# MEDTRACK: A Medical Appointment and Records Management System

# Pedro Ramirez

# Assignment 1 4/29/2025C

For this assignment, you will describe and implement release 1 of your term project. You will appropriately incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*. Choose a demanding project that interests you—preferably for the whole term, but you can introduce an additional project later if necessary. The instructor and your facilitator will be happy to help with a topic. You are expected to build a challenging application because you will be leveraging AI to the maximum and we are interested in what you do with it. We expect you to select a project with much more scope than you can accomplish in the course. We will not require you to complete every aspect of it. What we do expect is that you specify and implement an additional set of demanding but do-able requirements each week.

For this assignment only, the application is not required to read input from a file: you can build all data into the code if you wish.

Leverage an AI generator such as ChatGPT as much as you can to create a real-world application. As described in the evaluation criteria below, your work will be assessed in terms of *your value added* (not simply on AI-generated material). Your value added consists of your choice of prompts together with your edits and additions to AI-generated material that result in capable and high quality code. Show your value added in red font and by means of explanations. For figures, insert comments (in red) that describe clearly your value added.

Please provide all code in text format, not in screenshots, so you can highlight in red your value added. If you performed significant prompt work, please note this in the relevant sections with added explanations. Accompany code and diagrams with explanations.

Leveraging AI at the start of a project is easy. Continuing to leverage it as the project grows, however, requires discipline and well-structured code—a skill we will instruct you on to gain your mastery.

Submit this completed Word document. Insert your material as indicated. Please observe and retain the gray text. Your materials—in black 12-point Times New Roman—should not exceed 5 pages excluding the gray instructions, references, figures, and appendices. Use the Appendix sections for additional material if you need to and refer to them in the document body. These will be read only on an as-needed basis.

Please develop in Eclipse—preferably—or else IntelliJ (talk to your facilitator about exceptions). As you code, use JUnit tests whenever possible—package-by-package, class-by-class, and method-by-method, except for trivial methods and those requiring I/O. Use testing classes for testing the latter. Keep the evaluation criteria in mind, listed at the end.

Housekeeping:

1. Include a ReadMe file that contains necessary execution notes and describing where to run the application from. All JUnit tests will be assumed runnable.
2. After you have completed the questions, make sure you have saved the file.
3. Please save this completed document with the file name: METCS622\_Assignment1\_FirstnameLastname.
4. To upload the completed Draft Assignment 1, click the "Browse My Computer" to upload your Word file, and then click "Submit".
5. Export your project from your IDE using its export feature and provide it as a second attachment.

# 1. SUMMARY DESCRIPTION

Give a one- or two-paragraph overall description of your proposed term project—half-page (12-point Times New Roman) limit. By the end, term projects will incorporate most of the techniques discussed in the course. To do this, you may need to alter the direction of your project or introduce an additional project in future, but a wide scope has better potential for incorporating techniques. You will probably find it useful to use a project acronym.

# MEDTRACK is a Java-based application for managing medical appointments, records, and user roles such as patient, doctor, and administrator. This project employs object-oriented principles, including inheritance and polymorphism, to simulate real-world interactions in a clinic or hospital setting. In this initial release, the system focuses on creating and identifying users (patients and doctors), viewing available doctors, and booking appointments. The abstract class User provides a common interface for patients and doctors, while polymorphism enables dynamic method behavior based on the role. The long-term scope includes file I/O, patient history management, and GUI integration. The application will be created with IntelliJ IDEA and will include JUnit tests. 2. I/O EXAMPLE FROM PROJECTED COMPLETED PROJECT

Provide an example of projected *concrete* output of the anticipated complete application—for example input. You will not be held to fulfilling exactly this—it is just explanatory at this point, to indicate where your project is going. We recognize that project direction and details will change as the term progresses. This section refers to the project as a whole, not just to what you will produce this week.

Welcome to MEDTRACK

Please select your role:

1. Patient

2. Doctor

3. Administrator

4. Register New User

> 1

Enter your Patient ID:

> P1001

Welcome back, John Doe.

Main Menu:

1. Book an Appointment

2. View Appointment History

3. Exit

> 1

Available Doctors:

[2001] Dr. Emily Smith – Family Medicine (no available)

[2002] Dr. Daniel Lee – Pediatrics (available)

Enter the doctor ID:

> 2002

Enter preferred appointment date (YYYY-MM-DD):

> 2025-05-10

Enter preferred time (HH:MM, 24-hour format):

> 14:00

Appointment confirmed:

John Doe with Dr. Daniel Lee on 2025-05-10 at 14:00.

Confirmation Code: APT-P1001-20250510-1400

Supply the [functional requirements](https://docs.google.com/document/d/1eU7eINLDxmrf793D4OF2yGT4ry_SW3GQGoVDYzecGHc/edit?usp=sharing) that you accomplished for this assignment, i.e., functionality that the application provides for the user. Please state requirement in declarative form, as illustrated in the examples, because we want to know the functionality intended (*what*, not *how*). For example, the following is *not* a proper functional requirement (it is a software design statement): *TicTac will have a class for O’s and a class for X’s.* It is common to mistake design elements like this for functional requirements. To get started, state what the application will accept as input.

See [here](https://docs.google.com/document/d/1zdqGKhQ1hn-Lk_fWY0mrxiyNP88kmOnr8gc3wYHKXGc/edit?usp=sharing) for how to state requirements and how to specify functions (and methods). These formats are required.

Keep in mind that the implementation of your requirements will incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*; that will probably influence the requirements you choose to implement in this assignment.

## 3.1 **Requirement Title**: Book an Appointment with a Doctor The system enables patients to book an appointment by selecting a doctor from a list, providing a preferred date and time, and confirming the booking. The program will validate the input, check for availability, and return a confirmation message with a unique appointment code.

### 3.2 **Requirement Title: Register a New User (Patient or Doctor)**

### The system shall allow a new user to register by entering their name, ID, and role (either “Patient” or “Doctor”). Based on the selected role, the system shall instantiate the appropriate subclass of User and store the instance in a central user registry. The registry shall allow subsequent retrieval using the user ID.

🟥 I used this requirement to demonstrate runtime polymorphism during object creation and applied upcasting by storing *Patient* and *Doctor* instances as *User* references in the registry.

# 4 ILLUSTRATIVE OUTPUT FROM (ACTUAL) IMPLEMENTATION

### Provide illustrative output from your implemented application (so far) showing that the requirements have been met. State what *class.method*(s) implement each requirement.

Registered user: John Doe (P1001)

Registered user: Dr. Daniel Lee (D2002)

Appointment confirmed:

John Doe with Dr. Daniel Lee on 2025-05-10 at 14:00

Appointment successfully booked!

- Your confirmation code is: APT-P1001-D2002-20250510-1400

🟥 Connected runtime behavior directly to class design. The confirmation code format was customized to simulate realistic system output.

# 5 YOUR DIRECTORY

Show a screenshot of your directory. This should include a parallel directory of JUnit tests where possible—package-by-package, class-by-class, and method-by-method, except for trivial ones.   
The screenshot below shows the structure of my Java project in IntelliJ. Classes are organized into model and service packages. Unit tests are located in the test folder and mirror the structure of the source code.

🟥 Separated domain and service logic into different packages and mirrored the test directory to reflect best practices in modular design.

A screenshot of a computer

AI-generated content may be incorrect.

# 6 TECHNIQUES IMPLEMENTED

Your implementation should include *inheritance*, *polymorphism*, and *either an abstract class or interface* at least once, and in a manner that is useful to your application. Explain where and how you applied these, using the format below.

### 6.1 Class model and Sequence Diagram

Identify where you included *inheritance*, *polymorphism*, and *abstract classes* or *interfaces* in your class model. Make classes and members *static* or not as per their intended usage. To do this use tools (e.g., Visio, Lucidchart, or draw.io), PowerPoint, or a combined model as in [this example](https://docs.google.com/spreadsheets/d/1wnua9en_nwBITm-pNiQTn7QzP1ylu9LK2QJeELa2GrA/edit?usp=sharing) (which you are free to cut and paste from). Insert indications in red to show where the three features below apply.  
  
See Class Diagram and Sequence Diagram: Provided on following page with value-added labels in red.

### A diagram of a computer AI-generated content may be incorrect.

### 6.2 Code showing an abstract class or interface

Show the relevant code (only) implementing this and explain why an abstract class or interface is appropriate here. It should be clear where the code is located (class and method).

### package model;

### public abstract class User {

### protected String id;

### protected String name;

### public User(String id, String name) {

### this.id = id;

### this.name = name;

### }

### public String getId() {

### return id;

### }

### public String getName() {

### return name;

### }

### public abstract String getRoleInfo(); // Polymorphic method

### }

### 🟥 Used an abstract class to define common identity and structure for both patients and doctors while enforcing a required method *getRoleInfo()*. 6.3 Code showing polymorphism

Show the relevant code (only) and explain why *polymorphism* is appropriate here. Recall that polymorphism is implemented in one of two ways – overriding methods in subclasses or overloading methods in the same class where the method signatures are different – and allowing the language runtime to dynamically invoke the correct method. It should be clear where the code is located (class and method)

// Patient.java

@Override

public String getRoleInfo() {

return "Patient: " + name + " (ID: " + id + ")";

}

// Doctor.java

@Override

public String getRoleInfo() {

return "Doctor: " + name + ", Specialty: " + specialty;

}  
  
User user = new Patient("P1001", "John Doe", "BlueCross");

System.out.println(user.getRoleInfo()); // Output based on actual subclass  
  
🟥 Overrode getRoleInfo() in both subclasses to enable runtime polymorphism based on actual object type. This method is invoked through a User reference to demonstrate dynamic dispatch.  
  
6.4 Code showing upcasting or downcasting

Show the relevant code (only) and explain why upcasting or downcasting is appropriate here. It should be clear where the code is located (class and method).

**Upcasting example:**  
User patient = new Patient("P1001", "John Doe", "BlueCross"); // upcasting

userRegistry.registerUser(patient); // Accepts User, stores all subclasses  
  
**Dowcasting example:**  
User user = userRegistry.findUserById("P1001");

if (user instanceof Patient) {

Patient p = (Patient) user; // downcasting

p.viewAppointments(); // Patient-specific method

}  
  
🟥 Used upcasting to store all users as *User* references in the registry, enabling a uniform interface. Used downcasting to regain access to subclass-specific behavior such as *viewAppointments()*.

## 7 EVALUATION OF ASSIGNMENT

🟥 Based on the AI-generated evaluation criteria and my own review, I believe this submission demonstrates strong use of abstraction, upcasting, and polymorphism. The UML diagram and I/O simulation reflect my added value in design clarity and runtime logic. I plan to continue building out the system in future releases with file I/O and role-based behavior for doctors and admins.



## Appendix 1 (if needed; should be referenced above, and will be read as-needed only)

## Appendix 2 (if needed; should be referenced above, and will be read as-needed only)

**Part 1 – Summary Description**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* o Clearly articulate how object-oriented features (e.g., inheritance, polymorphism) will solve specific technical challenges in your project.
* o Tie the long-term goals (e.g., GUI, file I/O, or API integration) to specific components of the system.
* o Mention how the system will grow incrementally in future releases and how early design decisions support that growth.

**Part 2 – I/O Example from Projected Completed Project**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* o Demonstrate how the system handles invalid or edge-case input (e.g., selecting a doctor with no availability).
* o Include interaction examples for all user roles (e.g., Doctor or Administrator actions).
* o Show system feedback when input validation fails, such as invalid dates or unavailable time slots.

**Part 3.1 – Book an Appointment with a Doctor**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* o Explain how doctor availability is determined and updated after booking.
* o Describe how appointment data is stored (e.g., in memory or with an Appointment object).
* o Include how the system ensures that time conflicts or duplicate bookings are avoided.

**Part 6.1 – Class Model and Sequence Diagram**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* o Include a detailed sequence diagram that clearly shows the runtime interaction for a specific use case (e.g., booking an appointment).
* o Use explicit red annotations to highlight where inheritance, polymorphism, and abstraction are applied and explain their purpose.
* o Ensure class relationships (composition, dependency, inheritance) are clearly distinguished with proper UML notation and direction.