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 version, I added the ability to save and load patients, doctors, and appointments using .ser files. Now the program can keep its data between sessions. I also used streams and lambdas to show only future appointments and sort them by date. This made the app feel more realistic and modern.

## 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: I updated the AppointmentManager, FacadeService, and Main classes to save all data into .ser files, so users won’t lose their information when the app closes. This helps make the system feel more complete.

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: I used Java Streams and LocalDate to show only future appointments. This way, patients don’t have to scroll through past ones, making the output easier to read.

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: Now the system shows the doctor’s fname when a patient checks their appointments. It uses the Facade layer to get the doctor’s info based on their ID.

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: I had already set up error handling that skips bad lines in CSV files and shows a clear message instead of crashing. This helps keep the app running even with bad input.

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: I added a new JUnit test that checks if a patient can successfully book and view an appointment. It uses real data and checks the confirmation code to make sure everything works.

## 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**

Register a new patient via the command-line interface:

==== MEDTRACK MENU ====

1. Register as new patient

2. Book an appointment

3. View my appointments

4. Exit

Choose an option: 1

Enter your name: Emma Thompson

Enter your insurance provider: WellCare

**Output for 1**

✅ Registered successfully! Your patient ID: P8792

🟥 Value added: After registering a new patient, their data is saved to both CSV and .ser files, so it’s available next time.

**Input 2**

Book an appointment with a doctor using patient ID:

==== MEDTRACK MENU ====

1. Register as new patient

2. Book an appointment

3. View my appointments

4. Exit

Choose an option: 2

Enter your patient ID: P8000

Available Doctors:

- ID: D8000 | Dr. Lee (Cardiology)

- ID: D8001 | Dr. Smith (Pediatrics)

Enter doctor ID to book with: D8000

Enter date (YYYY-MM-DD): 2025-12-12

Enter time (HH:MM): 12:00

**Output for 2**

✅ Appointment booked! Confirmation code: APT-P8000-D8000-20251212-1200

🟥 Value added: When a new appointment is created, it’s saved in memory and also written to a .ser file for later use.

**Input 3**

View patient appointments (with filtering and doctor name resolution):

==== MEDTRACK MENU ====

1. Register as new patient

2. Book an appointment

3. View my appointments

4. Exit

Choose an option: 3

Enter your patient ID: P8000

**Output for 3**

📅 Your Appointments:

- 2025-12-12 12:00 with Dr. Lee [ID: D8000] (Confirmation Code: APT-P8000-D8000-20251212-1200)

🟥 Value added: The program filters out past appointments and shows only upcoming ones, with doctor names included.

**Input 4**

Load malformed data from CSV (data/patients.csv):

P8000 | Maria Gomez | BlueCross

MALFORMED LINE

P8002 | John Adams | Aetna

**Output for 4 (Console log)**

⚠️ Skipped malformed line: MALFORMED LINE → Bad patient line: MALFORMED LINE

🟥 Value added: When the app reads a bad line in the CSV file, it prints a warning but keeps going instead of stopping.

### 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.

🟥 Directory structure demonstrates clear layering of the MEDTRACK application into model, service, and CLI components. Data files reflect real-time serialization and I/O flows. Tests are organized into model-level and use case-level coverage, aligned with best practices.

A screenshot of a computer

AI-generated content may be incorrect.

## 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.

#### **Use Case: Book Appointment + View Upcoming Appointments**

**Actors:**

* Patient
* System (MEDTRACK)

**Preconditions:**

* Patient and doctor data are loaded from .csv or .ser files.
* Doctor availability is not fully booked.

**Main Flow:**

1. The user enters their patient ID via the CLI.
2. The system finds the corresponding Patient object using UserRegistry.findUserById().
3. The system lists all available doctors (retrieved from UserRepository<Doctor>).
4. The user selects a doctor ID, date, and time.
5. The system checks availability using Doctor.isAvailable() via AppointmentManager.
6. If available, the appointment is booked and stored in memory and in the .txt and .ser files.
7. Later, the patient can return to view their upcoming appointments.
8. The system filters and sorts future appointments using stream().filter(...).sorted(...) and displays them with full doctor names.

**Postconditions:**

* The new appointment is stored persistently.
* The filtered list of upcoming appointments is shown, ordered chronologically.

#### **Class Model – Relevant Classes and Relationships**

model package

└── User (abstract)

├── getId(), getName(), getRoleInfo()

├── Patient extends User

│ ├── getInsuranceProvider(), addAppointment(), getAppointments()

└── Doctor extends User

├── getSpecialty(), isAvailable(), addAppointment(), getAppointments()

Appointment

├── patientId, doctorId, date, time

└── generateConfirmationCode()

service package

└── AppointmentManager

├── bookAppointment()

├── checkAvailability()

├── loadAppointmentsFromFile(Patient)

├── saveAppointmentsToBinaryFile(List<Appointment>)

└── loadAppointmentsFromBinaryFile(): List<Appointment>

└── UserRegistry

├── registerPatient(), registerDoctor()

├── findUserById()

├── loadUsersFromFile()

├── loadUsersFromCsv()

├── savePatientsToBinaryFile(), loadPatientsFromBinaryFile()

└── saveDoctorsToBinaryFile(), loadDoctorsFromBinaryFile()

└── CsvLoader<T>, CsvParser<T>, UserRepository<T extends User>

└── Used for generic loading and storage of patient/doctor data

app package

└── Main

├── CLI menu logic

├── Uses FacadeService for all operations

└── Uses stream().filter() + LocalDate to sort upcoming appointments

Link to diagram:

<https://docs.google.com/spreadsheets/d/1Srxutu-G_5bswR1xzbCi4zFVPmfwhTKby1di8F7EdTw/edit?gid=0#gid=0>

🟥 Value added: I added another flow to the use case where the patient can check their upcoming appointments. This part uses streams and date filtering. In the class diagram, I also marked where the system reads/writes objects and where lambdas/streams are used.

A white sheet with red text

AI-generated content may be incorrect.

## 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.

To ensure persistence across application runs, MEDTRACK now uses Java's built-in object serialization via ObjectOutputStream and ObjectInputStream to read and write appointments, patients, and doctors as .ser files. This approach allows the system to maintain state between sessions, eliminating the need for reloading from .csv or re-entering users manually.

🟥 Value added: I chose binary serialization because it’s built into Java and works well for saving complete objects. It’s simpler than using a database and keeps things in one file. It also helps when we want to keep a patient’s full data, like their past appointments.

### 6.2 Code showing object read and write

This is an excerpt from AppointmentManager.java that demonstrates how appointment data is saved to and loaded from a binary file using Java object serialization.

// 🟥 Saves a list of appointments to a .ser file

public void saveAppointmentsToBinaryFile(List<Appointment> appointments) {

try (ObjectOutputStream out = new ObjectOutputStream(

new FileOutputStream("data/appointments.ser"))) {

out.writeObject(appointments);

System.out.println("✅ Appointments saved to binary file.");

} catch (IOException e) {

System.err.println("❌ Failed to save appointments: " + e.getMessage());

}

}

// 🟥 Loads appointments from a .ser file (if present)

public List<Appointment> loadAppointmentsFromBinaryFile() {

try (ObjectInputStream in = new ObjectInputStream(

new FileInputStream("data/appointments.ser"))) {

return (List<Appointment>) in.readObject();

} catch (IOException | ClassNotFoundException e) {

System.err.println("⚠️ Could not load appointments from binary file: " + e.getMessage());

return new ArrayList<>();

}

}

Each of the model classes involved (Appointment, Patient, Doctor, and User) implements Serializable and declares a serialVersionUID for compatibility:

public class Doctor extends User implements Serializable {

private static final long serialVersionUID = 1L;

...

}

🟥 Value added: The file save/load code is only in AppointmentManager, so other parts of the app don’t have to deal with it. Using .ser files makes the app start up faster and makes it easier to restore saved data like appointment history.

### 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

Modern Java development encourages the use of the **Stream API** and **lambda expressions** to write cleaner, more expressive, and more concise code, particularly for collections and data transformation.

In the MEDTRACK system, stream operations allow us to:

* Filter upcoming appointments for a patient by comparing each appointment's date against the current date.
* Sort appointments based on their date and time.
* Improve readability by chaining transformations (e.g., filtering, sorting, mapping) in a declarative style rather than using nested loops.

This style is not only shorter but also eliminates boilerplate and offloads control-flow complexity to the Java runtime.

Additionally, lambda expressions were used in combination with the generic CsvLoader<T> to:

* Pass parsing logic as a first-class function.
* Support loading either patients or doctors with a single method, reducing duplication and improving maintainability.

🟥 **Value added**: I used streams to filter and sort appointments and lambdas to load data from CSV files. This made the code shorter and easier to read. It also helps if we want to support other user types later.

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

🟥 Stream-based Filtering and Sorting of Appointments

// Inside Main.java, case "3": View my appointments

List<Appointment> appointments = viewPatient.getAppointments().stream()

.filter(a -> {

LocalDate apptDate = LocalDate.parse(a.getDate());

return apptDate.isAfter(LocalDate.now()) || apptDate.isEqual(LocalDate.now());

})

.sorted(Comparator.comparing(Appointment::getDate).thenComparing(Appointment::getTime))

.toList();

if (appointments.isEmpty()) {

System.out.println("📭 No upcoming appointments found.");

} else {

appointments.forEach(a -> {

String doctorName = Optional.ofNullable(facade.findUserById(a.getDoctorId()))

.map(User::getName).orElse("Unknown");

System.out.println("- " + a.getDate() + " " + a.getTime()

+ " with " + doctorName + " [ID: " + a.getDoctorId() + "]"

+ " (Confirmation Code: " + a.getConfirmationCode() + ")");

});

}

🟥 Lambda Parser for Generic CSV Loading

// Inside UserRegistry.java

private CsvParser<Patient> patientParser = line -> {

String[] parts = line.split("\\|");

if (parts.length < 3) throw new IllegalArgumentException("Malformed patient line: " + line);

return new Patient(parts[0].trim(), parts[1].trim(), parts[2].trim());

};

private CsvParser<Doctor> doctorParser = line -> {

String[] parts = line.split("\\|");

if (parts.length < 3) throw new IllegalArgumentException("Malformed doctor line: " + line);

return new Doctor(parts[0].trim(), parts[1].trim(), parts[2].trim());

};

🟥 Value added: The stream code helps show only what’s needed (future appointments), and the lambda functions make it easy to load patients and doctors with the same method. This makes the app more organized.

## 7 Evaluation

This release of **MEDTRACK** significantly enhances the system’s usability, maintainability, and robustness by implementing persistent object storage, functional programming constructs, and test-driven validation. The newly introduced .ser-based object serialization supports seamless data recovery across sessions, while the use of stream() and lambda expressions simplifies the processing of appointment data and CSV parsing.

In terms of software design, the system now features a clear separation of concerns between data loading, business logic, and user interaction. Generic classes like CsvLoader<T> and UserRepository<T> increase code reuse and accommodate future extensibility (e.g., supporting roles like nurses or admins). Malformed input handling has also improved, ensuring resilience during data loading.

Functional correctness is backed by use case-driven JUnit tests, and the user interface has been refined with clearer appointment listings, showing both doctor names and confirmation codes. These improvements result in a more realistic, user-friendly simulation of a medical appointment system and demonstrate mastery of object serialization, lambda expressions, and Java streams.

🟥 **Value added**: In this version, I focused on keeping data between runs, making the appointment list more useful, and making the code easier to manage. These updates help the app feel more complete and ready for more features in the future.

### 🔧 **Files Modified or Added in This Release**

### ✅ **Files Modified**

* Appointment.java – added implements Serializable
* Patient.java – added implements Serializable
* Doctor.java – added implements Serializable
* User.java – added implements Serializable
* AppointmentManager.java – added:
  + saveAppointmentsToBinaryFile(List<Appointment>)
  + loadAppointmentsFromBinaryFile()
* UserRegistry.java – added:
  + savePatientsToBinaryFile()
  + saveDoctorsToBinaryFile()
  + loadPatientsFromBinaryFile()
  + loadDoctorsFromBinaryFile()
* AppointmentManagerTest.java – added:
  + testSaveAndLoadAppointments()
* UserRegistryTest.java – added:
  + testSaveAndLoadUsersBinaryFile()
* Main.java – modified:
  + Improved doctor name printing in appointment view
  + Added prompt to show “All” or “Upcoming” appointments
  + Implemented stream filtering and sorting by date/time

#### 🧪 Test Files Modified

* test/model/UserRegistryTest.java – added test for loading users and checking persistence
* test/model/AppointmentManagerTest.java – added test for saving/loading appointments

#### 🧪 New Test Files Added

* test/usecases/UpcomingAppointmentFlowTest.java – new use case test for verifying upcoming appointment filtering and output

🟥 Value added: I added tests that go through full appointment scenarios, including saving and loading data and filtering upcoming visits. This helps make sure all the new features work as expected in real situations.



✅ **AI PROMPT + RESPONSE + VALUE (Section 2.1, 5, 6.1, 6.2)**

🔷 **Part Number – 2.1 Object Serialization of Core Data**  
**Prompt:**  
For part 2.1, evaluate the implementation of object serialization. What would make this part an excellent solution according to the instructor’s expectations?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Clear explanation of why serialization is being used and what problem it solves.  
• ✅ Demonstrates correct use of Serializable interface on custom classes like Patient, Doctor, and Appointment.  
• ✅ Shows actual code snippets of saving and loading serialized data using ObjectOutputStream and ObjectInputStream.  
• ✅ Includes fallback logic or graceful handling when .ser files are not yet present.  
• ✅ Highlights performance or design considerations, such as avoiding serialization of transient fields or cyclic dependencies.  
**Value Added:**  
🟥 I used .ser files to store all the important objects like patients, doctors, and appointments. I also made sure the program can handle the file not being there yet, so it doesn't crash.

🔷 **Part Number – 2.2 Robust CSV Line-by-Line Parsing with Exception Handling (NEW)**  
**Prompt:**  
This part introduces improved CSV parsing with exception handling. What would make this implementation excellent according to the assignment’s evaluation criteria?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Demonstrates line-by-line parsing that isolates and reports malformed lines without halting execution.  
• ✅ Uses try/catch inside the loop and logs which line failed and why.  
• ✅ Separates parsing logic using a functional interface (e.g., CsvParser).  
• ✅ Shows extensibility by supporting both Patient and Doctor record loading with reusable generic code.  
• ✅ Validated by corresponding unit tests to ensure the method tolerates malformed data.  
**Value Added:**  
🟥 I rewrote the CSV loading function to handle each line one by one. If a line has an error, it shows a message but keeps going. This makes the app more stable when the input file has issues.

🔷 **Part Number – 2.3 Reusable Generic Repositories (NEW)**  
**Prompt:**  
This requirement introduces reusable, generic data repositories. What would distinguish this addition as excellent?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Introduces a type-safe, reusable UserRepository that avoids code duplication.  
• ✅ Clearly separates logic for adding, retrieving, and listing users.  
• ✅ Explains the design motivation: scalability for additional user types in future (e.g., nurses).  
• ✅ Unit-tested with both Doctor and Patient objects to confirm reuse.  
• ✅ Used meaningfully in UserRegistry to show real-world integration.  
**Value Added:**  
🟥 I created a generic UserRepository<T> class so I wouldn’t have to write the same code for patients and doctors. It makes the code shorter and easier to expand later if I add more user types.

🔷 **Part Number – 2.4 Stream and Lambda Filtering for Lookup (NEW)**  
**Prompt:**  
This part uses Java stream() and lambdas for user lookup. What would constitute an excellent implementation here?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Uses Stream.concat() and lambda filtering to find users by ID across multiple repositories.  
• ✅ Demonstrates clear functional code and minimizes imperative logic.  
• ✅ Used in UserRegistry.findUserById() to unify doctor/patient search.  
• ✅ Maintains readability while embracing modern Java idioms.  
• ✅ Verified through automated tests for both found and not-found cases.  
**Value Added:**  
🟥 I simplified the findUserById() method by using streams and filters. This made the code cleaner and helped me test both doctor and patient searches in one place.

🔷 **Part Number – 2.5 Doctor Name Resolution in Appointment Summary (NEW)**  
**Prompt:**  
This requirement displays the doctor’s name in the patient’s appointment summary. How could it be made excellent?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Enhances UX by showing doctor name in human-readable form, not just ID.  
• ✅ Uses facade.findUserById() to retrieve doctor object from ID.  
• ✅ Handles edge cases (e.g., doctor not found) gracefully.  
• ✅ Clearly marked in console output for visibility.  
• ✅ Demonstrated in real I/O and captured in test coverage or screenshot.  
**Value Added:**  
🟥 I changed the appointment display so it shows the doctor’s full name instead of just the ID. This makes it easier for patients to know who they’re seeing.

🔷 **Part Number – 5: DESIGN**  
**Prompt:**  
For Part 5, evaluate how to improve the UML diagrams and design documentation to meet excellent standards.  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Includes one use case diagram, one class diagram, and one sequence diagram—all consistent with each other.  
• ✅ Clearly marks areas using object persistence (e.g., .ser) in red.  
• ✅ Annotates class diagram with stream/lambda usage points.  
• ✅ Uses UML conventions correctly—e.g., associations, multiplicity, inheritance.  
• ✅ Provides brief but clear text notes (in red) explaining the added value of the new design elements.  
**Value Added:**  
🟥 I updated the diagrams to show where I added file saving and stream usage. I marked the classes and methods that now use .ser files and lambdas, so it’s easier to understand the changes.

🔷 **Part Number – 6.1 Justification for Object Read/Write**  
**Prompt:**  
This part of the assignment involves implementing object persistence. What qualities would elevate Part 6.1 to an excellent solution, and what justifications should be included?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Explains the need for persistence across sessions (e.g., appointments, users must not be re-entered each time).  
• ✅ Justifies why binary serialization was chosen over alternatives like JSON or databases.  
• ✅ Discusses the tradeoffs of .ser files (fast read/write, but not human-readable or cross-platform).  
• ✅ Connects serialization logic to real use cases, such as loading patient history or restoring doctor availability.  
• ✅ Mentions integration with Java’s Serializable interface and long-term design extensibility.  
**Value Added:**  
🟥 I explained that I used binary serialization because it’s simple and works well for keeping data between runs. It also fits with this project since I didn’t need a full database and wanted to keep things file-based.

🔷 **Part Number – 6.2 Code Showing Object Read/Write**  
**Prompt:**  
This section requires demonstrating object serialization in Java. What would a model answer look like for showing object read/write operations?  
**AI Response:**  
An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:  
• ✅ Shows both saveObjects() and loadObjects() methods using ObjectOutputStream and ObjectInputStream.  
• ✅ Demonstrates usage of try-with-resources for safety and auto-closing.  
• ✅ Provides real example usage (e.g., storing list of patients or appointments).  
• ✅ Includes error handling for missing/corrupted files.  
• ✅ Marks code regions with comments in red showing where object persistence was added.  
**Value Added:**  
🟥 I added save and load methods for patients, doctors, and appointments using ObjectOutputStream. If the .ser files don’t exist, the app uses the .csv files instead. This way, the app always has data to work with.