5/20/2025

# Assignment 4

Pedro Ramirez

Implement the next release of your term project. You will incorporate *saving and retrieving objects*, *lambdas*, and *streams* into a very capable application. With your facilitator’s permission, other advanced techniques can be substituted for any of these concepts if you have already incorporated them. Your new requirements should be such that the techniques listed above are natural for their implementation.

The same instructions as for Assignment 3 apply, including leveraging AI, showing your value added in red font and by means of explanations, inserting comments (in red) in figures that describe clearly your value added, commenting on what you consider significant prompt contributions, and submitting.

## 1 SUMMARY DESCRIPTION

One- or two-paragraph overall description of your proposed term project. Color red the parts changed from previous assignments, if any.

MEDTRACK is a modular Java application that simulates essential operations in a medical clinic, including patient registration, doctor management, appointment booking, and record tracking. Built with object-oriented principles like encapsulation and polymorphism, the system maintains clean separation across model, service, and access layers.

In earlier releases, MEDTRACK introduced generics to unify patient and doctor management via UserRepository<T extends User> and a reusable CsvLoader<T> utility. These abstractions reduced duplication, improved extensibility, and enabled safe parsing of .csv files with exception handling and testable logic.

🟥 In this release, MEDTRACK incorporates object persistence by serializing appointments, patients, and doctors to .ser files. Users can now filter and view only upcoming appointments, leveraging streams, lambdas, and Java’s LocalDate API for date comparison and sorting. These enhancements deepen MEDTRACK’s real-world interactivity while showcasing modern Java features in a clean, test-driven design.

## 2 ADDITIONAL REQUIREMENTS IMPLEMENTED IN THIS RELEASE

Title and one or two sentences per requirement. Don’t repeat requirements implemented for prior assignments unless they are necessary to provide context—in which case, make it clear they are old.

2.1 Object Serialization of Core Data (NEW)

Appointments, patients, and doctors are now serialized and saved into .ser files, allowing the application to persist and reload data between runs without relying solely on CSVs.

🟥 Value added: Introduced serialization logic in AppointmentManager and updated FacadeService and Main to ensure persistence across sessions, creating a foundation for long-term data storage.

2.2 Upcoming Appointment Filtering (NEW)

Patients can now view only their upcoming appointments. This feature uses stream() and LocalDate to sort and filter future appointments dynamically.

🟥 Value added: Used Java Streams to compare each appointment’s date with the current date and display only future ones, improving clarity and real-world usability.

2.3 Doctor Name Lookup for Appointments (NEW)

When viewing appointments, patients now see the doctor’s name alongside the ID and confirmation code. This improves usability and clarity for the user.

🟥 Value added: Added doctor name resolution in the appointment viewing logic, fetching the Doctor object via the Facade layer and enhancing appointment readability.

2.4 Improved Load Handling for Malformed Records (OLD)

Maintained from a prior release: file loading methods now skip malformed lines individually and print contextual error messages without aborting the whole load.

🟥 Value added: Previously implemented granular error handling in loadUsersFromFile() using a generic parser function with try/catch logic on a per-line basis.

2.5 JUnit Use Case Test for Patient Lookup (NEW)

A use case-driven JUnit test was added to confirm successful appointment booking and retrieval, ensuring data integrity with minimal boilerplate.

🟥 Value added: Wrote a dedicated JUnit test covering the full lifecycle from booking to retrieval of an appointment, using real data files and including confirmation ID matching.

## 3. I/O SUPPORTING THE NEW REQUIREMENTS LISTED ABOVE

Provide examples of input / output generated by your application, showing clearly how the above requirements were implemented. “Input/output” refers to input and output from/to anywhere, including console, GUI, and secondary storage. Excellent assignments will typically include event-driven programming.

### Input 1

Your response replaces this.

### Output for 1

Your response replaces this.

### Input 2

Your response replaces this.

### Output for 2

Your response replaces this.

…

## 4. YOUR DIRECTORY

Show a screenshot of your directory. Include your “.dat” files (where objects are written) or JSON files. This should include JUnit tests—except for trivial and inappropriate ones.

Your response replaces this.

## 5. DESIGN

Supply a main use case, the class model, and the sequence diagram corresponding to the use case. These should be consistent. Indicate in red your class model where you applied object read (binary or JSON), object write, streams and lambdas.

Your response replaces this.

## 6. JUSTIFICATION AND CODE SNIPPETS

### 6.1 Justification for object read and write

Explain why your application’s requirements are best implemented with object read/write—whether binary or JSON.

Your response replaces this.

### 6.2 Code showing object read and write

Your response replaces this.

### 6.3 Justification for *stream*() and Lambdas (separate int0 6.m and 6.n if you wish)

Explain why your application’s requirements are best implemented with stream and lambdas

Your response replaces this.

### 6.4 Code Showing *stream*() and Lambdas (separate int0 6.m and 6.n if you wish)

Your response replaces this.

## 7 Evaluation

