Assignment 5: Concurrency 5/28/2025

**Pedro Ramirez**  
**CS622 – Concurrency**

Implement the next release of your term project. You will incorporate *concurrency* if concurrency can fit appropriately (it usually can!); otherwise, create a different project with concurrency.

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 that changed from Assignment 4.Excellent assignments are usually event-driven.

MEDTRACK is a versatile Java application designed to simulate key workflows in a medical clinic—such as registering patients, booking appointments, and maintaining accurate records. With a strong focus on extensibility and real-world applicability, the system uses object-oriented principles like inheritance and polymorphism to organize its model and service layers effectively. In this release, MEDTRACK incorporates concurrency features to boost performance and reliability:

🟥 Appointment booking is now thread-safe, preventing double-booking when multiple patients attempt to book simultaneously.

🟥 Appointments are saved asynchronously to avoid delays from file writing.

🟥 A background autosave service periodically writes patient data to disk, safeguarding against crashes or abrupt shutdowns.

## 2 ADDITIONAL FUNCTIONAL 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 Concurrent Appointment Booking (NEW)**  
The system now supports multiple patients attempting to book the same appointment slot concurrently. To avoid race conditions and prevent double bookings, a ReentrantLock was introduced in the AppointmentManager. This ensures that only one booking is processed when several threads attempt to book the same doctor and time slot simultaneously—improving system reliability under high load and simulating real-world usage scenarios.

🟥 **2.2 Asynchronous Appointment Saving (NEW)**  
Once an appointment is successfully booked, it is no longer written to the file immediately. Instead, the appointment is added to a thread-safe queue. A dedicated background worker consumes the queue and writes the data to appointments.txt asynchronously. This non-blocking approach ensures a faster user experience and improves scalability by decoupling appointment persistence from the booking flow.

🟥 **2.3 Periodic Autosave of Patient Data (NEW)**  
A background autosave service now runs at regular intervals, writing **only patient** data to a binary file to protect against data loss. The autosave operates on a separate thread and activates only when new patients are registered. This silent background safeguard ensures fault tolerance without interfering with user interactions or requiring manual saves.

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

Provide input/output illustrating all new features of your application. Clearly demonstrate every new requirement and tie it back to the corresponding requirement.

🟥 **3.1 Concurrent Appointment Booking**  
To test concurrent booking, I used a demo (ConcurrentBookingDemo.java) where **10 patients simultaneously attempt** to book the same appointment slot with one doctor.

**Input (Simulated via threads):**

Each patient thread tries to book:

Date: 2025-06-10

Time: 10:00

Doctor ID: D0001

**Output:**

✅ Patient 3 successfully booked.

❌ Patient 1 failed: Doctor is already booked at this time.

❌ Patient 2 failed: Doctor is already booked at this time.

...

**Validation:**  
Only one patient receives a successful booking message. The rest are gracefully rejected, confirming the locking logic in AppointmentManager worked correctly.

🟥 **3.2 Asynchronous Appointment Saving**  
Tested using AsyncAppointmentSavingTest.java to ensure appointments are saved in the background after being booked.

**Input (Test case):**

manager.bookAppointment(patient, doctor, "2025-12-01", "08:30");

**Output (Printed to terminal):**

🟢 Appointment saver started...

🛑 Appointment saver stopped.

✅ Appointment was successfully saved by the background thread.

**Validation:**  
The test reads appointments.txt and confirms the new appointment entry exists with the correct date, time, doctor, and patient information—verifying the background thread persisted it successfully.

🟥 **3.3 Periodic Autosave of Patient Data**  
Demonstrated using RegistryAutosaveTestRunner.java, which registers a new patient and waits for autosave to trigger.

**Input (Manual simulation):**

Patient p1 = new Patient("P9999", "Test User", "TestCare");

facade.registerUser(p1);

**Output (after 8 seconds):**

📂 Loading patients...

🟢 Autosave started (interval = 5s)

👤 Registered new patient: Test User

⏱️ Autosave should have occurred.

**Validation:**  
The binary file patients.ser is updated automatically after the registration. No manual save call is required, and the autosave service stops cleanly after the test.

## 4 YOUR DIRECTORY

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

A screenshot of a computer

AI-generated content may be incorrect.

### ✅ **FILES ADDED OR MODIFIED IN RELEASE**

🟥 **2.1 Concurrent Appointment Booking**  
• **Modified:**

* AppointmentManager.java  
      ↳ Added ReentrantLock to ensure thread-safe appointment booking and prevent double bookings.  
  • **Created:**
* AppointmentManagerConcurrencyTest.java  
      ↳ JUnit test simulating 10 patients trying to book the same slot concurrently.
* ConcurrentBookingDemo.java  
      ↳ Manual runner that demonstrates booking race conditions in real time using multiple threads.

🟥 **2.2 Asynchronous Appointment Saving**  
• **Modified:**

* AppointmentManager.java  
      ↳ Integrated a background thread and queue-based system to offload appointment saving.  
  • **Created:**
* AppointmentSaverWorker.java  
      ↳ Background worker that consumes a thread-safe queue and writes appointments to file asynchronously.
* AsyncAppointmentSavingTest.java  
      ↳ JUnit test verifying that appointments are written to file correctly by the background thread.

🟥 **2.3 Periodic Autosave of Patient Data**  
• **Modified:**

* UserRegistry.java  
      ↳ Added a dirty flag and saveAllUsersToBinaryFile() method to support background autosave.
* Main.java  
      ↳ Started the autosave scheduler at app launch and ensured graceful shutdown with a hook.
* FacadeService.java  
      ↳ Exposed access to the internal UserRegistry to enable autosave logic externally.  
  • **Created:**
* RegistryAutosaveService.java  
      ↳ Periodic background scheduler using ScheduledExecutorService to silently persist patient data.
* RegistryAutosaveTestRunner.java  
      ↳ Manual runner that verifies patient autosave triggers correctly after new registrations.

🟥 **Documentation**  
• **Modified:**

* README.md  
      ↳ Updated to reflect concurrency enhancements in booking, autosave, and asynchronous persistence. Instructions and test coverage were expanded to match new features.

## 5 TECHNIQUES IMPLEMENTED

Integrate concurrency to tangibly improve the application’s scalability or real-world usability. Because you have AI at your disposal, we hold you to a standard that assumes its use: we reward well-engineered solutions rather than merely deducting points for errors.

**Technique:** Asynchronous Appointment Saving

This technique improves responsiveness by offloading the task of saving appointment data to a background thread. When a patient books an appointment, the system queues the save request rather than writing it immediately. A dedicated background worker thread processes the queue and writes the data to a file asynchronously.

**Use Case: Save Appointment in the Background**

**Actors:**

* Patient
* System (MEDTRACK)

**Preconditions:**

* A patient has successfully booked an appointment.
* AppointmentManager and AppointmentSaverWorker are initialized and active.

**Main Flow:**

1. The patient books an appointment using the CLI.
2. AppointmentManager.bookAppointment() creates the appointment object.
3. The appointment is passed to AppointmentSaverWorker.saveLater(...).
4. The background thread retrieves the appointment from the queue using BlockingQueue.take().
5. The thread writes the appointment to appointments.txt using FileWriter.
6. The main thread continues without blocking.

**Postconditions:**

* The appointment is persisted to the file asynchronously.
* The worker thread remains active to handle future save requests.

**Class Model (🟥 = Concurrency Applied)**

└── AppointmentManager 🟥

├── private final AppointmentSaverWorker saverWorker

├── private final ExecutorService executor

├── AppointmentManager()

│ → executor.submit(saverWorker)

├── bookAppointment()

│ → saverWorker.saveLater(...)

└── shutdown()

→ saverWorker.stop()

└── AppointmentSaverWorker 🟥 NEW

├── saveLater(Appointment, String, String)

│ → Adds to BlockingQueue

├── run()

│ → Continuously reads from queue and writes to file

├── stop()

│ → Gracefully terminates the loop

└── record AppointmentSaveRequest(Appointment, String, String)

📌 [Diagram is referenced here](https://docs.google.com/spreadsheets/d/1Srxutu-G_5bswR1xzbCi4zFVPmfwhTKby1di8F7EdTw/edit?gid=0#gid=0)

A screenshot of a computer

AI-generated content may be incorrect.

## ✅ 5.2 Explanation: Why Concurrency Is Appropriate

### Concurrency is suitable in this context because file I/O operations—like saving appointments—are relatively slow and would block the main thread if performed directly. This could cause noticeable delays for users, especially during peak times when multiple bookings occur in quick succession. By delegating the saving task to a background thread, the system remains fast, responsive, and scalable. This design reflects how real-world scheduling systems function in clinics and hospitals, where user interactions must remain fluid while backend processes manage persistence asynchronously.

### ✅ 5.3 Code Showing Where Concurrency Is Defined

Concurrency is established through the use of a lock, a thread-safe queue, and a background worker thread. The code defines how the appointment booking and saving logic are safely executed in parallel without blocking the main thread.

#### 🟥 Inside AppointmentManager.java

// Protects critical section during booking to prevent race conditions

private final ReentrantLock lock = new ReentrantLock();

// Background worker responsible for saving appointments

private final AppointmentSaverWorker saverWorker = new AppointmentSaverWorker();

// Executor used to run the background worker thread

private final ExecutorService executor = Executors.newSingleThreadExecutor();

// Constructor: starts the saver in a separate thread

public AppointmentManager() {

executor.submit(saverWorker); // 🟥 Launch background thread

}

// Gracefully shuts down the background worker and executor

public void shutdown() {

saverWorker.stop(); // 🟥 Signal the saver to terminate

executor.shutdown(); // 🟥 Shutdown the thread pool

}

#### 🟥 Inside AppointmentSaverWorker.java

// Queue shared between main thread and worker for save requests

private final BlockingQueue<AppointmentSaveRequest> queue = new LinkedBlockingQueue<>();

// Flag to control the background thread’s loop

private volatile boolean running = true;

// Called by AppointmentManager to enqueue save requests

public void saveLater(Appointment appointment, String patientName, String doctorName) {

queue.offer(new AppointmentSaveRequest(appointment, patientName, doctorName));

}

// Signals the worker to stop

public void stop() {

running = false;

}

// Worker thread: pulls from queue and writes appointments to file

@Override

public void run() {

System.out.println("🟢 Appointment saver started...");

while (running || !queue.isEmpty()) {

try {

AppointmentSaveRequest request = queue.take(); // Blocks until item is available

Appointment a = request.appointment();

try (FileWriter writer = new FileWriter("data/appointments.txt", true)) {

writer.write(a.getConfirmationCode() + " | " +

request.patientName() + " | " +

request.doctorName() + " | " +

a.getDate() + " | " +

a.getTime() + "\n");

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("⚠️ Saver thread interrupted");

} catch (IOException e) {

System.err.println("❌ Error saving appointment: " + e.getMessage());

}

}

System.out.println("🛑 Appointment saver stopped.");

}

// Record that encapsulates appointment save data

public record AppointmentSaveRequest(Appointment appointment, String patientName, String doctorName) { }

### ✅ 5.4 Code Showing Where Concurrency Is Used

Concurrency is applied at runtime to ensure thread-safe appointment booking and non-blocking file persistence. A ReentrantLock ensures only one thread can book an appointment at a time, while a background worker asynchronously saves appointment data.

#### 🟥 Inside AppointmentManager.java

public Appointment bookAppointment(Patient patient, Doctor doctor, String date, String time) {

lock.lock(); // 🟥 Begin critical section — ensures exclusive booking access

try {

// Check if the doctor is available at the given date and time

if (!doctor.isAvailable(date, time)) {

throw new InvalidInputException("Doctor is already booked at this time.", doctor.getName());

}

// Create the appointment object

Appointment appointment = new Appointment(patient.getId(), doctor.getId(), date, time);

// Update both doctor and patient with the new appointment

doctor.addAppointment(appointment);

patient.addAppointment(appointment);

// 🟥 Pass appointment to background saver — prevents main thread from blocking

saverWorker.saveLater(appointment, patient.getName(), doctor.getName());

return appointment;

} finally {

lock.unlock(); // 🟥 End critical section — other threads may now proceed

}

}

🔍 **Concurrency Highlights:**

* lock.lock() / lock.unlock() ensures thread-safe access to the booking logic, preventing race conditions.
* saverWorker.saveLater(...) enqueues the save operation, handled by a separate thread for responsiveness.

## 6 Evaluation

**Final Personal Note:**  
Overall, the use of concurrency in this release of MEDTRACK significantly improves the system's responsiveness and fault tolerance. The integration of multithreaded booking, asynchronous file writing, and scheduled autosaving reflects both real-world application needs and a thoughtful application of software engineering principles. Prompt-assisted brainstorming helped identify concurrency use cases, while I added robust shutdown handling and error recovery to improve resilience.



AI Prompts

✅ **Part 2.1 – Concurrent Appointment Booking (NEW)**  
**Prompt:**  
Describe how to implement concurrency in the current project. How would an excellent solution look?

**Answer:**  
To implement concurrency in the booking process, use a ReentrantLock in the bookAppointment method inside AppointmentManager. This ensures that when multiple patients try to book the same doctor and time slot, only one booking is allowed through. Set up a demo where 10 patients attempt to book at the same time, and the system correctly allows only one to succeed.

**An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:**  
• Clearly demonstrates the race condition with additional test cases or scenarios  
• Extends support for concurrent booking with multiple doctors and multiple time slots  
• Provides clear user feedback when a booking fails due to unavailability  
• Uses modular, reusable locking logic that could be applied to other shared data or services  
**(My idea)**  
• Adds logging or tracing of booking attempts for better debugging and visibility

🟥 **Value Added:**  
By making the booking process thread-safe, the system avoids double bookings and allows multiple users to try booking at the same time without issues. This makes it more reliable and realistic. 🟥 I also added logging to track each booking attempt, which helps with debugging and gives a clear view of what’s happening when many users interact with the system at once.

✅ **Part 2.2 – Asynchronous Appointment Saving (NEW)**  
**Prompt:**  
Describe another new functional requirement that we can implement in this release, including context if needed.

**Answer:**  
To avoid blocking the booking process with file writing, introduce a background worker that saves appointments asynchronously. When an appointment is booked, it is added to a thread-safe queue. A separate worker thread processes the queue and writes appointments to the file. This keeps the user experience smooth and models how a real-world system might handle high activity.

**An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:**  
• Automatically shuts down the worker thread when the program exits without requiring a manual call  
• Confirms and logs each save to provide assurance that the appointment was written  
• Handles a high volume of queued appointments without loss or delay  
• Prevents duplicate or corrupted entries through additional checks  
**(My idea)**  
• Includes error handling and retry logic if file writing fails (e.g., disk error, permission issues)

🟥 **Value Added:**  
Saving appointments in the background keeps the app fast and responsive, especially when many users are booking at once. 🟥 I also added error handling and retry logic in case file writing fails. This makes the system more reliable and protects appointments from being lost if something goes wrong during saving.

✅ **Part 2.3 – Periodic Autosave of Patient Registries (NEW)**  
**Prompt:**  
Is there another feature that I can add to my project using concurrency? Concurrent Appointment Booking and Asynchronous Appointment Saving are already implemented.

**Answer:**  
To improve fault tolerance and protect patient and doctor records from accidental loss, introduce a scheduled background task that periodically saves the user registries to file. This task runs on a separate thread using a ScheduledExecutorService. By automatically persisting data at fixed intervals, the system simulates how real-world applications guard against crashes or shutdowns. The regular, concurrent autosave of patient records ensures minimal data loss and allows the main thread to focus on handling user interactions.

**An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:**  
• Uses a ScheduledExecutorService to write registries to file every few seconds  
• Autosaves do not block the main thread or interfere with patient registration  
• Logs or prints confirmation after each autosave cycle  
• Skips unnecessary writes by detecting if data has changed since the last save  
**(My ideas)**  
• Shuts down the scheduler gracefully when the program exits  
• Handles file writing errors with retry logic or user-friendly error messages

🟥 **Value Added:**  
This feature improves reliability by safeguarding data in the background without interrupting the main flow. 🟥 I also suggested making sure the scheduler shuts down properly and handles file errors gracefully, which helps avoid crashes and makes the system safer and easier to maintain.