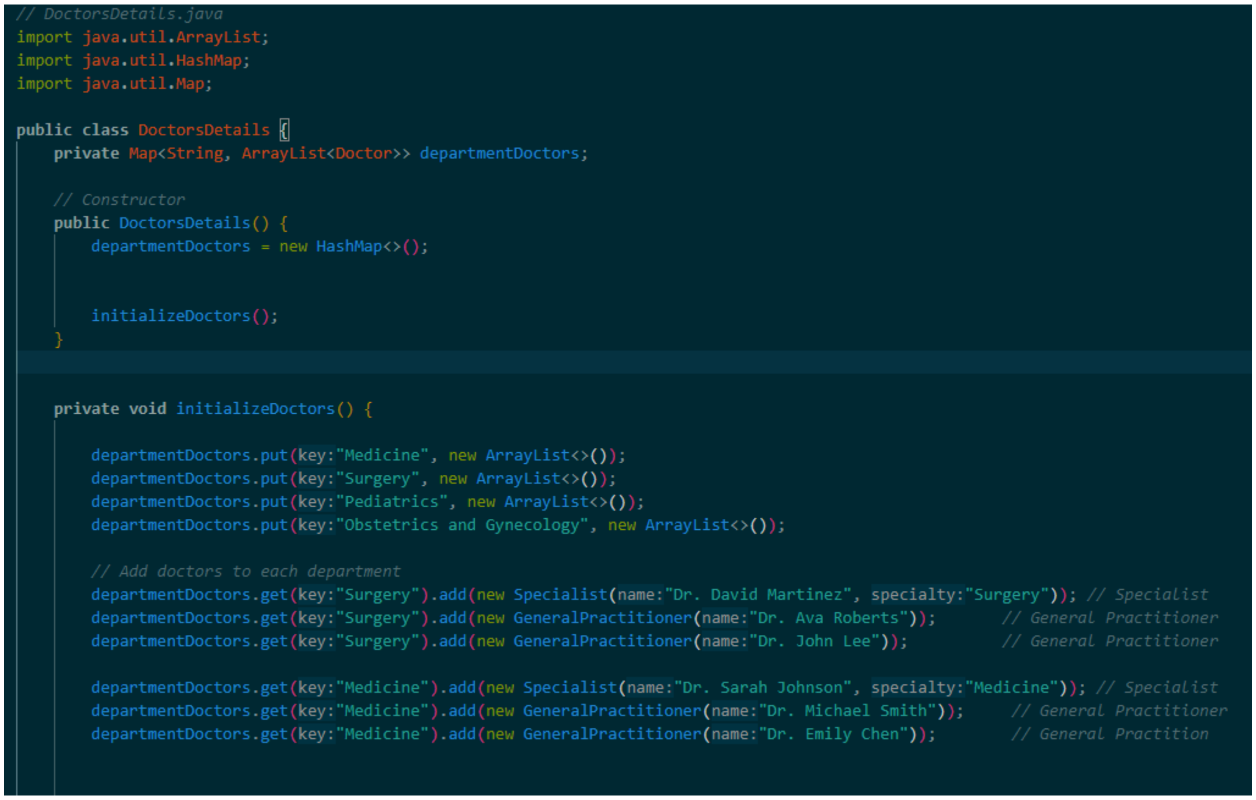


```
public class GeneralPractitioner extends Doctor {
    public GeneralPractitioner(String name) {
        super(name, specialty:"General Practitioner");
    }


    @Override
    public void displayAvailability() {
        System.out.println("Dr. " + getName() + " is available for walk-in patients from 9 AM to 5 PM.");
    }
}
```

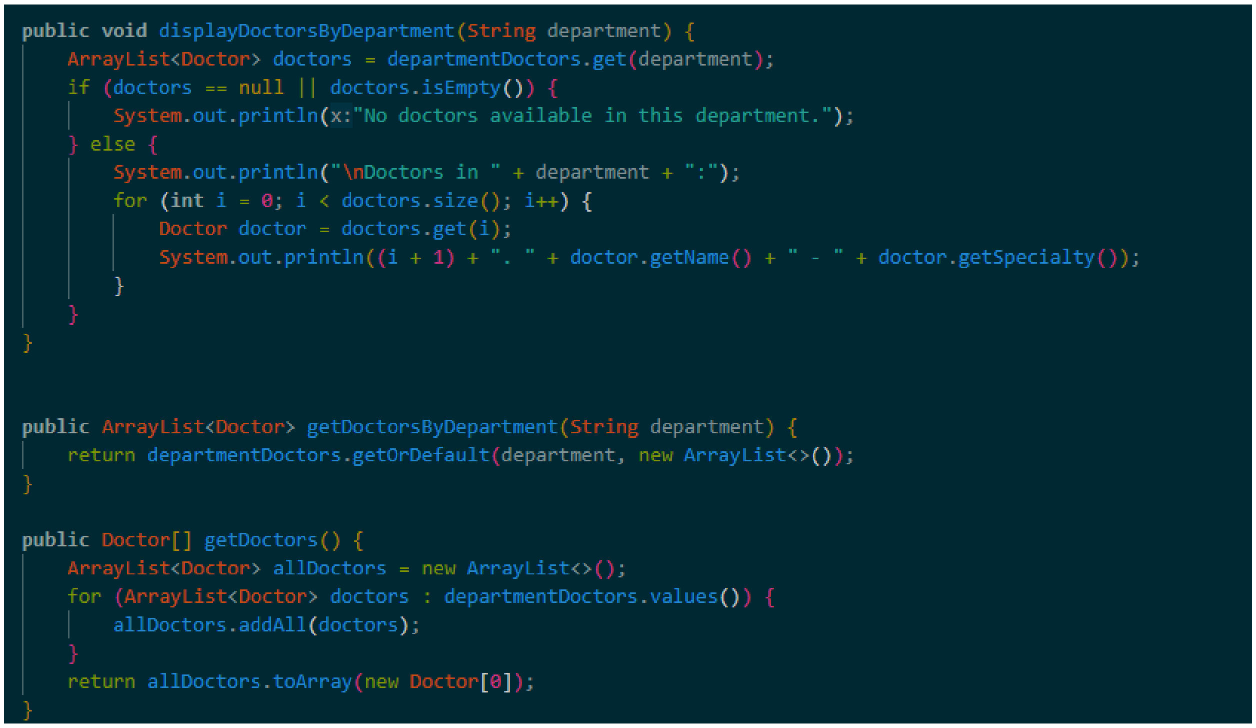


```
// Specialist.java
public class Specialist extends Doctor {

    public Specialist(String name, String specialty) {
        super(name, specialty);
    }

    @Override
    public void displayAvailability() {
        System.out.println(getName() + " is available by appointment confirmation only.");
    }
```

The **GeneralPractitioner and Specialist** class follows: **Class Declaration:** The class is declared as public, making it accessible from other classes. **Constructor:** The constructor takes a String parameter name and calls the superclass constructor using super(name, "General Practitioner"), which initializes the name and the type of doctor. **Method Override:** The displayAvailability method is **overridden** to provide specific availability information for a general practitioner. **Inheritance:** This is the OOP concept that allows a class to inherit fields and methods from another class. In this case, **GeneralPractitioner & Specialist** inherits from the Doctor class, which means it can be the properties and methods defined in Doctor. **Encapsulation:** This principle involves bundling the data **(attributes)** and methods that operate on the data into a single unit or 2 classes. It restricts direct access to some of the object's components. **Polymorphism:** This allows methods to do different things based on the object it is acting . In the provided code, the displayAvailability method is overridden to provide a specific implementation for the **GeneralPractitioner & Specialist**class.



```java
// DoctorsDetails.java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class DoctorsDetails {
    private Map<String, ArrayList<Doctor>> departmentDoctors;

    // Constructor
    public DoctorsDetails() {
        departmentDoctors = new HashMap<>();


        initializeDoctors();
    }


    private void initializeDoctors() {

        departmentDoctors.put(key:"Medicine", new ArrayList<>());
        departmentDoctors.put(key:"Surgery", new ArrayList<>());
        departmentDoctors.put(key:"Pediatrics", new ArrayList<>());
        departmentDoctors.put(key:"Obstetrics and Gynecology", new ArrayList<>());

        // Add doctors to each department
        departmentDoctors.get(key:"Surgery").add(new Specialist(name:"Dr. David Martinez", specialty:"Surgery")); // Specialist
        departmentDoctors.get(key:"Surgery").add(new GeneralPractitioner(name:"Dr. Ava Roberts"));       // General Practitioner
        departmentDoctors.get(key:"Surgery").add(new GeneralPractitioner(name:"Dr. John Lee"));          // General Practitioner

        departmentDoctors.get(key:"Medicine").add(new Specialist(name:"Dr. Sarah Johnson", specialty:"Medicine")); // Specialist
        departmentDoctors.get(key:"Medicine").add(new GeneralPractitioner(name:"Dr. Michael Smith"));     // General Practitioner
        departmentDoctors.get(key:"Medicine").add(new GeneralPractitioner(name:"Dr. Emily Chen"));        // General Practition
```

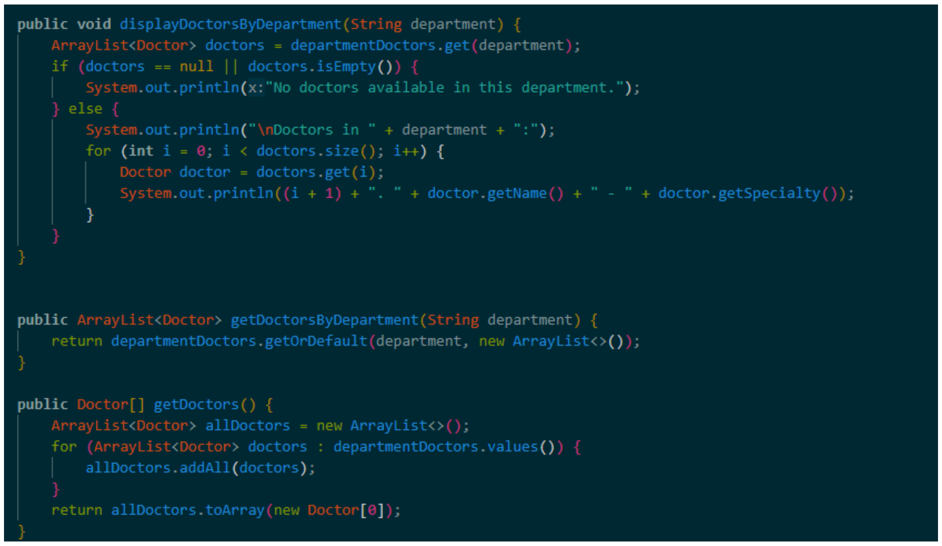


```java
public void displayDoctorsByDepartment(String department) {
    ArrayList<Doctor> doctors = departmentDoctors.get(department);
    if (doctors == null || doctors.isEmpty()) {
        System.out.println(x:"No doctors available in this department.");
    } else {
        System.out.println("\nDoctors in " + department + ":");
        for (int i = 0; i < doctors.size(); i++) {
            Doctor doctor = doctors.get(i);
            System.out.println((i + 1) + ". " + doctor.getName() + " - " + doctor.getSpecialty());
        }
    }
}


public ArrayList<Doctor> getDoctorsByDepartment(String department) {
    return departmentDoctors.getOrDefault(department, new ArrayList<>());
}

public Doctor[] getDoctors() {
    ArrayList<Doctor> allDoctors = new ArrayList<>();
    for (ArrayList<Doctor> doctors : departmentDoctors.values()) {
        allDoctors.addAll(doctors);
    }
    return allDoctors.toArray(new Doctor[0]);
}
```

The DoctorsDetails class is the core of the system, managing a collection of doctors initialized for the department. class: A private store department and their corresponding lists of doctors. A constructor that initializes and populates it with sample data. Methods to display doctors by department, retrieve doctors by department, and we get a list of all doctors.

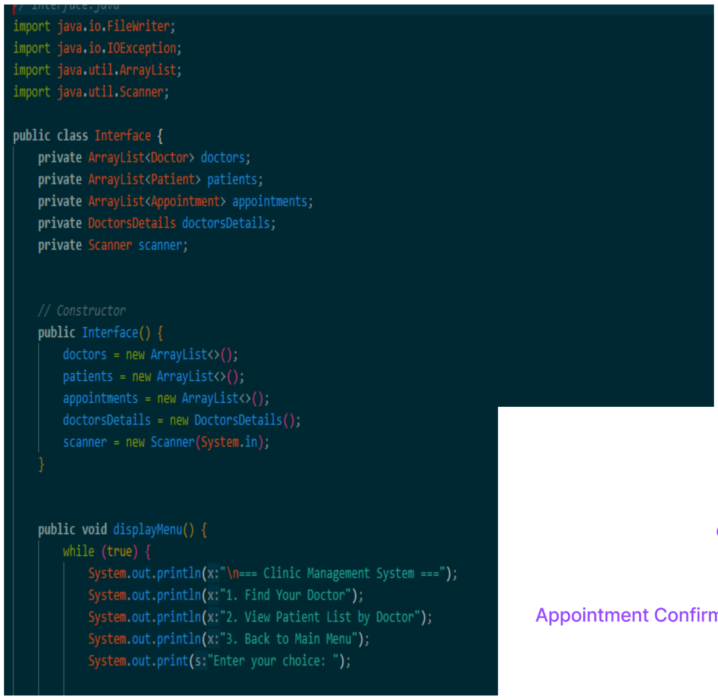
Inheritance: The class is called Doctor, from which specialized classes such as **Specialist & GeneralPractitioner inherit.** This allows for the extension of functionality while maintaining a common interface. **Polymorphism:** The use of a common method **getSpecialty()** in the Doctor class allows different types of doctors to be treated uniformly, enabling flexibility in how doctors are managed and displayed.
Constructor: Initializes the departmentDoctors and calls initializeDoctors() to populate it with data. initializeDoctors(): This private method sets up the initial state of the code by creating departments and adding doctors to them.

displayDoctorsByDepartment(String department): This method retrieves and displays the list of doctors in a specified department. It checks if the department exists and whether it contains any doctors, providing appropriate feedback. getDoctorsByDepartment(String department): Returns a list of doctors for a **specified department,** defaulting to an empty list if the department does not exist. **getDoctors():** Compiles a comprehensive list of all doctors across all departments and returns it as an array.

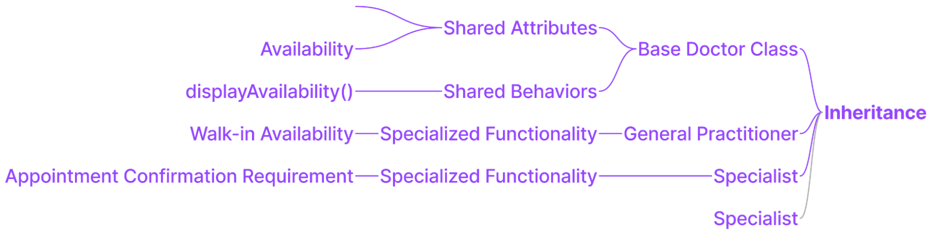


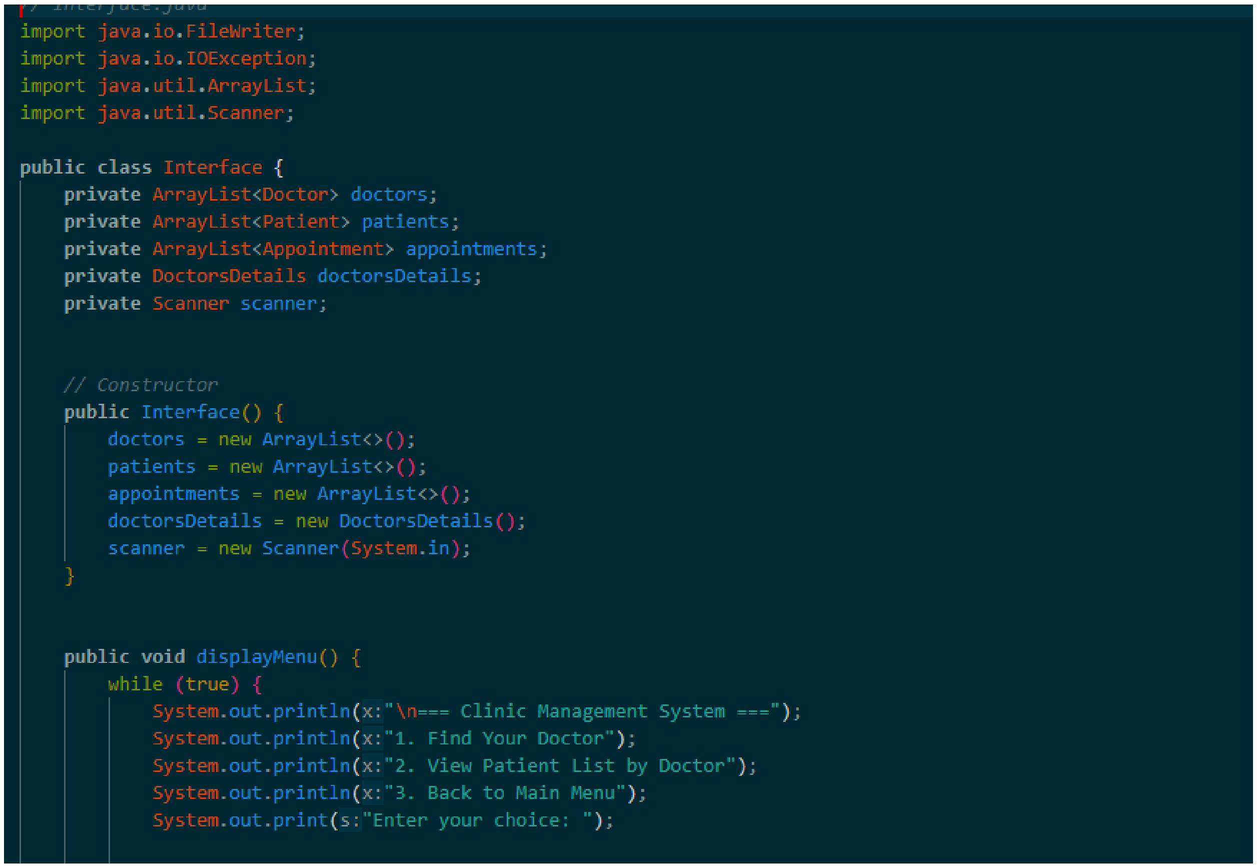
```
public void displayDoctorsByDepartment(String department) {
    ArrayList<Doctor> doctors = departmentDoctors.get(department);
    if (doctors == null || doctors.isEmpty()) {
        System.out.println(x:"No doctors available in this department.");
    } else {
        System.out.println("\nDoctors in " + department + ":");
        for (int i = 0; i < doctors.size(); i++) {
            Doctor doctor = doctors.get(i);
            System.out.println((i + 1) + ". " + doctor.getName() + " - " + doctor.getSpecialty());
        }
    }
}


public ArrayList<Doctor> getDoctorsByDepartment(String department) {
    return departmentDoctors.getOrDefault(department, new ArrayList<>());
}

public Doctor[] getDoctors() {
    ArrayList<Doctor> allDoctors = new ArrayList<>();
    for (ArrayList<Doctor> doctors : departmentDoctors.values()) {
        allDoctors.addAll(doctors);
    }
    return allDoctors.toArray(new Doctor[0]);
}
```

☐ Interface



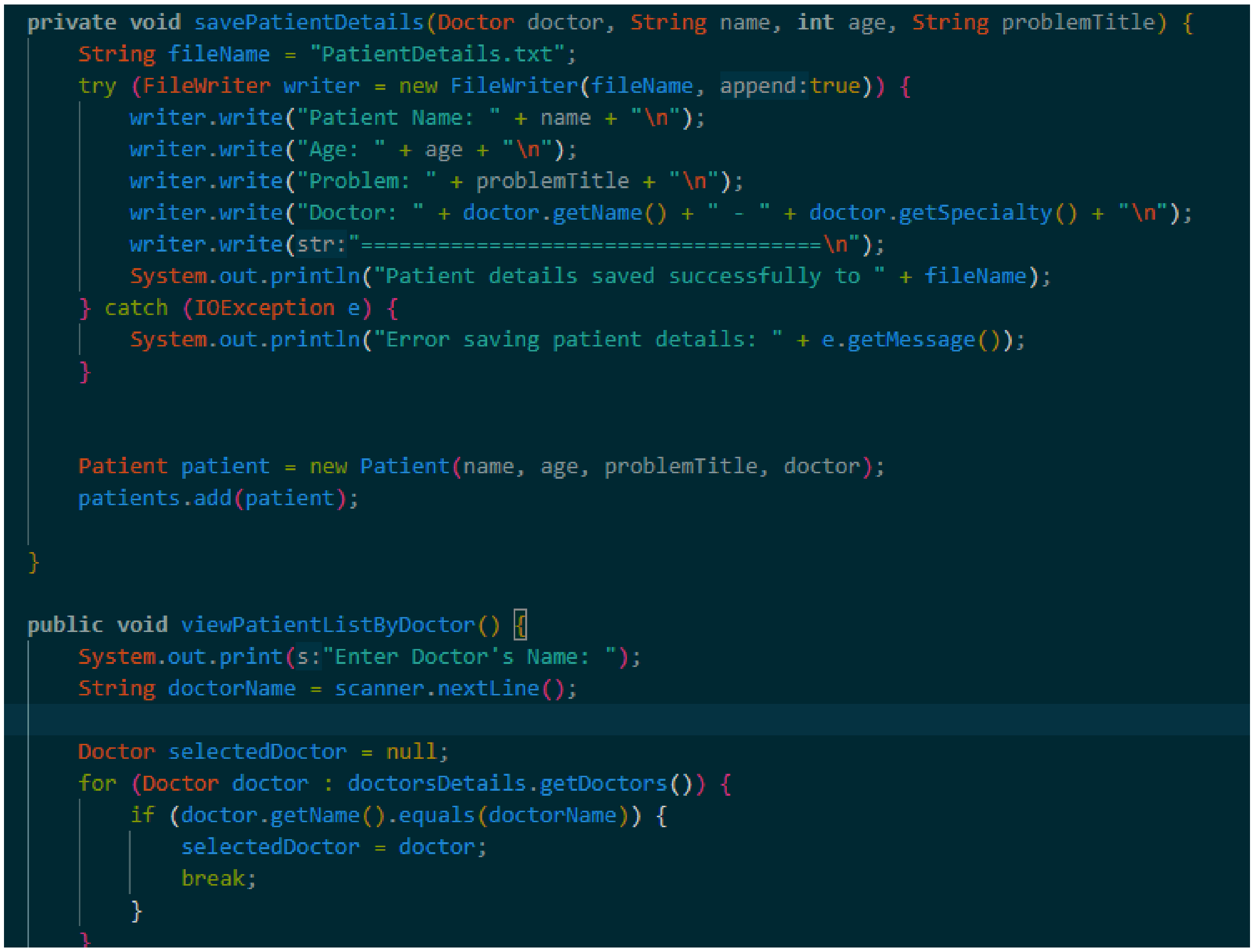
```java
// Interface.java
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class Interface {
    private ArrayList<Doctor> doctors;
    private ArrayList<Patient> patients;
    private ArrayList<Appointment> appointments;
    private DoctorsDetails doctorsDetails;
    private Scanner scanner;


    // Constructor
    public Interface() {
        doctors = new ArrayList<>();
        patients = new ArrayList<>();
        appointments = new ArrayList<>();
        doctorsDetails = new DoctorsDetails();
        scanner = new Scanner(System.in);
    }


    public void displayMenu() {
        while (true) {
            System.out.println(x:"\n=== Clinic Management System ===");
            System.out.println(x:"1. Find Your Doctor");
            System.out.println(x:"2. View Patient List by Doctor");
            System.out.println(x:"3. Back to Main Menu");
            System.out.print(s:"Enter your choice: ");
```



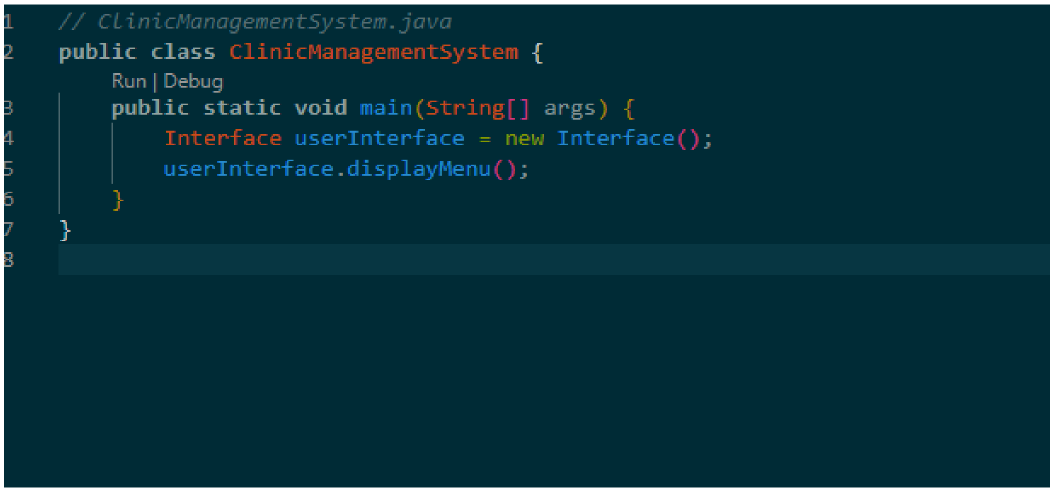


```java
// Interface.java
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class Interface {
    private ArrayList<Doctor> doctors;
    private ArrayList<Patient> patients;
    private ArrayList<Appointment> appointments;
    private DoctorsDetails doctorsDetails;
    private Scanner scanner;


    // Constructor
    public Interface() {
        doctors = new ArrayList<>();
        patients = new ArrayList<>();
        appointments = new ArrayList<>();
        doctorsDetails = new DoctorsDetails();
        scanner = new Scanner(System.in);
    }


    public void displayMenu() {
        while (true) {
            System.out.println(x:"\n=== Clinic Management System ===");
            System.out.println(x:"1. Find Your Doctor");
            System.out.println(x:"2. View Patient List by Doctor");
            System.out.println(x:"3. Back to Main Menu");
            System.out.print(s:"Enter your choice: ");
```

Encapsulation: The Interface class encapsulates the data related to doctors, patients, and appointments, along with methods to manipulate this data. Abstraction: The user interacts with a simplified interface without needing to understand the underlying complexities of data management.
Inheritance: Although not explicitly shown in the provided code, it is implied that classes such as Doctor, Patient, and Appointment may inherit from a common superclass or interface, promoting code reuse. Polymorphism: The use of method overloading and interfaces can allow for different **implementations of methods that share the same name,** although this is not explicitly demonstrated in the provided code. Code Structure The code is structured into a single class named Interface, **which contains: Attributes:** ArrayList<Doctor> doctors: Stores a list of doctors. ArrayList<Patient> patients: Stores a list of patients. ArrayList<Appointment> **appointments: Stores a list of appointments. DoctorsDetails doctorsDetails: An instance of a class that presumably manages doctor-related data.** Scanner scanner: user input. Constructor: **Methods: displayMenu():** Displays the main menu and handles user choices. **findYourDoctor(): Allows users to select a department and find doctors. selectDoctor(String department):** Lets users choose a doctor and input patient details.


```
private void savePatientDetails(Doctor doctor, String name, int age, String problemTitle) {
    String fileName = "PatientDetails.txt";
    try (FileWriter writer = new FileWriter(fileName, append:true)) {
        writer.write("Patient Name: " + name + "\n");
        writer.write("Age: " + age + "\n");
        writer.write("Problem: " + problemTitle + "\n");
        writer.write("Doctor: " + doctor.getName() + " - " + doctor.getSpecialty() + "\n");
        writer.write(str:"====================================\n");
        System.out.println("Patient details saved successfully to " + fileName);
    } catch (IOException e) {
        System.out.println("Error saving patient details: " + e.getMessage());
    }


    Patient patient = new Patient(name, age, problemTitle, doctor);
    patients.add(patient);

}

public void viewPatientListByDoctor() {
    System.out.print(s:"Enter Doctor's Name: ");
    String doctorName = scanner.nextLine();


    Doctor selectedDoctor = null;
    for (Doctor doctor : doctorsDetails.getDoctors()) {
        if (doctor.getName().equals(doctorName)) {
            selectedDoctor = doctor;
            break;
        }
    }
```

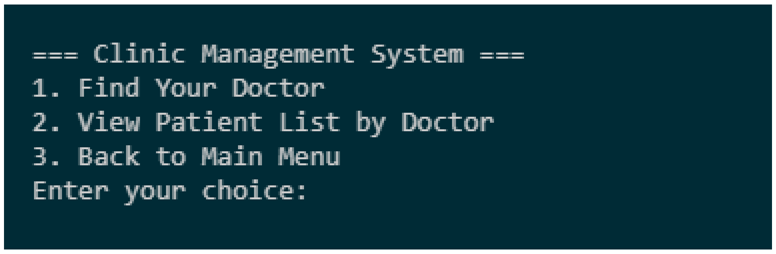
The code consists of two methods: savePatientDetails and viewPatientListByDoctor, along with a helper method getPatientsByDoctor. The code **savePatientDetails: This method takes a Doctor object and patient details as parameters, saves the information to a text file, and adds the patient to an internal list. viewPatientListByDoctor:** This method prompts the user for a doctor's name, retrieves the corresponding doctor object, and displays a list of patients associated with that doctor. getPatientsByDoctor: A private helper method that **filters the list of patients based on the specified doctor.**



```
// ClinicManagementSystem.java
public class ClinicManagementSystem {
    Run | Debug
    public static void main(String[] args) {
        Interface userInterface = new Interface();
        userInterface.displayMenu();
    }
}
```

Encapsulation: The ClinicManagementSystem class encapsulates the main functionality of the application, providing a clear entry point for execution. Abstraction: The use of an Interface class abstracts the complexities of user interaction, allowing the main class to focus on high-level operations. Modularity: By separating the user interface from the main application logic, the code promotes modularity, making it easier to maintain and extend. Code Structure The structure of the code is straightforward, consisting of a single class with a main method. The main method serves as the entry point for the application, where the execution begins. Within this method, an instance of the Interface class is created, and its displayMenu method is invoked to present the user with options.

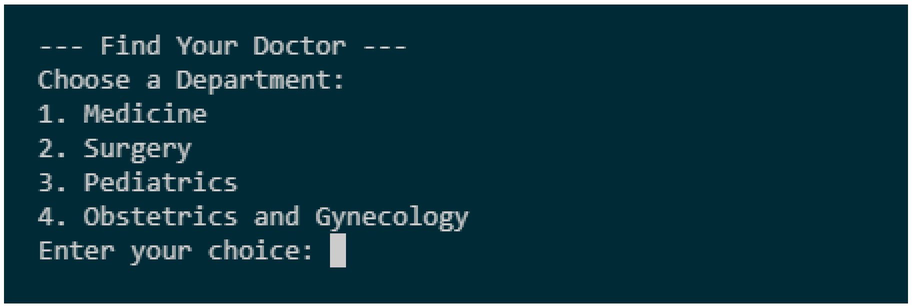
OutPut Explanation



```
=== Clinic Management System ===
1. Find Your Doctor
2. View Patient List by Doctor
3. Back to Main Menu
Enter your choice:
```

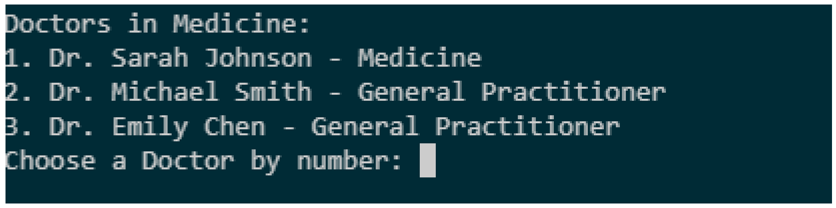
# There are 3 options in this basic interface
By pressing 1 it will show how many departments in this clinic.

- **By pressing 1**



```
--- Find Your Doctor ---
Choose a Department:
1. Medicine
2. Surgery
3. Pediatrics
4. Obstetrics and Gynecology
Enter your choice:
```

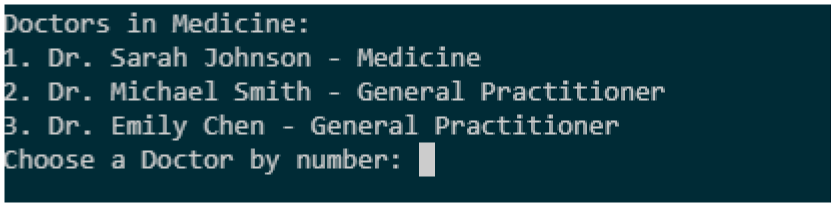
#There are 4 departments in this clinic system .Choose one of them to go on the Doctors list for this department.



```
Doctors in Medicine:
1. Dr. Sarah Johnson - Medicine
2. Dr. Michael Smith - General Practitioner
3. Dr. Emily Chen - General Practitioner
Choose a Doctor by number:
```

-

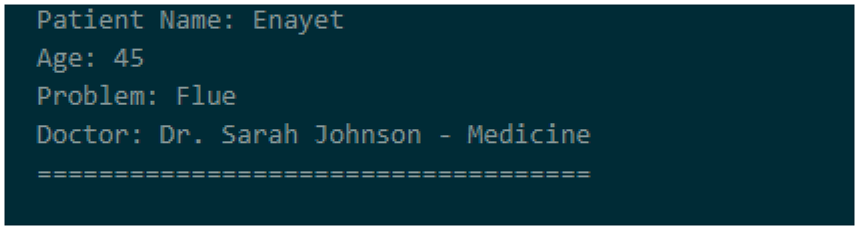
In this section there are 3 doctors in this department .Number 1 is specialist in this department other doctors are General practitioners. By choosing one of them, the system will ask the patient to give his information:name,age,number,problem title.



```
Doctors in Medicine:
1. Dr. Sarah Johnson - Medicine
2. Dr. Michael Smith - General Practitioner
3. Dr. Emily Chen - General Practitioner
Choose a Doctor by number:
```

❖

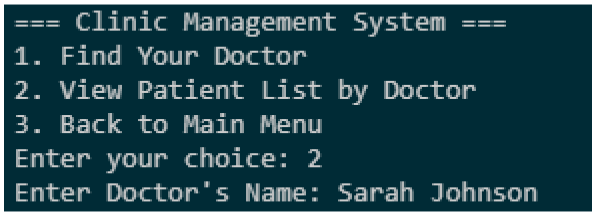
After this process the patient list will be saved and it shows in the patient interface.



```
Patient Name: Enayet
Age: 45
Problem: Flue
Doctor: Dr. Sarah Johnson - Medicine
====================================
```

❖

#Back to the main menu.BY presing 2 systems will ask the doctor's name .

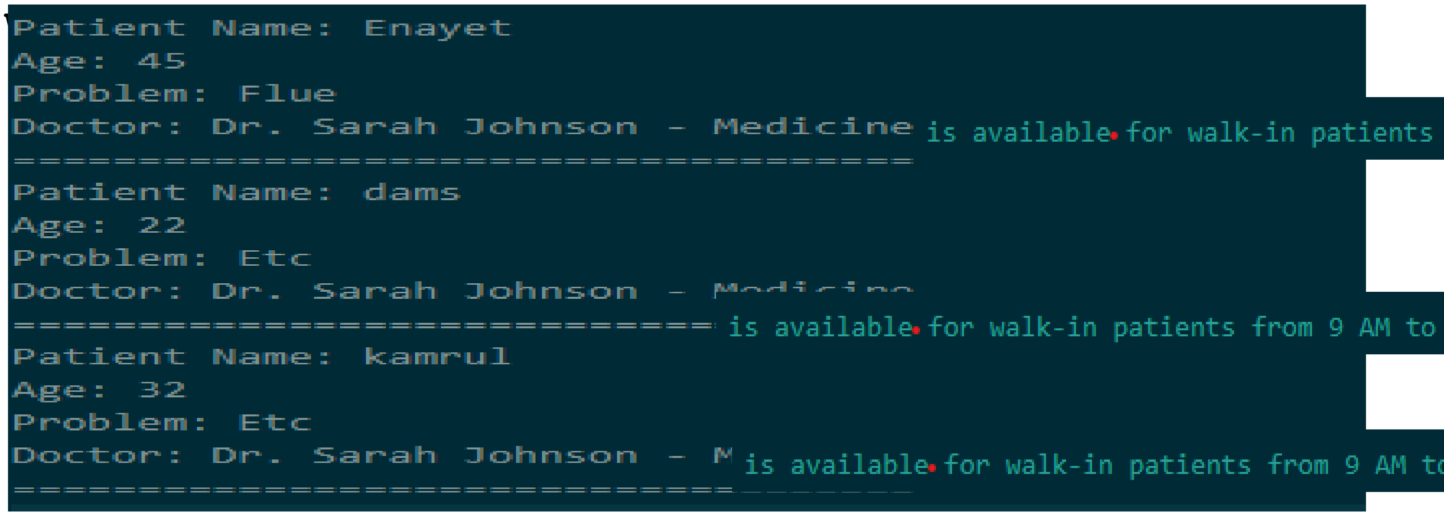


```
=== Clinic Management System ===
1. Find Your Doctor
2. View Patient List by Doctor
3. Back to Main Menu
Enter your choice: 2
Enter Doctor's Name: Sarah Johnson
```

And show his patient list

❖

#There are 12 doctors details save in DoctorsDetails() class



```
Patient Name: Enayet
Age: 45
Problem: Flue
Doctor: Dr. Sarah Johnson - Medicine is available for walk-in patients
====================================
Patient Name: dams
Age: 22
Problem: Etc
Doctor: Dr. Sarah Johnson - Medicine
============================= is available for walk-in patients from 9 AM to
Patient Name: kamrul
Age: 32
Problem: Etc
Doctor: Dr. Sarah Johnson - M is available for walk-in patients from 9 AM to
====================================
```

The Clinic Management System presented here is a comprehensive application designed to streamline the operations of a medical clinic by efficiently managing doctors, patients, and appointments. This system employs object-oriented programming principles such as inheritance, encapsulation, and polymorphism, providing a robust and scalable solution.

1. Doctor Management: The system categorizes doctors into different specialties.
2. Patient Management: The system facilitates the storage and retrieval of patient information associated with specific doctors. Patients can be linked to doctors based on their specialty, and their information is stored and managed efficiently, highlighting the system's use of classes and data encapsulation.
3. Appointment Scheduling and Patient Record Maintenance: The system provides functionalities for recording patient details and managing patient-doctor associations.
4. User Interaction through a Simple Interface: The system features a user-friendly interface that allows users to navigate various functionalities, such as finding doctors, selecting them, and viewing associated patient lists. This makes it accessible and easy to use for clinic staff and patie

Future Improvements:

- To expand the functionality of the Clinic Management System to include features such as adding and deleting doctors, as well as adding and deleting patients, you can modify the existing classes and add new classes to accommodate these operations.

At last the Clinic Management System serves as a foundational tool for clinics, offering a structured approach to managing healthcare services

.