Kingston University, BSc (Hons) (top-up)

Coursework Coversheet

Draft Coursework – Subject to Moderation

Part 1 - To Remain with the Assignment after Marking

Student ID: KAN00217052	**Student Name: A.A.M Aathif**
Module Code:	**Module Name: Programming - Pattern & Algorithms**
Assignment number:	**ESoft Module Leader:**
Date set:2024/10/19	**Date due: 2024/11/09**

Guidelines for the Submission of Coursework

1. Print this coversheet and securely attach both pages to your assignment. You can help us ensure work is marked more quickly by submitting at the specified location for your module. You are advised to keep a copy of every assignment.

2. Coursework deadlines are strictly enforced by the University.

3. You should not leave the handing in of work until the last minute. Once an assignment has been submitted it cannot be submitted again.

Academic Misconduct: **Plagiarism** and/or **collusion** constitute **academic misconduct** under the University's Academic Regulations. Examples of academic misconduct in coursework: making available your work to other students; presenting work produced in collaboration with other students as your own (unless an explicit assessment requirement); submitting work, taken from sources that are not properly referenced, as your own. By printing and submitting this coversheet with your coursework you are confirming that the work is your own.

ESoft Office Use Only: Date stamp: work received

Kingston University, BSc (Hons) (top-up)

Coursework Coversheet

Part 2 – Student Feedback

Student ID: KAN00217052	**Student Name: A.A.M Aathif**
Module Code:	**Module Name: Programming - Pattern & Algorithms**
Assignment number:	**ESoft Module Leader:**
Date set: 2024/10/19	**Date due: 2024/11/09**

Strengths (areas with well-developed answers)

Weaknesses (areas with room for improvement)

Additional Comments

ESoft Module Lecturer: Vikum Jayasundara

ESoft Module Marker:

Provisional mark as %:

Date marked:

CI6115 PROGRAMMING III - PATTERNS AND ALGORITHMS	
Weighting	50%
Description	The aim of this coursework is to develop and document a software application using any Object-Oriented Programming (OOP) language and compatible tools of any version. The application can be either console-based or GUI-based and should address the specified case study.
Expected Deliverables	The submission must include the following three parts in a single document, along with all project files uploaded as a separate compressed folder: **Class Diagram and Discussion**: • A comprehensive class diagram for the proposed system. • Accompanying discussion that highlights the key design decisions and rationale behind them. **Source Code**: • Complete source code of the proposed system, demonstrating the use of: i. Classes and objects ii. Object-Oriented Programming concepts iii. Data structures iv. Relevant algorithms **Test Cases**: • A full set of test cases used to validate the system.

Case Study

Aurora Skin Care is a medium-scale, affordable private skin clinic situated in Colombo to provide dermatological services to out-patients who are unable to receive medical facilities from the expensive skin clinics. There are two dermatologists visiting the clinic and the available consultation dates and times are as follows:

- Monday: 10:00am - 01:00pm
- Wednesday: 02:00pm - 05:00pm
- Friday: 04:00pm - 08:00pm
- Saturday: 09:00am - 01:00pm

Note that one dermatologist requires a 15-minute session per patient, and the selection of the dermatologist is based on the patient's preferences.

The front desk operator is responsible for booking appointments for patients by scanning through doctors' consultation schedules for the date(s) requested by a patient and then informing the patient of the available options. If the patient is satisfied with a specific appointment time, they can reserve the appointment for that date and time by paying a registration fee of LKR 500 and providing patient information such as NIC, name, email, and telephone number.

Once the patient has received the treatment, the final fee is calculated according to the treatment type. A 2.5% tax is added to the final amount, which is then rounded up to the nearest decimal number. The treatment types with fixed prices are listed below:

Treatment	Price
Acne Treatment	2750.00
Skin Whitening	7650.00
Mole Removal	3850.00
Laser Treatment	12 500.00

The Proposed system should perform the following functions:

1. Make appointments
2. Update appointment details.
3. View appointment details filtered by date.
4. Search for an appointment using patient name or appointment ID.
5. Accept registration fee when placing appointments.
6. Calculate the total fee and taxes for the treatments after the appointment.
7. Generate an invoice for the payment, clearly stating how the total was calculated.

Further information and reminders:

- **Class Diagram**: Use any standard UML tool to draw the class diagram. Ensure that the images are clear. Watermarks are acceptable if they are not obtrusive. **Each class name in the diagram must be prefixed with your Kingston University student ID, otherwise, marks will be deducted**.
- **Programming Language**: You may select any object-oriented programming language, but Java is preferable.
- **Frameworks**: Use frameworks with careful evaluation. If a framework generates all the classes for you, marks will be deducted.
- **User Interfaces**: User interfaces can be either Command Line Interfaces (CLI) or Graphical User Interfaces (GUI). Evaluate the user experience in terms of UI/UX for the chosen interface type. The choice of interface type will not impact grading; the focus should be on the quality of implementation.
- **Screenshots**: Include clear screenshots of the code and user interfaces, following the detailed instructions provided.
- **Test Cases**: Use a standard test case template. General testing procedures are sufficient; there is no need to follow higher standards or use automation mechanisms.

Marking Scheme

Item		Marks Allocation
Class Diagram (20)		
1	Identification of classes & objects	05
2	Application of objects, fields, and methods	05
3	Use of associations effectively	05
4	Sensible naming of programmer-defined variables, classes, properties, and methods	05
Software Implementation (60)		
1	Completion of functional modules	30
2	Effective use of object-oriented programming concepts	10
3	Effective use of data structures	05
4	Effective use of algorithms	05
5	Use of Test Cases	10
Programming Style (20)		
1	Clarity of code which shows the underlying algorithm	05
2	Sensible naming of programmer-defined variables, classes, properties, and methods	05
3	Useful comments in code	05
4	User interface design and usability of the system	05
Total		**100**

Academic Integrity:

Academic integrity means demonstrating honest, moral behaviours when producing academic work. This involves acknowledging the work of others, giving appropriate credit to others where their ideas are presented as part of your work and the importance of producing work in your own voice. Contributions by artificial intelligence (AI) tools must be properly acknowledged. As part of a learning community students share ideas and develop new ones - you need to be able to interpret and present other people's ideas and combine these with your own when producing work.

Plagiarism (including copying, self-plagiarism and collusion)

The act of presenting the work of another person (or people) and/or content generated by artificial intelligence (AI) tools as your own without proper acknowledgement. This includes copying the work of another student or other students. The University expects students to take responsibility for the security of their work (i.e. with written work, to ensure that other students do not get access to electronic or hard copy of the work). Failure to keep work secure may allow others to cheat and could result in an allegation of academic misconduct for students whose work have been copied, particularly if the origin of the work is in doubt.

Self-plagiarism

The act of presenting part or all of your work that has been previously submitted to meet the requirements of a different assessment, except where the nature of the assessment makes this permissible.

Collusion

The act, by two or more students of presenting a piece of work jointly without acknowledging the collaboration. This could include permitting or assisting another to present work that has been copied or paraphrased from your own work. The University also defines collusion as the act of one student presenting a piece of work as their own independent work when the work was undertaken by a group. With group work, where individual members submit parts of the total assignment, each member of a group must take responsibility for checking the legitimacy of the work submitted in his/her name. If even part of the work is found to contain academic misconduct, penalties will normally be imposed on all group members equally.

Purchasing or Commissioning

The act of attempting to purchase or purchasing work for an assessment including, for example from the internet, or attempting to commission, or commissioning someone else to complete an assessment on your behalf. The procedures for investigating suspected cases of academic misconduct are set out in Academic Regulations 6 Academic Integrity - Taught Courses 2023/24

Acknowledging Generative AI in coursework

Where generative AI has contributed to an assignment the following information should be included in the submission: A statement on the use of generative AI as part of the assessment, including the extent of use, and how it was used as part of all stages in creating the final submission, e.g., including planning, and generating ideas. This should normally be provided at the end of a written assignment with the heading 'Acknowledgement of AI Contribution'. For other assignment types, module staff will advise on how this should be done.

You must meet all deadlines set. Failure to do so will result in a penalty.

Work submitted late but within a week of the deadline will be capped at 40% and receive a grade of LP (Late Pass) unless it is not of a passing standard in which case it will receive a grade of LF (Late Fail). Work submitted beyond a week of the deadline without approval will get 0% with a grade of F0. If, however, you have a serious problem, which prevents you from, meeting the deadline you may be able to negotiate an extension in advance. In the first instance you should contact the module team for advice. However, any extension will need to be formally agreed by the Faculty via the Mitigating Circumstances process, your work will then be marked without penalty.

Table of Contents

List of Figure

List of Tables

Acknowledgment

I would like to express my sincere gratitude to **Mr. VIKUM JAYASUNDARA** for his invaluable guidance and support throughout this coursework. His insightful lectures and constructive feedback were instrumental in shaping my understanding of patterns and algorithms. I would also like to thank my family and friends for their unwavering support and encouragement. Their belief in me motivated me to persevere through challenges and strive for excellence.

I am grateful to my peers for the collaborative learning experiences and for sharing their knowledge and insights. Their contributions enriched my understanding of the subject matter. I would like to acknowledge the support of Kingston University and its staff for providing the necessary resources and assistance.

Activity 01 – System Overview and Requirements

1.1 Introduction

Aurora Skin Care is a medium-scale, affordable private skin clinic located in Colombo, designed to provide accessible dermatological services to outpatients unable to afford premium services at higher-end clinics. With a team of two experienced dermatologists and a schedule that accommodates various time slots throughout the week, Aurora aims to offer high-quality, affordable skincare.

The clinic offers a simple and efficient appointment system facilitated by a front desk operator who manages **Scheduling, Patient Data, and Fee Processing**. Patients can choose their preferred dermatologist and reserve appointments based on their availability. To secure their appointments, patients must provide basic identification details and pay a registration fee of LKR 500.

Aurora Skin Care's services include a variety of standard treatments such as **Acne Treatment, Skin Whitening, Mole Removal, and Laser Treatment,** each with a fixed price. Upon completion of a treatment, taxes are applied to the final bill and rounded appropriately to provide clarity and transparency. The proposed system for Aurora Skin Care will manage **Appointment Bookings, Patient Data, Payments,** and invoicing, with an emphasis on creating an organized and user-friendly experience for both patients and clinic staff.

This project aims to develop a reliable software solution tailored to the unique needs of Aurora Skin Care, incorporating **Object-Oriented Programming Principles** to enhance the efficiency and scalability of clinic operations. The proposed system will streamline key functions such as **Appointment Scheduling, Patient Information Management, and Payment Processing**. By implementing robust features like **Search and Filter Options for Appointments**, **Automated Calculation of Final Treatment Fees with Applicable Taxes**, and **Detailed Invoice Generation**, the solution will provide a **seamless experience** for both the clinic's staff and patients.

1.2 Requirements Analysis

1.2.1 Functional Requirements

1. **Appointment Scheduling**
   - Allow the front desk operator to schedule appointments by reviewing dermatologist availability on specific dates and times.
   - Enable patients to select preferred time slots based on their choice of dermatologist.

2. **Update Appointment Details**
   - Allow operators to modify appointment times or details if needed by patients or doctors.
   - Enable cancellation or rescheduling of appointments when necessary.

3. **View Appointment Details by Date**
   - Provide the ability to filter and view scheduled appointments for a specific date, which assists in managing daily operations.

4. **Search for Appointment**
   - Allow operators to search for an appointment by **patient name** or **appointment ID** to quickly locate and manage bookings.

5. **Registration Fee Processing**
   - Process a **registration fee** of LKR 500 to confirm appointment bookings.
   - Store patient details (NIC, name, email, phone number) during registration.

6. **Treatment Billing**
   - Calculate the total fee for treatments post-appointment, including a **2.5% tax** added to the base treatment cost.
   - Round up the final calculated fee to ensure clear billing.

7. **Invoice Generation**
   - Generate a clear invoice for each patient's treatment, including registration fee, treatment type, base cost, taxes, and total amount.

1.2.2 Non-Functional Requirements

1. **Usability**
   - Provide an easy-to-use interface for front desk operators with clearly labeled fields for appointment scheduling, patient registration, and billing.
   - The design should ensure that operators can navigate quickly between screens with minimal training.

2. **Performance**
   - Ensure fast response times for scheduling, searching, and viewing appointments to avoid delays in a busy clinic environment.
   - Generate invoices and calculate taxes swiftly to ensure patients experience minimal waiting time.

3. **Scalability**
   - The system should support additional functionalities in the future, such as adding new treatment types, adjusting fees, or updating scheduling details.
   - Design should allow for the easy addition of new dermatologists or expansion of available appointment slots.

4. **Security**
   - Protect sensitive patient information (e.g., NIC, contact information) through encryption and secure data storage practices.
   - Ensure that only authorized staff have access to patient details and financial transactions.

5. **Maintainability**
   - The system should be modular and well-documented, making it easy for future developers to update or modify the code as required.
   - Logs should be implemented to help track system issues, appointment changes, or billing discrepancies.

Activity 02 – System Design and Implementation

2.1 Class Diagram

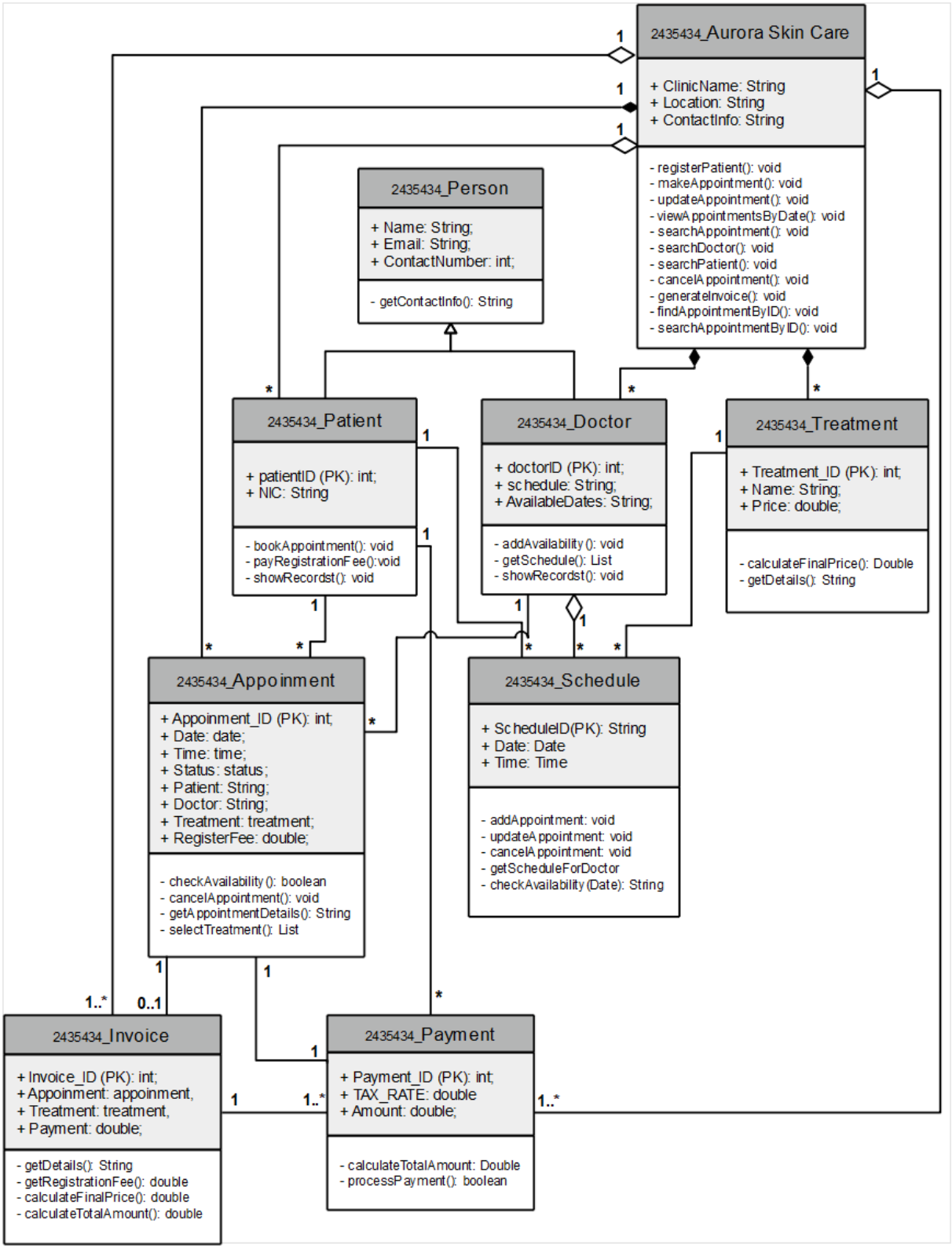


Figure 1 Class Diagram of Aurora Clinic Care System

2.1.1 How System Works Based on this UML Diagram

In the Aurora Skin Care system, the appointment and payment process are designed to provide a seamless experience for both patients and doctors. For example, suppose a patient named **Daenerys Stormborn** wants to schedule a skin treatment at Aurora Skin Care. Daenerys registers as a patient in the system, providing her contact information. She then checks the availability of her preferred doctor, **Dr. Ijlan (D001)**, by viewing Dr. Ijlan's schedule in the system. After finding a suitable date, Daenerys uses the **bookAppointment** method to make an appointment with Dr. Ijlan for a treatment. This booking is managed by the Appointment class, which stores details such as the appointment date, time, doctor, and a registration fee.

On the scheduled date, Daenerys arrives at the clinic, and Dr. Ijlan performs her chosen treatment. Suppose she opted for **Skin Whitening**, priced at **7,650.00**. The treatment details, including the name and price, are recorded in the system under the Treatment class. After the treatment, the payment process begins. The system calculates the total fee, including any applicable taxes and additional charges, using the **calculateTotalAmount** method in the Payment class. Daenerys can then proceed to pay her bill, which the system processes using the **processPayment** method.

Invoice is generated, detailing the cost breakdown of Daenerys's treatment, providing her with a clear record of services rendered. This workflow, supported by the Aurora Skin Care system, ensures efficient management of appointments, treatments, and payments, enhancing the overall experience for both patients and clinic staff.

2.1.2 Class Overview

2.1.2.1 Aurora Skin Care

- **Purpose**: Represents the clinic itself, handling patient registration, appointment scheduling, doctor management, and invoice generation.
- **Methods**:
  - registerPatient(): Registers a new patient.
  - makeAppointment(): Creates a new appointment.
  - updateAppointment(): Modifies an existing appointment.
  - viewAppointmentsByDate(): Lists all appointments scheduled for a specific date.
  - searchAppointment(): Searches for appointments based on certain criteria.
  - searchPatient(): Searches for a specific patient in the system.
  - generateInvoice(): Creates an invoice for an appointment.
  - findAppointmentByID(): Finds an appointment using a unique ID.
  - searchAppointmentByID(): Searches for an appointment using its ID.
- **Relationships**:
  - **Aurora Skin Care → Patient**: One-to-many association, as the clinic can have multiple patients.
  - **Aurora Skin Care → Doctor**: One-to-many association, allowing the clinic to manage multiple doctors.
  - **Aurora Skin Care → Appointment**: One-to-many association, enabling the clinic to manage multiple appointments.
  - **Aurora Skin Care → Treatment**: One-to-many association, representing the various treatments available at the clinic.
  - **Aurora Skin Care → Invoice**: One-to-many association, allowing the clinic to generate invoices for each appointment or treatment.
- **Design Decision**: The choice to centralize operations like registration, appointment handling, and invoice generation within this class makes it an essential controller, enabling smooth management of all high-level actions.

- **Rationale**: This design simplifies the system's core functionalities by placing them in one class, which facilitates efficient data handling and processing across all entities.

2.1.2.1 Person

- **Purpose**: Serves as a base class for individuals in the system, including patients and doctors, containing common information.
- **Methods**:
  - getContactInfo(): Returns the contact details of the person.
- **Relationships**:
  - **Person → Patient**: Inheritance, as Patient is a specialized form of Person.
  - **Person → Doctor**: Inheritance, as Doctor is a specialized form of Person.
- **Design Decision**: By abstracting common attributes and methods in this superclass, such as getContactInfo(), the system reduces redundancy and improves reusability.

- **Rationale**: This design decision aligns with principles of object-oriented design by promoting inheritance and maintaining cleaner, more maintainable code.

2.1.2.2 Patient

- **Purpose**: Represents a patient, storing their unique identifiers and managing their interactions with the clinic.
- **Methods**:
  - bookAppointment(): Allows the patient to book an appointment.
  - payRegistrationFee(): Processes the patient's registration fee.
  - showRecords(): Displays the patient's medical records or appointment history.
- **Relationships**:
  - **Patient → Appointment**: One-to-many association, as a patient can book multiple appointments.
  - **Patient → Aurora Skin Care**: Managed through the clinic's relationship, as Aurora Skin Care oversees all patient activities.
- **Design Decision**: The inclusion of these methods supports patient-specific actions, while inheriting basic details from **Person**.

- **Rationale**: Separating patient functionality into its own class makes it easier to manage patient interactions without complicating the **Person** class.

2.1.2.3 Doctor

- **Purpose**: Represents a doctor, storing their unique identifiers and schedule.
- **Methods**:
  - addAvailability(): Adds availability slots for the doctor.
  - getSchedule(): Retrieves the doctor's schedule.
  - showRecords(): Displays the doctor's records or availability.
- **Relationships**:
  - **Doctor → Appointment**: One-to-many association, as a doctor can have multiple appointments.
- **Design Decision**: By giving doctors control over their availability and allowing patient consultation through the **Schedule** class.

- **Rationale**: Doctors require unique functionalities that differ from patients, so it's practical to maintain them as a separate class while allowing interaction through the shared **Schedule** class.

2.1.2.4 Schedule

- **Purpose**: Manages scheduling details, including dates and times, primarily for appointments and doctor availability.
- **Methods**:
  - addAppointment(): Adds an appointment to the schedule.
  - updateAppointment(): Updates details of a scheduled appointment.
  - checkAvailability(Date): Checks if a specific date is available.
- **Relationships**:
  - **Schedule → Doctor**: Many-to-one association, allowing each doctor to have multiple schedules.
- **Design Decision**: The separation of scheduling functionality from **Doctor** enables a more modular approach, allowing scheduling logic to be handled independently.

- **Rationale**: Keeping scheduling in a separate class allows for flexible date and time management, supporting potential future expansions like allowing multiple doctors with different schedules.

2.1.2.1 Appointment

- **Purpose**: Manages appointment details, including the date, time, patient, doctor, and treatment.
- **Methods**:
  - checkAvailability(): Checks if the appointment slot is available.
  - cancelAppointment(): Cancels the appointment.
  - getAppointmentDetails(): Provides details of the appointment.
  - selectTreatment(): Allows selection of a treatment for the appointment.
- **Relationships**:
  - **Appointment → Patient**: Many-to-one association, as each appointment is linked to one patient.
  - **Appointment → Doctor**: Many-to-one association, as each appointment is linked to one doctor.
  - **Appointment → Treatment**: One-to-one association, as each appointment involves a specific treatment.
  - **Appointment → Schedule**: One-to-one association, linking each appointment with a specific schedule.
- **Design Decision**: Placing all appointment-related details in one class allows for centralized tracking and management of appointments.

- **Rationale**: This class is crucial for maintaining the scheduling system, enabling easy retrieval, modification, and cancellation of appointments, thereby streamlining workflow.

2.1.2.2 Treatment

- **Purpose**: Represents different treatments offered at the clinic, including cost details.
- **Methods**:
  - calculateFinalPrice(): Calculates the final price for the treatment.
  - getDetails(): Returns details about the treatment.
- **Relationships**:
  - **Treatment → Appointment**: One-to-one association, as each appointment may involve a single treatment.
- **Design Decision**: By consolidating treatment details in one class, the system can easily manage and retrieve treatment information.

- **Rationale**: Centralizing treatments ensures consistency, especially for price calculations and retrieving detailed information, helping maintain accuracy in billing.

2.1.2.3 Payment

- **Purpose**: Manages financial transactions related to appointments and treatments.
- **Methods**:
  - calculateTotalAmount(): Calculates the total amount owed for services.
  - roundOffAmount(): Rounds the total amount to ensure accurate billing.
  - processPayment(): Processes the payment and validates its success or failure.
- **Relationships**:
  - **Payment → Appointment**: One-to-zero-or-one association, as each payment may be linked to a specific appointment.
  - **Payment → Invoice**: One-to-one association, as each payment is associated with a specific invoice.
- **Design Decision**: Placing all payment-related operations in one class allows the system to handle transaction processing in a centralized and standardized way.

- **Rationale**: Isolating payment functionality makes it easier to adjust payment processing rules and ensures consistency in transaction handling.

2.1.2.4 Invoice

- **Purpose**: Represents billing details for appointments and treatments, including registration fees and total charges.
- **Methods**:
  - getDetails(): Returns details of the invoice.
  - getRegistrationFee(): Retrieves the registration fee amount.
  - calculateFinalPrice(): Calculates the final price of the treatment or appointment.
  - calculateTotalAmount(): Calculates the total amount due on the invoice.
- **Relationships**:
  - **Invoice → Appointment**: One-to-one association, as each invoice corresponds to a specific appointment.
  - **Invoice → Treatment**: One-to-one association, as each invoice includes the treatment cost.
  - **Invoice → Payment**: One-to-one association, linking each invoice to a specific payment.
- **Design Decision**: Separating invoices from **Payment** supports modularity, allowing the **Invoice** class to handle detailed billing and calculation of the final payable amount independently.

- **Rationale**: This design decision aligns with good practices by decoupling financial calculations and payment processing, making the system more adaptable and manageable.

2.2 System Design

2.2.1 Main Menu of the Aurora Skin Care System

the Aurora Skin Care Clinic system offers a convenient and efficient way to manage their appointments and interact with the clinic. The menu provides easy access to essential functions like booking appointments, updating information, viewing appointment details, and searching for specific information. This system can save users time and effort by eliminating the need for phone calls or in-person visits for routine tasks.

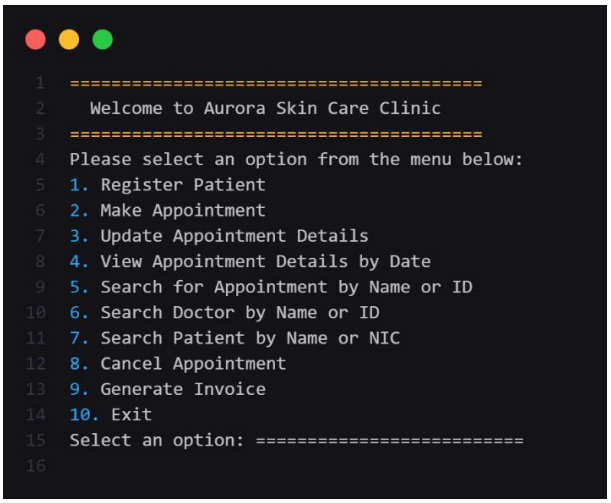


Figure 2 CLI Snap 01 - Main Menu of the system

2.2.2 Patient Registration

The user selects the "Register New Patient" option from the menu and is then prompted to enter the patient's name, email address, contact number, and NIC (National Identity Card) number. Once the information is entered, the system displays a message confirming that the patient has been registered successfully.

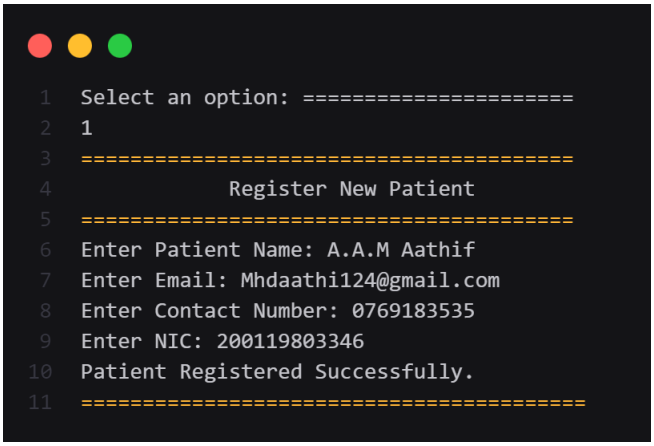


Figure 3 CLI Snap 02 - Patient Registration

2.2.3 Make Appointment

The appointment booking process appears to be straightforward and user-friendly. The clear prompts guide the user through the necessary steps, making it easy to input the required information. The confirmation message provides immediate feedback, assuring the user that the appointment was booked successfully. The system also provides a clear summary of the appointment details, including the appointment ID, date, time, doctor, treatment type, and price.

The user selects the "Make a New Appointment" option from the menu and is then prompted to enter the patient's NIC (National Identity Card) number, desired date and time, and choose a doctor and treatment type from the available options. Once the information is entered, the system displays a message confirming that the appointment has been booked successfully. In this specific case, the user A.A.M Aathif booked an appointment with Dr. Brian for Skin Whitening treatment, which costs LKR 7650.00.

```
1  ========================================
2              Make a New Appointment
3  ========================================
4  Enter Patient NIC: 200119803346
5  Enter Date (| Mon | Wed | Fri | Sat |): Wednesday
6  Enter Time (e.g., 10:00am): 11.00am
7
8  Select Doctor:
9  1: Dr. Ijlan
10 2: Dr. Brian
11 Select Doctor (1-2): 2
12
13 Select Treatment Type:
14 1: Name: Acne Treatment, Price: LKR 2750.0
15 2: Name: Skin Whitening, Price: LKR 7650.0
16 3: Name: Mole Removal, Price: LKR 3850.0
17 4: Name: Laser Treatment, Price: LKR 12500.0
18 Select Treatment (1-4): 2
19
20 Appointment booked successfully.
21 Appointment Details: Appointment ID: 1, Date: Wednesday, Time: 11.00am,
22 Status: BOOKED, Patient: A.A.M Aathif, Doctor: Dr. Brian,
23 Treatment: Name: Skin Whitening, Price: LKR 7650.0
24 ========================================
```

Figure 4 CLI Snap 03 - Make Appointment

2.2.4 Update Appointment by Date, Time

The user selects the "Update Appointment Details" option from the menu and is then prompted to enter the appointment ID to be updated, the New Date, and the New Time. Once the information is entered, the system displays a message confirming that the appointment has been updated successfully. In this specific case, the user successfully changed the appointment date from Monday to Saturday and the time from 11:00 AM to 1:00 PM.

```
1  ========================================
2         Update Appointment Details
3  ========================================
4  Enter Appointment ID to update: 1
5  Enter New Date (| Mon | Wed | Fri | Sat |): Saturday
6  Enter New Time (e.g., 10:00am): 01.00pm
7
8  Appointment updated to Date: Saturday, Time: 01.00pm
9  Appointment updated successfully.
10 Updated Appointment Details: Appointment ID: 1, Date: Saturday,
11 Time: 01.00pm, Status: BOOKED, Patient: A.A.M Aathif,
12 Doctor: Dr. Brian, Treatment: Name: Skin Whitening, Price: LKR 7650.0
13 ========================================
```

Figure 5 CLI Snap 04 - Update Appointment by Date, Time

2.2.5 View Appointment by Date

The user selects the "View Appointments by Date" option from the menu and is then prompted to enter the desired date to filter the appointments. Once the date is entered, the system displays a list of appointments scheduled for that date. In this specific case, the user entered "Saturday" and the system displayed the details of an appointment with ID 1, scheduled for Saturday at 1:00 PM with Dr. Brian for Skin Whitening treatment, which costs LKR 7650.0.

```
1  Select an option: ========================================
2  4
3  ========================================
4           View Appointments by Date
5  ========================================
6  Enter Date (| Mon | Wed | Fri | Sat |) to filter appointments: Saturday
7
8  Appointment ID: 1, Date: Saturday, Time: 01.00pm, Status: BOOKED,
9  Patient: A.A.M Aathif, Doctor: Dr. Brian,
10 Treatment: Name: Skin Whitening, Price: LKR 7650.0
11 ========================================
```

Figure 6 CLI Snap 05 - View Appointment by Date

2.2.6 Search Appointment by Patient Name / Appointment ID

2.2.6.1 Search by Appointment ID

The user selects the "Search Appointment by Patient Name / Appointment ID" option from the menu and is then prompted to enter either the appointment ID or the patient name to search for the appointment details. In this specific case, the **user entered "1" as the appointment ID** and the system displayed the details of the appointment with ID 1, scheduled for Saturday at 1:00 PM with Dr. Brian for Skin Whitening treatment, which costs LKR 7650.0.

```
Select an option: ========================
5
=========================================
 Search Appointment by Patient Name / Appointment ID
=========================================
Enter Appointment ID or Patient Name to search: 1

Appointment ID: 1, Date: Saturday, Time: 01.00pm, Status: BOOKED,
Patient: A.A.M Aathif, Doctor: Dr. Brian,
Treatment: Name: Skin Whitening, Price: LKR 7650.0
=========================================
```

Figure 7 CLI Snap 06 - Search Appointment by Appointment ID

2.2.6.2 Search by Patient Name

The user selects the "Search Appointment by Patient Name / Appointment ID" option from the menu and is then prompted to enter either the appointment ID or the patient name to search for the appointment details. In this specific case, the **user entered "A.A.M Aathif" as the patient name** and the system displayed the details of the appointment with ID 1, scheduled for Saturday at 1:00 PM with Dr. Brian for Skin Whitening treatment, which costs LKR 7650.0.

```
Select an option: ========================
5
=========================================
 Search Appointment by Patient Name / Appointment ID
=========================================
Enter Appointment ID or Patient Name to search: A.A.M Aathif

Appointment ID: 1, Date: Saturday, Time: 01.00pm, Status: BOOKED,
Patient: A.A.M Aathif, Doctor: Dr. Brian,
Treatment: Name: Skin Whitening, Price: LKR 7650.0
=========================================
```

Figure 8 CLI Snap 07 - Search Appointment by Patient Name

2.2.7 Search Doctor

2.2.7.1 Search by Doctor ID

The user selects the "Search for a Doctor" option from the menu and is then prompted to enter either the doctor's name or ID to search for the doctor's details. In this specific case, the user entered "D001" as the doctor's ID and the system displayed the details of the doctor with Employee ID D001, name Dr. Ijlan, email mohamedijlan02@gmail.com, and contact number 0776778795.

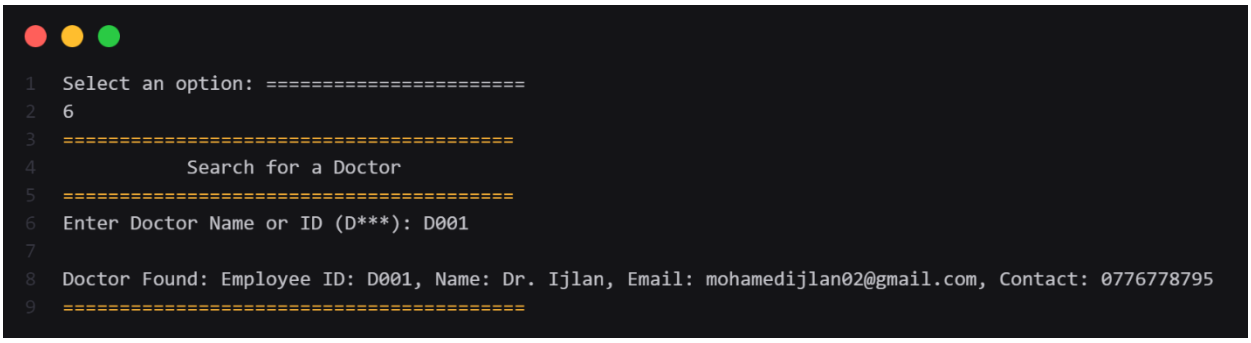


Figure 9 CLI Snap 08 - Search Doctor by Doctor ID

2.2.7.2 Search by Doctor Name

The user selects the "Search for a Doctor" option from the menu and is then prompted to enter either the doctor's name or ID to search for the doctor's details. In this specific case, the user entered "Dr. Brian" as the doctor's name and the system displayed the details of the doctor with Employee ID D002, name Dr. Brian, email jacobmichaelbrian01@gmail.com, and contact number 0764517561.

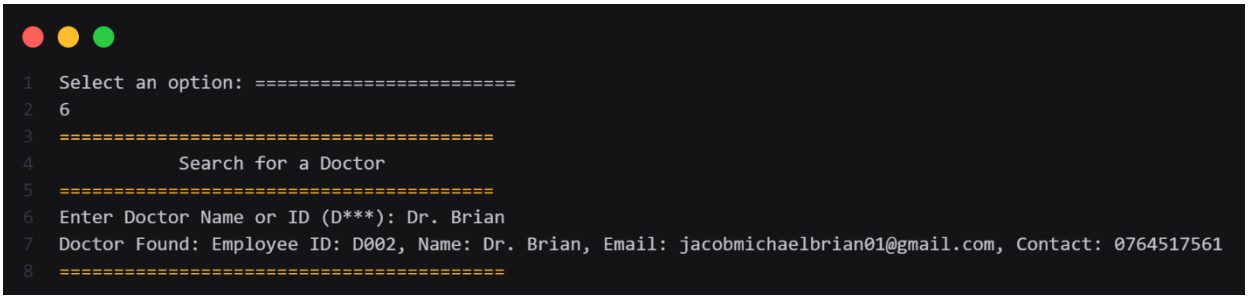


Figure 10 CLI Snap 09 - Search Doctor by Doctor Name

2.2.8 Search Patient by Name / NIC

2.2.8.1 Search by NIC

The user selects the "Search for a Patient" option from the menu and is then prompted to enter either the patient's name or NIC (National Identity Card) number to search for the patient's details. In this specific case, the user entered "200119803346" as the NIC number and the system displayed the details of the patient with name A.A.M Aathif, email Mhdaathi124@gmail.com, contact number 0769183535, and NIC 200119803346.

```
Select an option: ======================
7
========================================
          Search for a Patient
========================================
Enter Patient Name or NIC to search: 200119803346

Patient Found: Name: A.A.M Aathif, Email: Mhdaathi124@gmail.com, Contact: 0769183535, NIC: 200119803346
========================================
```

Figure 11 CLI Snap 10 - Search Patient by NIC

2.2.8.2 Search by Patient Name

The user selects the "Search for a Patient" option from the menu and is then prompted to enter either the patient's name or NIC (National Identity Card) number to search for the patient's details. In this specific case, the user entered "A.A.M Arshak" as the patient name and the system displayed the details of the patient with name A.A.M Arshak, email Mohamedarshak48@gmail.com, contact number 0778211464, and NIC 200335600465.

```
Select an option: ========================================
7
========================================
          Search for a Patient
========================================
Enter Patient Name or NIC to search: A.A.M Arshak

Patient Found: Name: A.A.M Arshak, Email: Mohamedarshak48@gmail.com, Contact: 0778211464, NIC: 200335600465
========================================
```

Figure 12 CLI Snap 11 - Search Patient by Patient Name

2.2.9 Cancel Appointment

The user selects the "Cancel Appointment" option from the menu and is then prompted to enter the appointment ID to be canceled. In this specific case, the user entered "1" as the appointment ID and the system displayed a message confirming that the appointment with ID 1 has been canceled. This indicates that the appointment for A.A.M Aathif has been successfully canceled.

```
Select an option: ======================
8
========================================
            Cancel Appointment
========================================
Enter Appointment ID to cancel: 1

Appointment canceled for: A.A.M Aathif
========================================
```

Figure 13 CLI Snap 12 - Cancel Appointment

2.2.10 Generate Invoice

The user selects the "Generate Invoice" option from the menu and is then prompted to enter the appointment ID and select the treatment type. In this specific case, the user entered "1" as the appointment ID and selected "2" for Skin Whitening treatment. The system will then generate an invoice for the specified appointment and treatment.

```
Select an option: ======================================
9
======================================
           Generate Invoice
======================================
Enter Appointment ID: 1
Select Treatment Type (1: Acne Treatment, 2: Skin Whitening, 3: Mole Removal, 4: Laser Treatment): 2
```

Figure 14 CLI Snap 13 - Generate Invoice

2.2.11 Invoice

The invoice shows the details of the appointment made by A.A.M Aathif on Saturday at 1:00 PM for Skin Whitening treatment. The total cost of the appointment, including the treatment fee, registration fee, and tax.

- **Tax (2.5%):** LKR 191.25 (Calculated as 2.5% of the total amount: (7650 + 500) * 0.025)
- **Total:** LKR 7841.25

(Calculated as the sum of treatment fee, registration fee, and tax: 7650 + 500 + 191.25)

The invoice also includes a thank you message from the clinic.

```
1  ================================
2                INVOICE
3  ================================
4  Appointment ID:    1
5  Patient:           A.A.M Aathif
6  Date:              Saturday
7  Time:              01.00pm
8  --------------------------------
9  Name: Skin Whitening: LKR 7650.0
10 Registration Fee:     LKR 500.00
11 Tax (2.5%):           LKR 191.25
12 --------------------------------
13 Total:                LKR 7841.25
14 ================================
15 Thank you for choosing our services!
16 ================================
```

Figure 15 CLI Snap 14 - Invoice

2.2.12 Exit

The user selects the "Exit" option from the menu, indicated by entering "10". The system then displays a farewell message, "Exiting the system. Goodbye!", confirming the user's exit from the system.

```
1  Select an option: ==========
2  10
3  Exiting the system. Goodbye!
```

Figure 16 CLI Snap 15 - Exit

Activity 03 – Implementation

3.1 The Implementation of the Aurora Skin Care

3.1.1 Main Program Class

```java
// Main Program Class
// ! This class serves as the entry point for the Aurora Skin Care Clinic.
public class AuroraSkinCareSystem {
    static List<Patient> patients = new ArrayList<>();
    static List<Doctor> doctors = new ArrayList<>();
    static List<Appointment> appointments = new ArrayList<>();
    static List<Treatment> availableTreatments = new ArrayList<>();
    static int invoiceCounter = 1; // * Counter for generating unique invoice IDs

    // ! Main method to run the application
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // * Scanner for user input

        // ! Manually add initial doctors
        doctors.add(new Doctor("Dr. Ijlan", "mohamedijlan02@gmail.com", "0776778795", "D001"));
        doctors.add(new Doctor("Dr. Brian", "jacobmichaelbrian01@gmail.com", "0764517561", "D002"));

        availableTreatments.add(new Treatment(1, "Acne Treatment", 2750.00));
        availableTreatments.add(new Treatment(2, "Skin Whitening", 7650.00));
        availableTreatments.add(new Treatment(3, "Mole Removal", 3850.00));
        availableTreatments.add(new Treatment(4, "Laser Treatment", 12500.00));

        // ! Handle user options with a switch statement
        while (true) {
            System.out.println("\n" + "=".repeat(40));
            System.out.println("       Welcome to Aurora Skin Care Clinic       ");
            System.out.println("=" + "=".repeat(38) + "=");
            System.out.println("Please select an option from the menu below:");
            System.out.println("1. Register Patient");
            System.out.println("2. Make Appointment");
            System.out.println("3. Update Appointment Details");
            System.out.println("4. View Appointment Details by Date");
            System.out.println("5. Search for Appointment by Name or ID");
            System.out.println("6. Search Doctor by Name or ID");
            System.out.println("7. Search Patient by Name or NIC");
            System.out.println("8. Cancel Appointment");
            System.out.println("9. Generate Invoice");
            System.out.println("10. Exit");
            System.out.print("Select an option: ");
            System.out.println("=" + "=".repeat(40));

            int option = scanner.nextInt();
            scanner.nextLine();

            switch (option) {
                case 1:
                    registerPatient(scanner); break;
                case 2:
                    makeAppointment(scanner); break;
                case 3:
                    updateAppointment(scanner); break;
                case 4:
                    viewAppointmentsByDate(scanner); break;
                case 5:
                    searchAppointment(scanner); break;
                case 6:
                    searchDoctor(scanner); break;
                case 7:
                    searchPatient(scanner); break;
                case 8:
                    cancelAppointment(scanner); break;
                case 9:
                    generateInvoice(scanner); break;
                case 10:
                    System.out.println("Exiting the system. Goodbye!");
                    return;
                default:
                    System.out.println("Invalid option, please try again.");
                    break;
            }
        }
    }
```

Figure 17 Code Snippet 01 - Main Program Class

3.1.2 Person Class

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

// Base Class: Person
// ! This class represents a generic person with basic contact information.
class Person {
    protected String name;
    protected String email;
    protected String contactNumber;

    // ! Constructor to initialize a Person object
    public Person(String name, String email, String contactNumber) {
        this.name = name;
        this.email = email;
        this.contactNumber = contactNumber;
    }

    // ! Method to get formatted contact information
    public String getContactInfo() {
        return "Name: " + name + ", Email: " + email + ", Contact: " + contactNumber;
    }
}
```

Figure 18 Code Snippet 02 - Person Class

The Person class in Java provides a structured way to represent individuals with basic contact details, including name, email, and contact number. When a Person object is created, its constructor is called with these details, which are stored in the object's fields. This information is encapsulated, meaning it can only be accessed through specific methods, ensuring data security and consistency.

The getContactInfo() method formats and returns a string combining the person's name, email, and contact number, making it easy to display or use the information elsewhere in the application. This class enables efficient management of contact information for multiple individuals within a program.

3.1.3 Patient Class

```java
// Patient Class extending Person
// ! Inheriting basic contact information from the Person class.
class Patient extends Person {
    private String nic; // * National Identity Card number of the patient

    public Patient(String name, String email, String contactNumber, String nic) {
        super(name, email, contactNumber); // * Call to the superclass constructor to set
        this.nic = nic;
    }

    public String getNIC() {
        return nic;
    }
}
```

Figure 19 Code Snippet 03 - Patient Class

The Patient class in Java extends the Person class, inheriting its basic contact information fields name, email, and contact number. It adds a specific field, nic, to store the patient's National Identity Card number. Through inheritance, Patient objects have access to the fields and methods of Person, including getContactInfo(), making it easy to reuse code.

The Patient constructor accepts the name, email, contact number, and NIC as arguments, using the Person constructor to initialize the inherited fields and setting nic for the specific patient. The getNIC() method provides access to the NIC value, making it easy to retrieve. This class showcases inheritance in object-oriented programming by building upon a general Person class to create a specialized Patient class with additional, relevant details.

3.1.3.1 Register Patient Method

```java
// ! Method to register a new patient.
public static void registerPatient(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("            Register New Patient            ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Patient Name: ");
    String name = scanner.nextLine().trim();
    System.out.print("Enter Email: ");
    String email = scanner.nextLine().trim();
    System.out.print("Enter Contact Number: ");
    String contactNumber = scanner.nextLine().trim();
    System.out.print("Enter NIC: ");
    String nic = scanner.nextLine().trim();

    // Validate input (simple validation example)
    if (name.isEmpty() || email.isEmpty() || contactNumber.isEmpty() || nic.isEmpty()) {
        System.out.println("All fields are required. Please try again.");
        return;
    }

    patients.add(new Patient(name, email, contactNumber, nic));
    System.out.println("Patient Registered Successfully.");
    System.out.println("=" + "=".repeat(40));
}
```

Figure 20 Code Snippet 04 - Register Patient Method

Creating the Patient Object:

- **New Patient Creation**: Once all input fields are successfully filled, a new Patient object is created using the provided details. The patient's name, email, contact number, and NIC are passed to the constructor of the Patient class to instantiate the object.

Adding to the Patients List:

- **Storing the Patient**: After the Patient object is created, it is added to the list of patients (patients list). This list stores all registered patients in the system, making it possible to manage and retrieve patient data later.

Error Handling:

- **Empty Fields Error**: If any of the required fields are left empty during input, the method displays an error message. This prevents incomplete data from being entered into the system and ensures that all required information is captured before the patient is registered.

3.1.3.2 Search Patient Method

```java
// ! Method to search for a patient by name or NIC.
public static void searchPatient(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("           Search for a Patient           ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Patient Name or NIC to search: ");
    String input = scanner.nextLine().trim(); // Get input and trim whitespace

    for (Patient patient : patients) {
        // * Check if the patient's name or NIC matches the input (case insensitive)
        if (patient.name.equalsIgnoreCase(input) || patient.getNIC().equalsIgnoreCase(input)) {
            System.out.println
            ("Patient Found: " + patient.getContactInfo() + ", NIC: " + patient.getNIC());
            return;
        }
    }
    System.out.println("Patient not found. Please check the name or NIC and try again.");

}
```

Figure 21 Code Snippet 05 - Search Patient Method

- **Input Handling**: The method ensures that the user provides a valid search term (name or NIC). It uses Scanner to capture this input and applies the trim() method to eliminate leading and trailing spaces, improving search accuracy.

- **Search Logic**: The method then loops through the list of patients stored in the system. For each patient, it compares the input string to both the patient's name and NIC in a case-insensitive manner using equalsIgnoreCase(). This makes the search more flexible and user-friendly, as it doesn't require the user to match the exact case of the letters in the name or NIC.

- **Output Handling**: If a match is found, the method prints the relevant contact information, including the patient's name, contact details, and NIC, to the console. This allows the user to quickly view the necessary details of the matched patient. If no patient is found that matches the search criteria, the method displays an error message.

- **Error Handling**: The method includes a simple error message to notify the user if no match is found, improving the user experience by providing feedback when the search is unsuccessful.

3.1.4 Doctor Class

```java
// Doctor Class extending Person
// ! Inheriting properties and methods from the Person class.
class Doctor extends Person {
    private List<String> schedule = new ArrayList<>(); // * List to hold the doctor's availability
    protected String employeeID; // Unique identifier for the doctor

    public Doctor(String name, String email, String contactNumber, String employeeID) {
        super(name, email, contactNumber); // * Call to the superclass constructor
        this.employeeID = employeeID; // Set the employee ID
        addAvailability(); // ! Method to populate the doctor's schedule
    }

    // ! Private method to add availability times to the doctor's schedule
    private void addAvailability() {

    }

    public List<String> getSchedule() {
        return schedule;
    }

    public String getEmployeeDetails() {
        return "Employee ID: " + employeeID + ", " + getContactInfo();
    }
}
```

Figure 22 Code Snippet 06 - Doctor Class

The Doctor class extends the Person class, inheriting its basic contact information properties. It adds two additional fields:

- schedule: A List to store the doctor's availability slots.
- employeeID: A unique identifier for the doctor.

The Doctor class also has a constructor that initializes the inherited fields from the Person class and sets the employeeID field. It also calls a private method addAvailability() to populate the schedule list with the doctor's availability times.

The getSchedule() method returns the doctor's availability schedule, and the getEmployeeDetails() method returns a string containing the doctor's employee ID and contact information. The Doctor class builds upon the Person class to create a specialized representation of a doctor, including their availability and unique identifier.

3.1.4.1 Search Doctor Method

```java
// ! Method to search for a doctor by name or ID.
public static void searchDoctor(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("           Search for a Doctor           ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Doctor Name or ID (D***): ");
    String input = scanner.nextLine().trim(); // Get input and trim whitespace

    boolean found = false; // Flag to check if the doctor is found
    for (Doctor doctor : doctors) {
        // Check if the doctor's name or employee ID matches the input (case insensitive)
        if (doctor.name.equalsIgnoreCase(input) || doctor.employeeID.equalsIgnoreCase(input)) {
            System.out.println("Doctor Found: " + doctor.getEmployeeDetails());
            found = true; // Set the flag to true if the doctor is found
            break;
        }
    }

    if (!found) {
        System.out.println("Doctor not found. Please check the name or ID and try again.");
    }

    System.out.println("=" + "=".repeat(40));
}
```

Figure 23 Code Snippet 07 - Search Doctor Method

- **Input Handling:** The method accepts a Scanner object as input, which is used to capture the search criteria (name or employee ID) from the user via the console. The method prompts the user to enter either the name or the employee ID of the doctor they wish to search for.
    - The input is trimmed to remove any leading or trailing whitespace to ensure the search is clean and accurate.

- **Search Logic:** After receiving the input, the method iterates through the list of doctors stored in the system. For each doctor, it checks whether the input (name or employee ID) matches the corresponding fields of any doctor.
    - The comparison is case-insensitive, ensuring that capitalization differences in the input do not affect the search results.

3.1.5 Appointment Class

```java
// Appointment Class
// ! This class represents an appointment made by a patient with a doctor.
class Appointment {
    private static int counter = 1; // ? Static counter for generating unique appointment IDs
    private int appointmentID;
    private String date;
    private String time;
    private Status status;
    private Patient patient;
    private Doctor doctor;
    private Treatment treatment;
    private double registrationFee;

    // * Constructor to initialize an Appointment object with date, time, patient,and doctor
    public Appointment(String date, String time, Patient patient, Doctor doctor, Treatment treatment) {
        this.appointmentID = counter++; // * Assign a unique ID and increment the counter
        this.date = date;
        this.time = time;
        this.status = Status.BOOKED; // ! Set the initial status to BOOKED
        this.patient = patient;
        this.doctor = doctor;
        this.treatment = treatment;
        this.registrationFee = 500.0; // ! Default registration fee
    }

    // * Getters for appointment details
    public int getAppointmentID() {
        return appointmentID;
    }

    public String getDate() {
        return date;
    }

    public String getTime() {
        return time;
    }

    public Patient getPatient() {
        return patient;
    }

    public Doctor getDoctor() {
        return doctor;
    }

```

```java
    // ! Method to update the date and time of the appointment
    public void updateDateTime(String date, String time) {
        this.date = date;
        this.time = time;
        System.out.println("Appointment updated to Date: " + date + ", Time: " + time);
    }

    // ! Method to get a formatted string of appointment details
    public String getDetails() {
        return "Appointment ID: " + appointmentID + ", Date: " + date + ", Time: " + time + ", Status: " + status +
                ", Patient: " + patient.name + ", Doctor: " + doctor.name + ", Treatment: " + treatment.getDetails();
    }

    public void confirm() {
        status = Status.BOOKED;
        System.out.println("Appointment confirmed for: " + patient.name + " with Dr. " + doctor.name);
    }

    public void cancel() {
        status = Status.CANCELED;
        System.out.println("Appointment canceled for: " + patient.name);
    }

    public Treatment getTreatment() {
        return treatment;
    }

    public static Treatment selectTreatment(Scanner scanner, List<Treatment> availableTreatments) {
        System.out.println("Available Treatments:");
        for (int i = 0; i < availableTreatments.size(); i++) {
            System.out.println((i + 1) + ". " + availableTreatments.get(i).getDetails());
        }
        System.out.print("Select a treatment by number: ");
        int choice = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        return availableTreatments.get(choice - 1);
    }

    // * Getter for the registration fee
    public double getRegistrationFee() {
        return registrationFee;
    }
}
```

Figure 24 Code Snippet 08 - Appointment Class

The Appointment class models an appointment between a patient and a doctor within a healthcare system, including essential details like a unique appointmentID, date, time, status, associated patient and doctor, planned treatment, and registration fee. It provides a constructor to initialize these fields and getter methods to access each attribute. Key methods include updateDateTime() to modify the appointment's date and time, getDetails() to retrieve a formatted summary of appointment information, confirm() to mark the appointment as "BOOKED," and cancel() to mark it as "CANCELED." selectTreatment() allows users to choose from a list of treatments, while getTreatment() and getRegistrationFee() return the assigned treatment and fee. This class enables comprehensive management and tracking of patient appointments, providing functionality to confirm, cancel, and update appointments as needed.

3.1.5.1 Make New Appointment Method

```java
// ! Method to make a new appointment.
public static void makeAppointment(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("            Make a New Appointment            ");
    System.out.println("=" + "=".repeat(38) + "=");

    // Get patient NIC and verify if the patient exists
    System.out.print("Enter Patient NIC: ");
    String nic = scanner.nextLine().trim();
    Patient patient = findPatientByNic(nic);

    if (patient == null) {
        System.out.println("Patient not found. Please register the patient first.");
        return;
    }

    // Get appointment date and time
    System.out.print("Enter Date (| Mon | Wed | Fri | Sat |): ");
    String date = scanner.nextLine().trim();
    System.out.print("Enter Time (e.g., 10:00am): ");
    String time = scanner.nextLine().trim();

    // Display and select doctor
    System.out.println("Select Doctor:");
    for (int i = 0; i < doctors.size(); i++) {
        System.out.printf("%d: %s%n", i + 1, doctors.get(i).name);
    }
    System.out.print("Select Doctor (1-" + doctors.size() + "): ");
    int docChoice = scanner.nextInt();
    scanner.nextLine(); // Consume newline

    if (docChoice < 1 || docChoice > doctors.size()) {
        System.out.println("Invalid doctor selection. Please try again.");
        return;
    }
    Doctor doctor = doctors.get(docChoice - 1);

    // Assuming availableTreatments is a List<Treatment>
    System.out.println("Select Treatment Type:");
    for (int i = 0; i < availableTreatments.size(); i++) {
        System.out.printf("%d: %s%n", i + 1, availableTreatments.get(i).getDetails());
    }
    System.out.print("Select Treatment (1-" + availableTreatments.size() + "): ");
    int treatmentChoice = scanner.nextInt();
    scanner.nextLine(); // Consume newline

    if (treatmentChoice < 1 || treatmentChoice > availableTreatments.size()) {
        System.out.println("Invalid treatment selection. Please try again.");
        return;
    }

    // Get the actual Treatment object instead of a string
    Treatment selectedTreatment = availableTreatments.get(treatmentChoice - 1);

    // Create and add the appointment
    Appointment appointment = new Appointment(date, time, patient, doctor, selectedTreatment);
    appointments.add(appointment);

    System.out.println("Appointment booked successfully.");
    System.out.println("Appointment Details: " + appointment.getDetails());
    System.out.println("=" + "=".repeat(40));
}
```

Figure 25 Code Snippet 09 - Make New Appointment Method

Input Handling:

- Prompts the user to enter required information (e.g., patient name, doctor ID, appointment details).
- Validates the input to ensure it meets specific criteria (e.g., non-empty strings, valid date formats).
- Trims any leading or trailing whitespace from the input.

Object Creation:

- Creates instances of classes (e.g., Patient, Doctor, Appointment) to represent real-world entities.
- Initializes the objects with appropriate values based on user input or other data sources.

Search Logic:

- Iterates through a list of objects (patients, doctors, appointments) to find a match.
- Compares the user's input with the relevant fields of each object (e.g., name, ID, date).
- Uses case-insensitive comparisons to improve search accuracy.

List Addition:

- Adds newly created objects to relevant lists (e.g., patients, doctors, appointments).
- Manages the list of objects to ensure efficient searching and retrieval.

Error Handling:

- Checks for invalid input (e.g., empty strings, invalid dates).
- Handles exceptions that may occur during input/output operations or data processing.
- Provides informative error messages to guide the user.

3.1.5.2 Update Existing Appointment Method

```java
// ! Method to update the details of an existing appointment.
public static void updateAppointment(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("            Update Appointment Details            ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Appointment ID to update: ");
    int id = scanner.nextInt();
    scanner.nextLine(); // * Consume the newline character

    Appointment appointment = findAppointmentByID(id);
    if (appointment == null) {
        System.out.println("Appointment not found. Please check the ID and try again.");
        return;
    }

    System.out.print("Enter New Date (| Mon | Wed | Fri | Sat |): ");
    String newDate = scanner.nextLine().trim(); // * Get new appointment date
    System.out.print("Enter New Time (e.g., 10:00am): ");
    String newTime = scanner.nextLine().trim(); // * Get new appointment time

    // Update the appointment's date and time
    appointment.updateDateTime(newDate, newTime); // * Update method call
    System.out.println("Appointment updated successfully.");
    System.out.println("Updated Appointment Details: " + appointment.getDetails());
    System.out.println("=" + "=".repeat(40));
}
```

Figure 26 Code Snippet 10 - Update Existing Appointment Method

Input Handling:

- Prompts the user to enter the appointment ID to be updated.
- Validates the entered ID by searching for the corresponding appointment in the list.
- If the appointment is not found, an error message is displayed.
- Prompts the user to enter the new date and time for the appointment.

Search Logic:

- Uses the findAppointmentById method to locate the appointment with the specified ID.
- Iterates through the list of appointments to find a match based on the ID.

Object Modification:

- Calls the updateDateTime method of the Appointment object to modify its date and time fields.
- Updates the appointment's state within the list of appointments.

Error Handling:

- Checks if the appointment with the given ID exists.
- Provides an error message if the appointment is not found.

3.1.5.3 View Appointment Method

```java
// ! Method to view appointments by a specific date.
public static void viewAppointmentsByDate(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("         View Appointments by Date         ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Date (| Mon | Wed | Fri | Sat |) to filter appointments: ");
    String date = scanner.nextLine().trim(); // Get the date from the user to filter appointments

    boolean found = false; // Flag to check if any appointments are found
    for (Appointment appTime : appointments) {
        if (appTime.getDate().equals(date)) { // Check if the appointment's date matches the input date
            System.out.println(appTime.getDetails());
            found = true; // Set the flag to true if at least one appointment is found
        }
    }

    if (!found) {
        System.out.println("No appointments found for the date: " + date);
    }

    System.out.println("=" + "=".repeat(40));
}
```

Figure 27 Code Snippet 11 - Update Existing Appointment Method

Input Handling:

- Prompts the user to enter a specific date (e.g., "Mon", "Wed", "Fri", "Sat") to filter appointments.
- Validates the input to ensure it matches one of the allowed date formats.

Search Logic:

- Iterates through a list of appointments and checks if each appointment's date matches the input date.
- Uses the equals method to compare the date strings.

Output:

- If appointments are found for the specified date, the method prints the details of each appointment, including information like date, time, patient, doctor, and treatment.
- If no appointments are found, an appropriate message is displayed.

Error Handling:

- Checks if any appointments were found during the search.
- If no appointments are found, an error message is displayed to inform the user.

3.1.5.4 Search Appointment by Patient Name / Appointment ID

```java
// ! Method to search for an appointment by ID or patient name.
public static void searchAppointment(Scanner scanner) {
    System.out.print("Enter Appointment ID or Patient Name to search: ");
    String input = scanner.nextLine();

    try {
        int id = Integer.parseInt(input);
        Appointment appTime = findAppointmentByID(id);
        if (appTime != null) {
            System.out.println(appTime.getDetails());
        } else {
            System.out.println("No appointment found with ID: " + id);
        }
    } catch (NumberFormatException e) {
        // * Handle case where input is not an integer (search by patient name)
        for (Appointment appTime : appointments) {
            // * Check if the patient's name matches the input
            if (appTime.getPatient().name.equalsIgnoreCase(input)) {
                System.out.println(appTime.getDetails());
            }
        }
    }
}
```

Figure 28 Code Snippet 12 - Search Appointment by Patient Name / Appointment ID Method

Input Handling:

- Prompts the user to enter an appointment ID or patient name to search for.
- Stores the user's input in a string variable.

Search Logic:

- **Search by ID:**
  - Attempts to parse the input as an integer.
  - If successful, calls the findAppointmentById method to locate the appointment with the given ID.
  - If the appointment is found, its details are printed.
  - If the appointment is not found, an error message is displayed.

Error Handling:

- Uses a try-catch block to handle potential NumberFormatException errors if the input cannot be parsed as an integer.
- Provides informative error messages for both ID-based and name-based searches.

3.1.5.5 Cancel Appointment Method

```java
// ! Method to cancel an existing appointment.
public static void cancelAppointment(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40)); // Decorative line
    System.out.println("            Cancel Appointment            ");
    System.out.println("=" + "=".repeat(38) + "="); // Decorative line

    System.out.print("Enter Appointment ID to cancel: ");
    int id = scanner.nextInt(); // * Get the Appointment ID from user input
    Appointment appointment = findAppointmentByID(id); // * Find the appointment by ID

    if (appointment != null) {
        appointment.cancel(); // * Call the cancel method on the appointment if found
    } else {
        System.out.println("Appointment not found.");
    }
    System.out.println("=" + "=".repeat(40)); // Decorative line
}
```

Figure 29 Code Snippet 13 - Cancel Appointment Method

Input Handling:

- Prompts the user to enter the appointment ID to be canceled.
- Reads the input and stores it in an integer variable.

Search Logic:

- Calls the findAppointmentById method to locate the appointment with the specified ID.
- Iterates through the list of appointments to find a match based on the ID.

Object Modification:

- If the appointment is found, the cancel() method is called on the appointment object to set its status to "CANCELED".

Error Handling:

- Checks if the appointment with the given ID exists.
- If the appointment is not found, an error message is displayed.

Output:

- Prints a confirmation message if the cancellation is successful.
- Prints an error message if the appointment is not found.

3.1.6 Treatment Class

```java
// Treatment Class
// ! This class represents a treatment that can be provided to a patient.
class Treatment {
    private int treatmentID;
    private String name;
    private double price;

    public Treatment(int treatmentID, String name, double price) {
        this.treatmentID = treatmentID;
        this.name = name;
        this.price = price;
    }

    // * Method to calculate the final price of the treatment (no discount applied in this example)
    public double calculateFinalPrice() {
        return price;
    }

    // * Method to get a formatted string of treatment details
    public String getDetails() {
        return "Name: " + name + ", Price: LKR " + price;
    }
}
```

Figure 30 Code Snippet 14 - Treatment Class

The Treatment class represents medical treatments within a healthcare application, capturing key information about each treatment. It includes a constructor to initialize a treatment with a treatmentID, name, and price. The calculateFinalPrice() method provides the treatment's cost, while getDetails() returns a formatted string containing the treatment's name and price.

This class allows for the management of different treatments, including options like Acne Treatment (2750.00), Skin Whitening (7650.00), Mole Removal (3850.00), and Laser Treatment (12,500.00), offering a foundational structure for handling treatment details and pricing within the system.

3.1.7 Payment Class

```java
// Payment Class
// ! This class represents a payment transaction for a specific amount.
class Payment {
    private static final double TAX_RATE = 0.025; // * Constant representing the tax rate (2.5%)
    private double amount; // * The amount to be processed for payment

    public Payment(double amount) {
        this.amount = amount;
    }

    // * Method to calculate the total amount including tax
    public double calculateTotalAmount() {
        return Math.round(amount * (1 + TAX_RATE) * 100.0) / 100.0;
    }
}
```

Figure 31 Code Snippet 15 - Payment Class

The Payment class models a payment transaction within a system, focusing on processing an amount with tax calculations. It includes a constant TAX_RATE of 2.5% and an amount field for the payment amount. The constructor initializes the payment with a specified amount, while the calculateTotalAmount() method calculates the total amount including tax by applying the tax rate and rounding the result to two decimal places. This class offers a straightforward structure for handling payments and accurately determining the total transaction amount with tax included.

- **Tax Calculation**: The class ensures that any payment transaction accurately includes the tax amount, offering a straightforward way to compute the total transaction value.

- **Clear Constructor Initialization**: The constructor initializes the payment with a given amount, making it easy to instantiate and manage payment records.

- **Efficient Total Calculation**: The method calculateTotalAmount() calculates the full amount due, ensuring the tax is applied and the total is rounded to two decimal places, which is crucial for financial transactions.

3.1.8 Invoice Class

```java
// Invoice Class
// ! This class represents an invoice generated for a specific appointment and treatment.
class Invoice {
    private int invoiceID;
    private Appointment appointment;
    private Treatment treatment;
    private Payment payment;

    public Invoice(int invoiceID, Appointment appointment, Treatment treatment) {
        this.invoiceID = invoiceID;
        this.appointment = appointment;
        this.treatment = treatment;
        this.payment = new Payment(treatment.calculateFinalPrice());
        // Create a payment object based on treatment price
    }

    public void generateInvoice() {
        System.out.println("\n" + "=".repeat(30));
        System.out.println("               INVOICE               ");
        System.out.println("=" + "=".repeat(28) + "=");

        // Display appointment details
        System.out.printf("Appointment ID:    %d%n", appointment.getAppointmentID());
        System.out.printf("Patient:           %s%n", appointment.getPatient().name);
        System.out.printf("Date:              %s%n", appointment.getDate());
        System.out.printf("Time:              %s%n", appointment.getTime());
        System.out.println("-".repeat(30));

        // Display treatment details
        System.out.printf("%s%n", treatment.getDetails());
        System.out.printf("Registration Fee:  LKR %.2f%n", appointment.getRegistrationFee());

        // Calculate and display tax
        double tax = treatment.calculateFinalPrice() * 0.025;
        System.out.printf("Tax (2.5%%):       LKR %.2f%n", tax);

        // Calculate and display total amount
        double totalAmount = payment.calculateTotalAmount();
        System.out.printf("Total:            LKR %.2f%n", totalAmount);

        System.out.println("=" + "=".repeat(28) + "=");
        System.out.println("Thank you for choosing our services!");
        System.out.println("=" + "=".repeat(30));
    }
}
```

Figure 32 Code Snippet 16 - Invoice Class

The Invoice class models an invoice for a specific appointment and treatment, enabling organized billing within a healthcare system. It contains fields for a unique invoiceID, the associated Appointment and Treatment objects, and a Payment object based on the treatment's price. The constructor initializes the invoice with these details. Through the generateInvoice() method, it produces a formatted invoice that includes the appointment ID, patient name, date, time, treatment name, treatment price, registration fee, tax (2.5% of the treatment price), and total amount (treatment price plus tax), ending with a thank-you message. This class provides a comprehensive structure for generating clear, itemized invoices for appointments, facilitating accurate billing and payment processing.

3.1.8.1 Generate Invoice Method

```java
// ! Method to generate an invoice for a specific appointment.
public static void generateInvoice(Scanner scanner) {
    System.out.println("\n" + "=".repeat(40));
    System.out.println("            Generate Invoice            ");
    System.out.println("=" + "=".repeat(38) + "=");

    System.out.print("Enter Appointment ID: ");
    int id = scanner.nextInt();
    Appointment appointment = findAppointmentByID(id);

    if (appointment == null) {
        System.out.println("Appointment not found.");
        return;
    }

    System.out.print(
            "Select Treatment Type (1: Acne Treatment, 2: Skin Whitening, 3: Mole Removal, 4: Laser Treatment): ");
    int treatmentChoice = scanner.nextInt();
    Treatment treatment;

    // ! Determine the treatment based on user selection
    switch (treatmentChoice) {
        case 1:
            treatment = new Treatment(1, "Acne Treatment", 2750.00);
            break;
        case 2:
            treatment = new Treatment(2, "Skin Whitening", 7650.00);
            break;
        case 3:
            treatment = new Treatment(3, "Mole Removal", 3850.00);
            break;
        case 4:
            treatment = new Treatment(4, "Laser Treatment", 12500.00);
            break;
        default:
            System.out.println("Invalid treatment option.");
            return;
    }

    // * Create a new Invoice object using the generated invoice ID.
    Invoice invoice = new Invoice(invoiceCounter++, appointment, treatment);
    invoice.generateInvoice();

    System.out.println("=" + "=".repeat(40));
}
```

Figure 33 Code Snippet 17 - Generate Invoice Method

Input Handling:

- Prompts the user to enter the appointment ID.
- Validates the appointment ID by searching for the corresponding appointment in the list.

Treatment Selection:

- Displays a list of available treatment options.
- Prompts the user to select a treatment type.
- Creates a Treatment object based on the user's choice.

Invoice Generation:

- Creates a new Invoice object with the generated invoice ID, the selected appointment, and the chosen treatment.
- Calls the generateInvoice() method of the Invoice object to print the invoice details, including:
    - Appointment details (ID, date, time, patient, doctor)
    - Treatment details (name, price)
    - Tax amount (calculated based on the treatment price)
    - Total amount (treatment price + tax)

Error Handling:

- Checks if the appointment with the given ID exists.
- If the appointment is not found, an error message is displayed.
- Validates the user's treatment choice and displays an error message for invalid choices.

3.2 Effective use of Object-Oriented Programming (OOP) concepts

3.2.1 Encapsulation

The **Person** class encapsulates the basic contact information of a person (name, email, and contact number) and provides a method **getContactInfo()** to access this information without exposing the data directly.

```java
// ! This class represents a generic person with basic contact information.
class Person {
    protected String name;
    protected String email;
    protected String contactNumber;

    // ! Constructor to initialize a Person object
    public Person(String name, String email, String contactNumber) {
        this.name = name;
        this.email = email;
        this.contactNumber = contactNumber;
    }

    // ! This method access the information without exposing the data directly
    public String getContactInfo() {
        return "Name: " + name + ", Email: " + email + ", Contact: " + contactNumber;
    }
}
```

Figure 34 Code Snippet 18 – Encapsulation of OOP Concept

3.2.2 Inheritance

The **Patient** and **Doctor** classes extend the **Person** class, inheriting its attributes and methods. This allows both **Patient** and **Doctor** to have the basic contact information without needing to redefine it.

```java
class Patient extends Person {
    private String nic; // National Identity Card number of the patient

    public Patient(String name, String email, String contactNumber, String nic) {
        super(name, email, contactNumber); // Call to the superclass constructor
        this.nic = nic;
    }

    public String getNIC() {
        return nic;
    }
}

class Doctor extends Person {
    private List<String> schedule = new ArrayList<>();
    protected String employeeID; // Unique identifier for the doctor

    public Doctor(String name, String email, String contactNumber, String employeeID) {
        super(name, email, contactNumber); // Call to the superclass constructor
        this.employeeID = employeeID;
    }
}
```

Figure 35 Code Snippet 18 – Inheritance of OOP Concept

3.2.3 Polymorphism

The **getDetails()** method in the **Appointment** class provides a way to get formatted details of the appointment. Although the method name is the same, it operates on different types of objects (i.e., **Appointment**, **Patient**, **Doctor**).

```
// ! Same method name operates on different types of objects
    public String getDetails() {
        return "Appointment ID: " + appointmentID + ", Date: " + date + ", Time: " + time + ", Status: " + status +
                ", Patient: " + patient.name + ", Doctor: " + doctor.name + ", Treatment: " + treatment.getDetails();
    }
```

Figure 36 Code Snippet 18 – Polymorphism of OOP Concept

3.2.4 Abstraction

The **Invoice** class abstracts the details of generating an invoice, allowing users to generate an invoice without needing to understand the underlying calculations for taxes and totals.

```
// ! This Method allowing users to generate an invoice without needing to understand calculation
s for taxes and totals.
    public void generateInvoice() {
        System.out.println("\n" + "=".repeat(30));
        System.out.println("               INVOICE              ");
        System.out.println("=" + "=".repeat(28) + "=");

        // Display appointment details
        System.out.printf("Appointment ID:    %d%n", appointment.getAppointmentID());
        System.out.printf("Patient:           %s%n", appointment.getPatient().name);
        System.out.printf("Date:              %s%n", appointment.getDate());
        System.out.printf("Time:              %s%n", appointment.getTime());
        System.out.println("-".repeat(30));

        // Display treatment details
        System.out.printf("%s%n", treatment.getDetails());
        System.out.printf("Registration Fee:  LKR %.2f%n", appointment.getRegistrationFee());

        // Calculate and display tax
        double tax = treatment.calculateFinalPrice() * 0.025;
        System.out.printf("Tax (2.5%%):       LKR %.2f%n", tax);

        // Calculate and display total amount
        double totalAmount = payment.calculateTotalAmount();
        System.out.printf("Total:            LKR %.2f%n", totalAmount);
```

Figure 37 Code Snippet 18 – Abstraction of OOP Concept

3.3 Effective use of Data Structures

3.3.1 Classes as Data Structures

The **Appointment** class organizes related data (appointment details) and methods.

```java
// ! This class represents an appointment made by a patient with a doctor.
class Appointment {
    private static int counter = 1;
    private int appointmentID;
    private String date;
    private String time;
    private Status status;
    private Patient patient;
    private Doctor doctor;
    private Treatment treatment;
    private double registrationFee;

    // * Constructor to initialize an Appointment object with date, time, patient,and doctor
    public Appointment(String date, String time, Patient patient, Doctor doctor, Treatment treatment) {
        this.appointmentID = counter++; // * Assign a unique ID and increment the counter
        this.date = date;
        this.time = time;
        this.status = Status.BOOKED; // ! Set the initial status to BOOKED
        this.patient = patient;
        this.doctor = doctor;
        this.treatment = treatment;
        this.registrationFee = 500.0; // ! Default registration fee
    }

    // * Getters for appointment details
    public int getAppointmentID() {
        return appointmentID;
    }

    // * Getter for registration fee
    public double getRegistrationFee() {
        return registrationFee;
    }
```

Figure 38 Code Snippet 19 – Classes as Data Structures

3.3.2 Collections Framework

The **AuroraSkinCare** class uses an **ArrayList** to manage multiple objects, showcasing the use of a collection.

```java
// ! This class serves as the entry point for the Aurora Skin Care Clinic.
public class AuroraSkinCareSystem {
    static List<Patient> patients = new ArrayList<>();
    static List<Doctor> doctors = new ArrayList<>();
    static List<Appointment> appointments = new ArrayList<>();
    static List<Treatment> availableTreatments = new ArrayList<>();
```

Figure 39 Code Snippet 20 – Collections Framework of Data Structure

3.3.3 Encapsulation of Data

The **Treatment** class encapsulates treatment details and provides methods for accessing and manipulating that data.

```java
// ! This class represents a treatment that can be provided to a patient.
class Treatment {
    private int treatmentID;
    private String name;
    private double price;

    public Treatment(int treatmentID, String name, double price) {
        this.treatmentID = treatmentID;
        this.name = name;
        this.price = price;
    }

    // * Method to get a formatted string of treatment details
    public String getDetails() {
        return "Name: " + name + ", Price: LKR " + price;
    }

    // * Method to calculate the final price of the treatment
    public double calculateFinalPrice() {
        return price;
    }
}
```

Figure 40 Code Snippet 21 – Encapsulation of Data of Data Structure

3.3.4 Use of Primitive Types

The **Payment** class uses **double** for fees and tax rates, demonstrating the use of primitive types for basic data handling.

```java
// ! This class represents a payment transaction for a specific amount.
class Payment {
    private static final double TAX_RATE = 0.025;
    private double amount; // * The amount to be processed for payment

    public Payment(double amount) {
        this.amount = amount;
    }

    // * Method to calculate the total amount including tax
    public double calculateTotalAmount() {
        return Math.round(amount * (1 + TAX_RATE) * 100.0) / 100.0;
    }
}
```

Figure 41 Code Snippet 22 - Uses of Primitive Types of Data Structure

3.4 Effective use of Algorithms

3.4.1 Searching Algorithms: Searching for a Patient by NIC or Name

This method uses Java Streams, which provide a high-level abstraction for processing sequences of elements. The **filter** operation applies a predicate to each element in the **patients** list, efficiently narrowing down the results to find the first patient that matches the provided NIC.

```
// ! Method to find a patient by NIC.
    public static Patient findPatientByNic(String nic) {
        // * Use a stream to filter the list of patients and find the first patient with the matching NIC
        return patients.stream().filter(p -> p.getNIC().equals(nic)).findFirst().orElse(null);
    }
```

Figure 42 Code Snippet 23 - Searching Algorithms

3.4.2 Data Manipulation Algorithms: Adding a New Appointment

When adding a new appointment, the **add()** method of **ArrayList** is utilized. This method is efficient as it typically operates in constant time, O(1), unless the internal array needs to be resized (which happens infrequently).

The algorithm behind **ArrayList** manages dynamic resizing and ensures that the addition of new elements remains efficient.

```
// Create and add the appointment
Appointment appointment = new Appointment(date, time, patient, doctor, selectedTreatment);
appointments.add(appointment);
```

Figure 43 Code Snippet 24 - Data Manipulation Algorithms

3.4.3 Invoice Generation Algorithms: Calculating Total Amounts

The **calculateTotalAmount()** method demonstrates a simple mathematical algorithm for calculating the total amount including tax. The use of **Math.round()** ensures that the result is rounded to two decimal places, which is crucial for financial calculations.

```
    // * Method to calculate the total amount including tax
    public double calculateTotalAmount() {
        return Math.round(amount * (1 + TAX_RATE) * 100.0) / 100.0;
    }
}
```

Figure 44 Code Snippet 25 - Invoice Generation Algorithms

Activity 04 – Testing & Conclusion

4.1 Black-Box Testing

The Aurora Skin Care System underwent rigorous testing to ensure its functionality and usability. This testing involved a combination of black-box and validation techniques.

Black-box testing focused on evaluating the system's behavior from an end-user perspective. By interacting with the command-line interface (CLI) and observing the system's responses to various inputs, the author verified that each function operated as expected, aligning with the system's specifications. This approach effectively simulated real-world user interactions.

Validation testing concentrated on confirming that the system met its requirements and performed its intended functions efficiently. The author meticulously examined the system's responses to user commands, ensuring that patient registration, appointment scheduling, and invoice generation were accurate and reliable. This testing method helped identify any discrepancies between the system's behavior and user expectations, ensuring that the system functioned effectively in various scenarios.

By combining black-box and validation testing, the author established a comprehensive testing strategy that addressed both technical and user-centric aspects of the system. This approach enhanced the system's quality, reliability, and user-friendliness. The synergy of these two testing methods played a crucial role in identifying and rectifying functional issues and usability gaps, ultimately resulting in a robust and efficient Aurora Skin Care System that met the needs of both the clinic and its users.

4.2 Unit Testing: Test Cases for Aurora Skin Care

4.2.1 Patient Related Test Cases

Test Case 01 – Register New Patient	
Test ID	TC - 001
Test Name	Register New Patient
Steps to Test	1. Start the Aurora Skin Care system. 2. Select the option "Register Patient". 3. Enter all required patient details (Name, NIC, Contact). 4. Confirm registration.
Pre-defined Function	None
Post-defined Function	Search Patient by NIC
Date of Testing	2024-11-08
Expected Outcome	New patient is successfully registered, and confirmation message is displayed.
Actual Outcome	========== Register New Patient ========== Enter Patient Name: A.A.M Aathif Enter Email: Mhdaathi124@gmail.com Enter Contact Number: 0769183535 Enter NIC: 200119803346 Patient Registered Successfully. ==========
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Check if all mandatory fields are validated.

Table 1 Test Case 01 Patient - Register New Patient

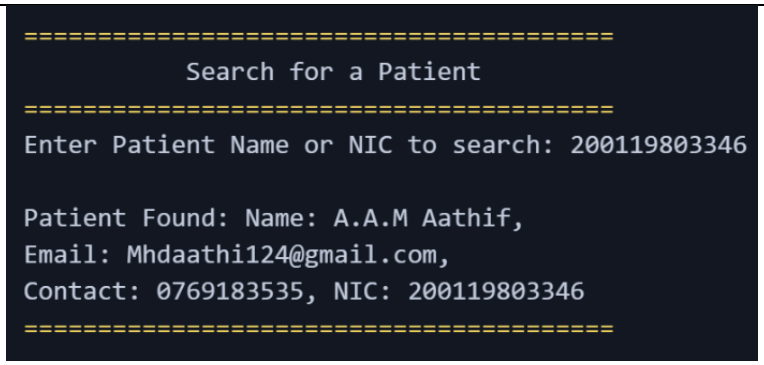
Test Case 02 – Search Patient by NIC	
Test ID	TC - 002
Test Name	Search Patient by NIC
Steps to Test	1. Start the Aurora Skin Care system. 2. Select the option "Search Patient by Name or NIC". 3. Enter a valid NIC. 4. Verify if the correct patient details are displayed.
Pre-defined Function	Register New Patient
Post-defined Function	Search Patient by Name
Date of Testing	2024-11-08
Expected Outcome	System retrieves and displays the correct patient details based on NIC.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with both existing and non-existing NIC values.

Table 2 Test Case 02 Patient - Search Patient by NIC

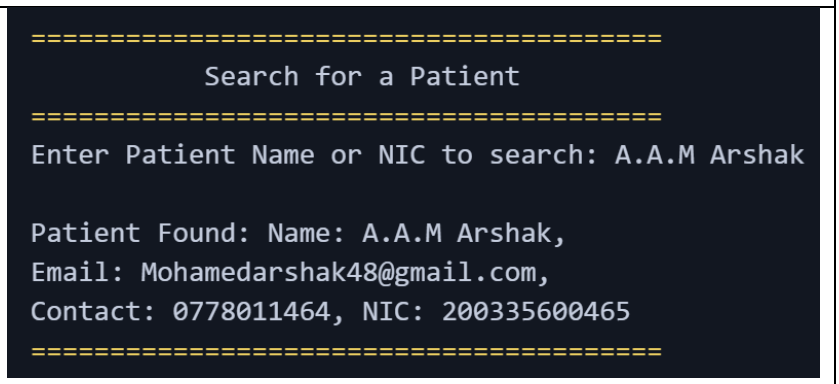
Test Case 03 – Search Patient by Name	
Test ID	TC - 003
Test Name	Search Patient by Name
Steps to Test	1. Start the Aurora Skin Care system. 2. Select the option "Search Patient by Name or NIC". 3. Enter the Patient Name Correctly. 4. Verify if the correct patient details are displayed.
Pre-defined Function	Search Patient by NIC
Post-defined Function	Invalid NIC for Search
Date of Testing	2024-11-08
Expected Outcome	System retrieves and displays the correct patient details based on Name.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with both partial and full names.

Table 3 Test Case 03 Patient - Search Patient by Name

Test Case 04 – Invalid NIC for Search	
Test ID	TC - 004
Test Name	Invalid NIC for Search
Steps to Test	1. Start the Aurora Skin Care system. 2. Select the option "Search Patient by Name or NIC". 3. Enter an invalid NIC (e.g., incomplete or incorrectly formatted NIC).
Pre-defined Function	Search Patient by NIC
Post-defined Function	Search Doctor by ID
Date of Testing	2024-11-08
Expected Outcome	System retrieves and displays the correct patient details based on NIC.
Actual Outcome	``` == Search for a Patient == Enter Patient Name or NIC to search: 200119803346V Patient not found. Please check the name or NIC and Try Again. == ```
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify system handling for different invalid NIC formats.

Table 4 Test Case 04 Patient - Invalid NIC for Search

4.2.2 Doctor-Related Test Cases

Test Case 05 – Search Doctor by ID	
Test ID	TC - 005
Test Name	Search Doctor by ID
Steps to Test	1. Start the system. 2. Select "Search Doctor by Name or ID". 3. Enter a valid doctor ID.
Pre-defined Function	Invalid NIC for Search
Post-defined Function	Search Doctor by Name
Date of Testing	2024-11-08
Expected Outcome	Doctor details are displayed for the entered ID.
Actual Outcome	========== Search for a Doctor ========== Enter Doctor Name or ID (D***): D001 Doctor Found: Employee ID: D001, Name: Dr. Ijlan, Email: mohamedijlan02@gmail.com, Contact: 0776778795 ==========
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with both valid and invalid doctor IDs.

Table 5 Test Case 05 Doctor - Search Doctor by ID

Test Case 06 – Search Doctor by Name	
Test ID	TC - 006
Test Name	Search Doctor by Name
Steps to Test	1. Start the system. 2. Select "Search Doctor by Name or ID". 3. Enter a valid doctor name.
Pre-defined Function	Search Doctor by ID
Post-defined Function	Invalid Doctor ID
Date of Testing	2024-11-08
Expected Outcome	Doctor details are displayed for the entered name.
Actual Outcome	``` ========== Search for a Doctor ========== Enter Doctor Name or ID (D***): Dr. Brian Doctor Found: Employee ID: D002, Name: Dr. Brian, Email: jacobmichaelbrian01@gmail.com, Contact: 0764517561 ========== ```
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with both partial and full names.

Table 6 Test Case 06 Doctor - Search Doctor by Name

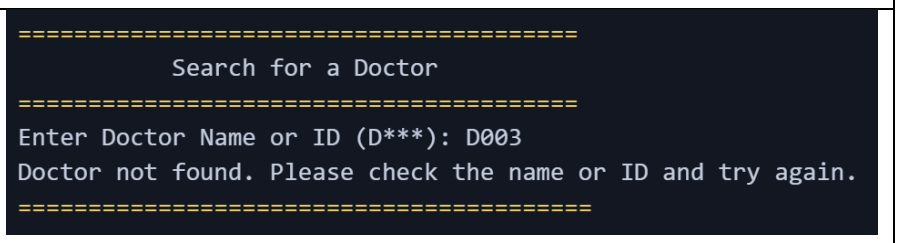
Test Case 07 – Invalid Doctor ID	
Test ID	TC - 007
Test Name	Invalid Doctor ID
Steps to Test	1. Start the system. 2. Select "Search Doctor by Name or ID". 3. Enter an invalid doctor ID.
Pre-defined Function	Search Doctor by Name
Post-defined Function	Invalid Doctor ID
Date of Testing	2024-11-08
Expected Outcome	System displays "Doctor not found" message.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with a variety of invalid formats.

Table 7 Test Case 07 Doctor - Invalid Doctor ID

Test Case 08 – Invalid Doctor Name	
Test ID	TC - 008
Test Name	Invalid Doctor Name
Steps to Test	1. Start the system. 2. Select "Search Doctor by Name or ID". 3. Enter an invalid doctor Name.
Pre-defined Function	Invalid Doctor ID
Post-defined Function	Make Appointment
Date of Testing	2024-11-08
Expected Outcome	System displays "Doctor not found" message.

Actual Outcome	`========== Search for a Doctor ==========`
	`Enter Doctor Name or ID (D***): Dr. Aathif`
	`Doctor not found. Please check the name or ID and try again.`
	`==========`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with a variety of invalid formats.

Table 8 Test Case 08 Doctor - Invalid Doctor Name

4.2.3 Appointment-Related Test Cases

Test Case 09 – Make Appointment	
Test ID	TC - 009
Test Name	Make Appointment
Steps to Test	1. Start the system. 2. Select "Make Appointment". 3. Enter patient details, doctor, date, and time.
Pre-defined Function	Invalid Doctor Name
Post-defined Function	View Appointment by Date
Date of Testing	2024-11-08
Expected Outcome	Appointment is successfully created.

Actual Outcome	`Appointment booked successfully.`
	`Appointment Details: Appointment ID: 1, Date: Wednesday, Time: 11.00am,`
	`Status: BOOKED, Patient: A.A.M Aathif, Doctor: Dr. Brian,`
	`Treatment: Name: Skin Whitening, Price: LKR 7650.0`
	`==========`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Appointment is successfully created.

Table 9 Test Case 09 Appointment - Make Appointment

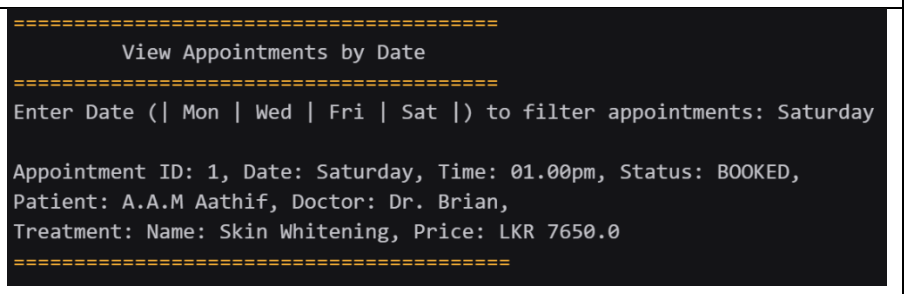
Test Case 10 – View Appointment by Date	
Test ID	TC – 010
Test Name	View Appointment by Date
Steps to Test	1. Start the system. 2. Select "View Appointment Details by Date". 3. Enter a specific date to view appointments.
Pre-defined Function	Make Appointment
Post-defined Function	Update Appointment Details
Date of Testing	2024-11-08
Expected Outcome	Appointment Found. and Display Appointment Details
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify list displays all entries for that date.

Table 10 Test Case 10 Appointment - View Appointment Details by Date

Test Case 11 – Update Appointment Details	
Test ID	TC – 011
Test Name	Update Appointment Details
Steps to Test	1. Start the system. 2. Select "Update Appointment Details". 3. Enter New Date. 4. Enter New Time. 5. Press Enter to Update Appointment.
Pre-defined Function	View Appointment by Date
Post-defined Function	Search Appointment by Name / Appointment ID
Date of Testing	2024-11-08
Expected Outcome	Appointment Updated Successfully.

Actual Outcome	`========================================`
	`Update Appointment Details`
	`========================================`
	`Enter Appointment ID to update: 1`
	`Enter New Date (| Mon | Wed | Fri | Sat |): Friday`
	`Enter New Time (e.g., 10:00am): 11.00am`
	
	`Appointment updated to Date: Friday, Time: 11.00am`
	`Appointment updated successfully.`
	`Updated Appointment Details: Appointment ID: 1, Date: Friday,`
	`Time: 11.00am, Status: BOOKED, Patient: A.A.M Aathif,`
	`Doctor: Dr. Ijlan, Treatment: Name: Acne Treatment, Price: LKR 2750.0`
	`========================================`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify that updated information reflects in Appointment records.

Table 11 Test Case 11 Appointment - Update Appointment Details

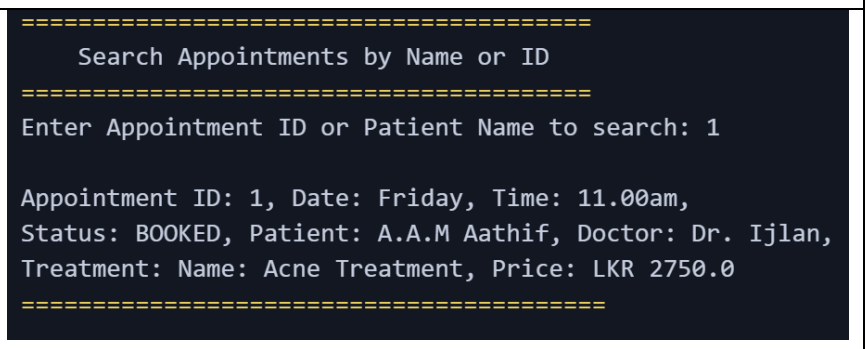
Test Case 12 – Search Appointment by Name / Appointment ID	
Test ID	TC – 012
Test Name	Search Appointment by Name / Appointment ID
Steps to Test	1. Start the system. 2. Select "Search Appointment by Name / ID". 3. Enter Appointment ID.
Pre-defined Function	Update Appointment Details
Post-defined Function	Cancel Appointment
Date of Testing	2024-11-08
Expected Outcome	Appointment Updated Successfully.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify that updated information reflects in Appointment records.

Table 12 Test Case 12 Appointment - Search Appointment by Name / Appointment ID

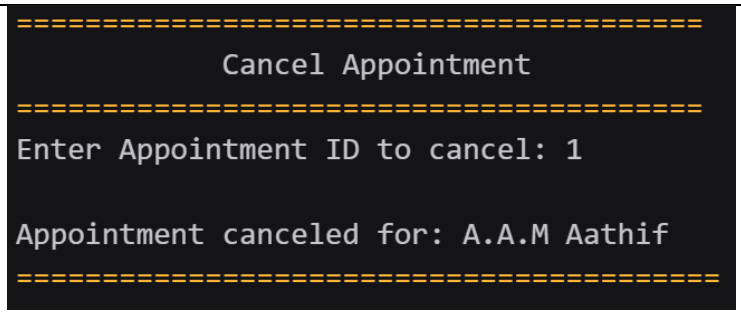
Test Case 13 – Cancel Appointment	
Test ID	TC – 013
Test Name	Cancel Appointment
Steps to Test	1. Start the system. 2. Select “Cancel Appointment”
Pre-defined Function	Search Appointment by Name / Appointment ID
Post-defined Function	Invalid Appointment Date
Date of Testing	2024-11-08
Expected Outcome	Appointment is successfully cancelled.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Check if appointment slot is freed up after cancellation.

Table 13Test Case 13 Appointment - Cancel Appointment

Test Case 14 – Invalid Appointment Date	
Test ID	TC – 014
Test Name	Invalid Appointment Date
Steps to Test	1. Start the system. 2. Select "Make Appointment". 3. Enter an invalid or past date.
Pre-defined Function	Cancel Appointment
Post-defined Function	View Appointment by Invalid Date
Date of Testing	2024-11-08
Expected Outcome	System displays error message about invalid date.

Actual Outcome	`========== Search for a Doctor ==========`
	`Enter Patient NIC: 200119803346`
	`Enter Date (| Mon | Wed | Fri | Sat |): Sunday`
	
	`Invalid Date. Please provide the correct Available Date`
	`==========`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with both past and non-date formats.

Table 14 Test Case 14 Appointment - Invalid Appointment Date

Test Case 15 – View Appointment by Invalid Date	
Test ID	TC – 015
Test Name	View Appointment by Invalid Date
Steps to Test	1. Start the system. 2. Select "View Appointment Details by Date". 3. Enter an invalid or past date.
Pre-defined Function	Invalid Appointment Date
Post-defined Function	Search Appointment by Invalid Appointment ID
Date of Testing	2024-11-08
Expected Outcome	System displays error message about invalid date.

Actual Outcome	`========== View Appointments by Date ==========`
	`Enter Date (| Mon | Wed | Fri | Sat |) to filter appointments: Saturday`
	`No appointments found for the date: Saturday`
	`==========`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify list displays all entries for that date.

Table 15 Test Case 15 Appointment - View Appointment by Invalid Date

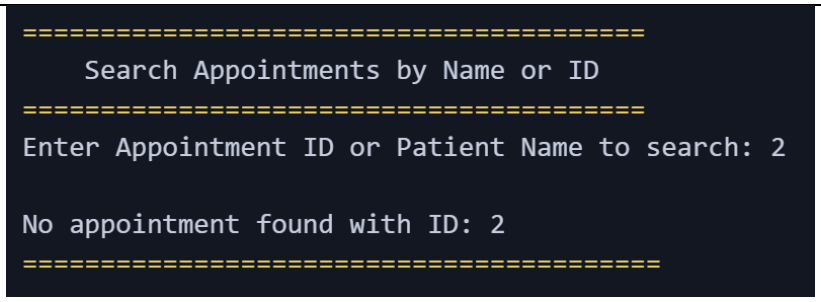
Test Case 16 – Search Appointment by Invalid Appointment ID	
Test ID	TC – 016
Test Name	Search Appointment by Name / Appointment ID
Steps to Test	1. Start the system. 2. Select "Search Appointment by Name / ID". 3. Enter Invalid ID.
Pre-defined Function	View Appointment by Invalid Date
Post-defined Function	Cancel Non-Existent Appointment
Date of Testing	2024-11-08
Expected Outcome	No Appointment found with ID.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify list displays all Appointment ID's.

Table 16 Test Case 16 Appointment - Search Appointment by Name / Appointment ID

Test Case 17 – Cancel Non-Existent Appointment	
Test ID	TC – 017
Test Name	Cancel Non-Existent Appointment
Steps to Test	1. Start the system. 2. Select "Cancel Appointment". 3. Enter Invalid ID.
Pre-defined Function	Search Appointment by Name / Appointment ID
Post-defined Function	Calculate Total Amount with Tax
Date of Testing	2024-11-08
Expected Outcome	No Appointment found with ID.
Actual Outcome	`========== Search Appointments by Name or ID ========== Enter Appointment ID or Patient Name to search: 2 No appointment found with ID: 2 ==========`
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify list displays all Appointment ID's.

Table 17 Test Case 17 Appointment - Cancel Non-Existent Appointment

4.2.4 Payment-Related Test Cases

Test Case 18 – Calculate Total Amount with Tax	
Test ID	TC – 018
Test Name	Calculate Total Amount with Tax
Steps to Test	1. Start payment for a treatment. 2. Select Treatment Type. 3. Press Enter to Generate Invoice.
Pre-defined Function	Cancel Non-Existent Appointment
Post-defined Function	Generate Invoice
Date of Testing	2024-11-08
Expected Outcome	Select Treatment Type, and confirmation is displayed.

Actual Outcome	`==========`
	`Search for a Doctor`
	`==========`
	`Enter Patient NIC: 200119803346`
	`Enter Date (| Mon | Wed | Fri | Sat |): Sunday`
	
	`Invalid Date. Please provide the correct Available Date`
	`==========`

Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with different treatment types if available.

Table 18 Test Case 18 Payment - Calculate Total Amount with Tax

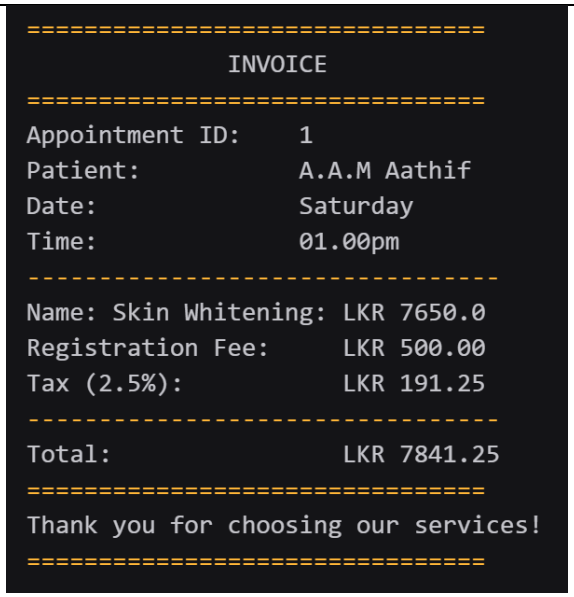
Test Case 19 – Generate Invoice	
Test ID	TC – 019
Test Name	Generate Invoice
Steps to Test	1. Enter Appointment ID. 2. Generate invoice.
Pre-defined Function	Calculate Total Amount with Tax
Post-defined Function	Generate Invoice by Invalid ID
Date of Testing	2024-11-08
Expected Outcome	Invoice is generated and saved.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Verify that the invoice includes all details.

Table 19 Test Case 19 Payment - Generate Invoice

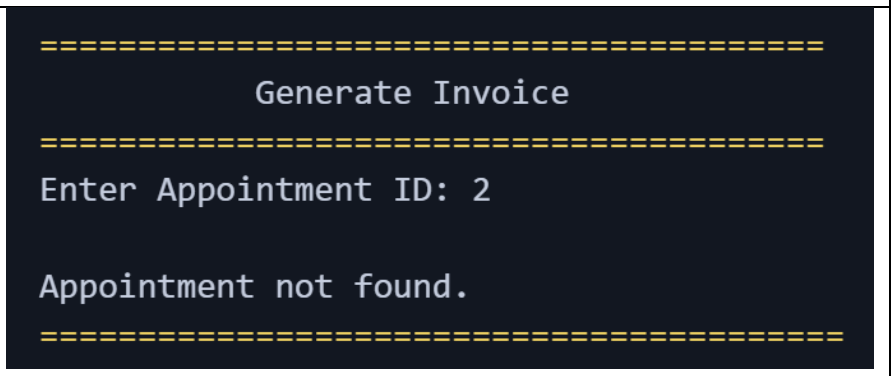
Test Case 20 – Generate Invoice by Invalid ID	
Test ID	TC – 020
Test Name	Generate Invoice by Invalid ID
Steps to Test	1. Select "Generate Invoice". 2. Enter Invalid Appointment ID.
Pre-defined Function	Generate Invoice
Post-defined Function	None
Date of Testing	2024-11-08
Expected Outcome	Appointment Not Found.
Actual Outcome	
Test Status	Pass
Name of the Tester	A.A.M Aathif
Comments	Test with non-existent Appointment IDs and invalid formats.

Table 20 Test Case 20 Payment - Generate Invoice by Invalid ID

4.3 Discussion: Key Challenges Encountered

4.3.1 NullPointerException when Accessing Patient / Doctor Information

Issue: The application encountered a **NullPointerException** *(Figure 45)* when attempting to access the name field of a Patient object that was null. This error occurred because the specified patient was not found in the system.

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field "name" because "this.patient" is null
        at Appointment.getDetails(AuroraSkinCareSystem.java:134)
        at AuroraSkinCareSystem.makeAppointment(AuroraSkinCareSystem.java:427)
        at AuroraSkinCareSystem.main(AuroraSkinCareSystem.java:303)
```

Figure 45 Error 01 - NullPointerException

Solution: To resolve this issue, the code was modified to check if the **Patient** object is **null** before accessing its **Name** field *(Figure 46)*. If the **Patient** object is **null**, an informative error message is displayed, indicating that the patient is not found and needs to be registered first.

```
if (patient == null) {
    System.out.println("Patient not found. Please register the patient first.");
    return;
}
```

Figure 46 Solution 01 - Null Check and Error Handling

With this solution, the program now displays a clear and informative error message *(Figure 47)* when the patient is not found, guiding the user to register the patient before making an appointment. This significantly improves the user experience and prevents the program from crashing due to unexpected null values.

```
========================================
            Make Appointment
========================================
Enter Patient NIC: 123456789V
Patient not found. Please register the patient first.
========================================
```

Figure 47 Output 01 - Informative Error Message

4.3.2 IndexOutOfBoundsException in Treatment Selection

Issue: The application encountered an **IndexOutOfBoundsException** *(Figure 48)* when the user provided an invalid treatment selection, such as entering "5" when only options 1-4 were available. This error disrupted the program's flow and displayed an uninformative error message.

```
Select Treatment (1-4): 6
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 5 out of bounds for length 4
        at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
        at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
        at java.base/java.util.Objects.checkIndex(Objects.java:361)
        at java.base/java.util.ArrayList.get(ArrayList.java:427)
        at AuroraSkinCareSystem.makeAppointment(AuroraSkinCareSystem.java:420)
        at AuroraSkinCareSystem.main(AuroraSkinCareSystem.java:303)
```

Figure 48 Error 02 - IndexOutOfBoundsException

Solution: To resolve this issue, the code was modified to validate the user's input and ensure that it falls within the valid range of treatment options *(Figure 49)*. This is achieved by using an if condition to check if the **treatmentChoice** is within the valid range (1 to **availableTreatments.size()**). If the input is invalid, an informative error message is displayed, and the method returns, preventing further execution with invalid input.

```
if (treatmentChoice < 1 || treatmentChoice > availableTreatments.size()) {
    System.out.println("Invalid treatment selection. Please try again.");
    return;
}
```

Figure 49 Solution 02 - Input Validation

With this solution, the program now displays a clear and informative error message *(Figure 50)* when the user enters an invalid treatment selection, guiding them to provide a correct input. This significantly improves the user experience and prevents the program from crashing due to unexpected input.

```
Select Treatment (1-4): 5
Invalid treatment selection. Please try again.
==========================================
```

Figure 50 Output 02 - Informative Error Message

4.3.3 InputMismatchException in Main Menu Selection

Issue: The application encountered an **InputMismatchException** *(Figure 51)* when the user provided invalid input, such as entering a string when an integer was expected. This error disrupted the program's flow and displayed an uninformative error message.

```
Select an option: ========================================
asas
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at AuroraSkinCareSystem.main(AuroraSkinCareSystem.java:293)
```

Figure 51 Error 03 - InputMismatchException

Solution: To resolve this issue, a **try-catch** block was implemented to gracefully handle the exception *(Figure 52)*. The try block attempts to read the user's input using **scanner.nextInt().** If an **InputMismatchException** arises, the catch block is executed, which prints a user-friendly error message and clears the invalid input from the input buffer. This ensures that the program can prompt the user for input again without causing further errors.

```
try {
    int option = scanner.nextInt(); // This line can throw InputMismatchException
    scanner.nextLine(); // Consume newline

    switch (option) {
        //......
    }

} catch (InputMismatchException e) {
    System.out.println("Invalid input. Please enter a number.");
    scanner.nextLine(); // Clear the invalid input
}
```

Figure 52 Solution 03 – Try-Catch Block

With this solution, the program now displays a clear and informative error message *(Figure 53)* when the user enters invalid input, guiding them to provide a correct input. This significantly improves the user experience and prevents the program from crashing due to unexpected input.

```
Select an option: ========================================
a@B.
Invalid input. Please enter a number.
```

Figure 53 Output 03 - Informative Error Message

4.4 Conclusion

The **Aurora Skin Care Clinic Management System** successfully integrates key functionalities to enhance patient and clinic interactions. Through effective management of patient registrations, appointment scheduling, treatment records, and payment processing, the system ensures a streamlined and user-friendly experience for both patients and clinic staff. Testing and validation have confirmed the system's ability to handle essential tasks such as booking appointments, generating invoices, updating patient records, and managing doctor schedules, demonstrating robustness and accuracy across multiple scenarios, including edge cases and invalid inputs.

Project outcomes include a fully functional console-based application that employs Object-Oriented Programming (OOP) principles, with well-structured classes and methods for each module. This structure allows for clear data flow and facilitates maintenance. The class diagram and underlying design decisions reflect a modular approach, promoting scalability for future enhancements, such as adding additional features or expanding into a GUI-based system.

Limitations identified during testing include constraints in appointment overlap management for high-demand doctors and limitations in handling complex payment structures, such as multi-session treatments or bundled services. Additionally, the system is currently console-based, which, while functional, may not provide the optimal user experience in a real-world scenario where a graphical interface would be beneficial.

4.5 Future Enhancements

1. Enhanced User Interface

- **GUI Implementation:** Develop a user-friendly graphical user interface (GUI) to replace the current text-based interface. This will improve the overall user experience and make the system more accessible to a wider range of users.
- **Intuitive Navigation:** Design a clear and intuitive navigation system within the GUI to allow users to easily access different functionalities.

2. Expanded Functionality

- **Doctor Management:** Add features to manage doctor information, including scheduling, availability, and specialization.
- **Treatment Expansion:** Incorporate additional treatment options, including their details, pricing, and duration.
- **Flexible Appointment Scheduling:** Allow for more flexible appointment scheduling, including options for recurring appointments and multiple appointments per day.
- **Online Booking Integration:** Integrate an online booking system to enable patients to book appointments directly through a website or mobile app.

3. Improved Error Handling and Input Validation

- **Robust Error Handling:** Implement comprehensive error handling mechanisms to prevent system crashes and provide informative error messages to the user.
- **Input Validation:** Validate user input to ensure data integrity and prevent invalid data entry.

4. Advanced Financial Tracking

- **Detailed Financial Reports:** Generate detailed financial reports, including revenue, expenses, and profitability analysis.
- **Payment Integration:** Integrate with payment gateways to allow for online payments and secure transactions.
- **Inventory Management:** Track inventory levels of medical supplies and generate automated reordering alerts.

(OpenAI, 2024)

Gantt Chart

Task	Start Date	End Date	Duration (Days)
Requirements Gathering	19/10/2024	25/10/2024	6
System Design: Class Diagram	26/10/2024	01/11/2024	6
Coding & Development	02/11/2024	07/11/2024	6
Functional Testing	08/11/2024	08/11/2024	1
Documentation & Report	09/11/2024	09/11/2024	1

Table 21 Gantt Chart

References

OpenAI, 2024. *ChatGPT (November 2024 version) [Large language model]*. [Online]
Available at: https://www.openai.com/
[Accessed 03 11 2024].