# Assignment 3 5/23/25

Implement the next release of your term project. **You will incorporate Java generics and file I/O in a fitting manner (i.e., where they are needed), as specified below.**

Leverage an AI generator such as ChatGPT as much as you can to **continue creating** 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. **To get credit for your prompt work, comment on what you consider significant prompt contributions.**

Accompany code and diagrams with explanations.

For functions, use the functionName(arguments) / INTENT / EXAMPLE / DEFINITIONS / PRECONDITIONS / POSTCONDITIONS format.

Your application must provide an interactive input mechanism, commonly a CLI or GUI. The user must be able to supply different values and responses without recompiling, relaunching, or editing source code.

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—**point out your contributions with these in mind.**

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, UPDATED VERSION**

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 using core object-oriented design principles such as inheritance, encapsulation, and polymorphism, the system maintains a clean separation of concerns across model, service, and access layers.

🟥 In this release, MEDTRACK introduces two significant generic components:

* UserRepository<T extends User>: a reusable type-safe structure used by UserRegistry to manage both patients and doctors.
* CsvLoader<T>: a file-loading utility that accepts a parsing strategy (CsvParser<T>) to generically deserialize user records from .csv files.

Together, these refactors eliminate redundant logic, reduce coupling, and prepare the system for future user types like nurses or administrators with minimal code changes. The loading methods now handle malformed lines safely using exceptions, and their behavior is fully validated by newly introduced use case tests.

🟥 Additionally, .txt files were formally renamed to .csv for clarity and industry alignment. Test coverage was expanded to include realistic booking scenarios, double-booking prevention, dynamic availability checks, and input validation flows. All functionality continues to be exposed through the CLI via a centralized FacadeService, ensuring clean architecture and user interaction.

What continues to distinguish MEDTRACK from a typical academic prototype is its real-world orientation and architectural foresight. The system anticipates future enhancements such as GUI integration (via JavaFX), patient dashboards, historical appointment views, and backend persistence via JDBC. This release strengthens the system’s foundation with robust generics, real-world input handling, and scalable design.

## 2 ADDITIONAL REQUIREMENTS IMPLEMENTED IN THIS RELEASE

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

#### **2.1 Title: Introduce a Generic User Repository (🟥 NEW REQUIREMENT)**

**Requirement (user-facing):**  
The system shall manage both patient and doctor records using a single, reusable data structure that ensures type safety and eliminates code duplication.

🟥 **Value Added:**  
I introduced a generic class UserRepository<T extends User> that encapsulates common operations such as addUser(), findById(), and getAllUsers(). This replaces previous logic that manually managed two separate List<Patient> and List<Doctor> collections in UserRegistry, improving maintainability and extensibility. The structure ensures that only valid user types are stored and sets the stage for future expansion to roles like nurses or administrators.

#### **2.2 Title: Refactor UserRegistry to Use Generics Internally (🟥 NEW REQUIREMENT)**

**Requirement (user-facing):**  
The system shall unify its internal patient and doctor storage under a generic mechanism, while preserving the public interface for external callers.

🟥 **Value Added:**  
I replaced internal list fields in UserRegistry with UserRepository<Patient> and UserRepository<Doctor>, without altering the external API. This decouples internal logic from the CLI and FacadeService, making the codebase more modular, maintainable, and testable.

#### **2.3 Title: Validate Integration of Generics Through Use Case Tests (🟥 NEW REQUIREMENT)**

**Requirement (user-facing):**  
The system shall demonstrate correct behavior across generic and non-generic layers through realistic, integrated test cases.

🟥 **Value Added:**  
I introduced a new usecases test package with comprehensive integration tests (AppointmentFlowTest, InvalidInputFlowTest, DoubleBookingTest, DoctorAvailabilityTest). These tests simulate complete user workflows, verifying that generic logic integrates correctly across all layers of the application, from CLI to persistence.

#### **2.4 Title: Load Users from CSV Using a Generic File Loader (🟥 NEW REQUIREMENT)**

**Requirement (user-facing):**  
The system shall support loading user data from .csv files using a generic strategy that works for multiple user types.

🟥 **Value Added:**  
I created a generic CsvLoader<T> utility that reads any CSV-style file and converts each line into a domain object using a passed CsvParser<T> strategy. This approach powers loadPatientsFromFile() and loadDoctorsFromFile() in UserRegistry, eliminating duplicate file-reading logic. It also supports error recovery by skipping malformed entries with descriptive logging. The system now loads from .csv files instead of .txt, aligning with industry conventions.

## 3 I/O EVIDENCE OF ACCOMPLISHING THE REQUIREMENTS LISTED ABOVE

Provide examples and explanations of actual input / output corresponding to the requirements above. **Include enough scenarios to make your accomplishment of the requirements entirely clear.**

Your response replaces this.

#### **Input File(s):**

* **patients.csv**P8000 | Maria Gomez | HealthPlus
* **doctors.csv**D8000 | Dr. Lee | Pediatrics
* **appointments.txt**APT-P8000-D8000-20251230-1430 | Maria Gomez | Dr. Lee | 2025-12-30 | 14:30

🟥 **Value Added:**These files are created and updated automatically through registerUser() and bookAppointment() calls via FacadeService. The format was updated to ensure time is saved as HH:MM, matching the expected input for robust parsing and validation.

#### **Console or CLI Input / Output:**

**Example CLI session:**

==== Welcome to MEDTRACK ====

1. Register new user

2. Book appointment

3. View my appointments

4. Exit

Enter your user ID: P8000

1. Book appointment

Enter Doctor ID: D8000

Enter date (YYYY-MM-DD): 2025-12-30

Enter time (HH:MM): 14:30

✅ Appointment booked! Confirmation code: APT-P8000-D8000-20251230-1430

🟥 **Value Added:**  
User commands interact only with the unified FacadeService. Invalid entries (e.g., bad date formats or unknown IDs) are handled gracefully through InvalidInputException, preventing crashes and providing clear feedback.

### Output File(s) (if applicable)

File: **appointments.txt**

APT-P8000-D8000-20251230-1430 | Maria Gomez | Dr. Lee | 2025-12-30 | 14:30

🟥 **Value Added:**   
The output format is tightly aligned with the parsing logic used by loadAppointmentsFromFile(). The refactor guarantees consistency between what is written and what can be reliably read and restored.

## 4 YOUR DIRECTORY

Continue to divide your code into well-named packages, each containing a singleton Facade object. If the package is named my.package, the Façade object should be named FacadeMyPackage. Obtaining the singleton object should be done with getTheInstance(). Access to functionality within each my.package should be only via myPackageAccess().

Your directory should include a parallel directory of JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones.

🟥 **Value Added:**

* This structure follows SOLID principles and clean layering:
  + model holds all domain entities.
  + service encapsulates logic and reusable helpers (e.g., CsvLoader<T>, UserRepository<T>).
  + app contains the CLI entry point (Main).
  + test/model targets unit-level testing, while test/usecases focuses on integration-level testing across flows.
* Files were renamed to .csv for consistency with real-world data processing tools.
* The generic loading logic is isolated in CsvLoader and validated in both functional and edge-case scenarios through new use case tests.

A screenshot of a computer

AI-generated content may be incorrect.

## 5 YOUR UPDATED CLASS MODEL AND CLARIFICATION OF HOW THE EXECUTION WORKS

Supply a main use case, the class model, and the sequence diagram corresponding to the use case. These should be consistent and clear. Indicate clearly in your class model where you applied generics. To do this use tools, PowerPoint, or a combine models as in [this example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing). Insert indications in red to show where generics apply.

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

A screenshot of a document

AI-generated content may be incorrect.

## 6 WHERE GENERICS ARE IMPLEMENTED

### 6.1 Class model fragment showing generic class(es)

Explain where and how you applied *generic classes* in your class model.

This release of **MEDTRACK** introduces multiple **Java generics** to eliminate redundancy and enforce type safety across the system. The following are the key classes and their generic structure as reflected in the updated class model:

#### 🟥 UserRepository<T extends User>

* **Purpose**: A reusable, type-safe container that stores users (Patient, Doctor, etc.) with methods like addUser(T user) and getAllUsers(): List<T>.
* **Class Model Fragment**:

Class

UserRepository<T>

+ addUser(T user)

+ getAllUsers(): List<T>

**Used in**: UserRegistry, which now declares:

private final UserRepository<Patient> patients = new UserRepository<>();

private final UserRepository<Doctor> doctors = new UserRepository<>();

**Why**: Previously, UserRegistry had hardcoded lists (List<Patient>, List<Doctor>). This change makes the registry extensible to future user types and enforces type safety at compile time.

#### 🟥 CsvLoader<T>

* **Purpose**: A generic loader utility that reads lines from a .csv file and maps them to typed domain objects.
* **Class Model Fragment**:

Class

|CsvLoader<T>

+ load(String, CsvParser<T>): List<T>   
 **Used in**: UserRegistry.loadUsersFromFile(...) to load both patients and doctors.

 **Why**: This abstracts the file loading logic away from domain-specific code and works with any type of object that has a corresponding CsvParser.

#### 🟥 CsvParser<T>

* **Purpose**: A functional interface for converting a CSV line into a typed object.
* **Class Model Fragment**:

Class

CsvParser<T>

+ parse(String): T

### **Why**: Makes the CsvLoader reusable for different domain objects like Patient or Doctor by supplying the correct parser lambda. **✅ Summary of Generics in the Model:**

* **Generic Classes Introduced**: UserRepository<T>, CsvLoader<T>, CsvParser<T>
* **Placement in Class Model**: Service package (used by UserRegistry)
* **Design Benefit**: Prevents duplication, ensures type safety, allows future extensibility (e.g., Nurse, Admin) without rewriting core logic.

### 6.2 Code (including test code), input (if applicable), and output showing generics and collections

#### Include at least two uses of Collection classes. **✅** Example 1: Generic User Repository

**Class**: UserRepository<T extends User>  
**Collection Used**: List<T>  
**Code**:  
  
package service;

import model.User;

import java.util.ArrayList;

import java.util.List;

public class UserRepository<T extends User> {

private final List<T> users = new ArrayList<>();

public void addUser(T user) {

users.add(user);

}

public List<T> getAllUsers() {

return users;

}

}

**Usage**:

private final UserRepository<Patient> patients = new UserRepository<>();

private final UserRepository<Doctor> doctors = new UserRepository<>();

🟥 **Value Added**:  
The use of List<T> enables UserRepository to store and retrieve any type of User, enforcing type safety and eliminating redundant code that previously managed patients and doctors separately.

**✅ Example 2: Generic CSV Loader and Parser  
Class**: CsvLoader<T>  
**Collection Used**: List<T>  
**Code**:  
package service;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.List;

import java.util.stream.Collectors;

public class CsvLoader<T> {

public List<T> load(String filename, CsvParser<T> parser) throws IOException {

return Files.readAllLines(Paths.get(filename)).stream()

.map(String::trim)

.filter(line -> !line.isEmpty())

.map(parser::parse)

.collect(Collectors.toList());

}

}

**Supporting Interface**:

@FunctionalInterface

public interface CsvParser<T> {

T parse(String line);

}  
  
**Usage**:  
loadUsersFromFile(

"patients.csv",

line -> {

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

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

},

this::registerPatient

);

#### 🟥 **Value Added**: This pattern enables loading any user type from a .csv file into a typed List<T> with minimal boilerplate and strong compile-time guarantees. It’s easy to test, extensible, and prevents accidental type mismatches during parsing. ✅ Input Example

**File**: patients.csv

P8000 | Maria Gomez | HealthPlus

P8001 | John Doe | BlueCross  
  
✅ Output Example (From Unit Test)

@Test

public void testGenericPatientLoad() {

UserRegistry registry = new UserRegistry();

registry.loadUsersFromCsv("patients.csv", UserRegistry.UserType.PATIENT);

List<User> all = registry.getAllUsers();

assertTrue(all.size() >= 1);

assertEquals("P8000", all.get(0).getId());

}

### 🟥 **Value added**

* **Tests the generics pipeline end-to-end: from file → parser → registration → memory.**
* **Confirms correctness of loadUsersFromCsv(...) with enum-based dispatching.**
* **Guards against regressions if someone changes the CSV format or registration logic.**
* **Ensures Patient-specific parsing works without needing a separate loadPatientsFromFile() method anymore.**

### 6.3 Explanation of generics use

Explain why your use of *generics* is appropriate here. Explain what the class model and code would be without i

In this release of the **MEDTRACK** system, generics were used to replace duplicate logic and enforce type safety across multiple components of the application. The central motivation was to **eliminate redundant code patterns** for managing Patient and Doctor objects, which both extend the abstract base class User.

#### ✅ Where Generics Were Used:

1. **UserRepository<T extends User>**:
   * Replaces previous separate List<Patient> and List<Doctor> fields in UserRegistry.
   * Provides shared methods like addUser(), findById(), and getAllUsers() that work for any subclass of User.
2. **CsvLoader<T> and CsvParser<T>**:
   * A generic file-loading utility that accepts a type-specific parser to convert raw CSV lines into domain objects like Patient or Doctor.
3. **Refactored UserRegistry.loadUsersFromFile(...)**:
   * Now takes a CsvParser<T> and Consumer<T> to abstract the parsing and registration logic for both Patient and Doctor.

### 🟥 What This Looks Like Without Generics:

Without generics, the code would contain:

* Duplicated classes or logic for managing patients and doctors (e.g., PatientRepository, DoctorRepository)
* Separate parsing logic in loadPatientsFromFile() and loadDoctorsFromFile()
* Type casting or instanceof checks scattered across methods
* Less modular and more error-prone code that’s harder to scale

For example, without generics:

private final List<Patient> patients = new ArrayList<>();

private final List<Doctor> doctors = new ArrayList<>();

public void registerPatient(Patient p) { patients.add(p); }

public void registerDoctor(Doctor d) { doctors.add(d); }

// ... then repeat for each method with two implementations

### ✅ Why This Use of Generics Is Appropriate:

* **Type Safety**: Generics ensure that only valid subclasses of User can be used with UserRepository<T>, avoiding runtime ClassCastExceptions.
* **Modularity**: Code can be extended to future user types (e.g., Admin, Nurse) by simply implementing a parser and reusing existing loading logic.
* **Maintainability**: Core methods like loadUsersFromFile and registerUser are defined once and reused across types, significantly reducing surface area for bugs.
* **Testability**: Tests like testGenericPatientLoad() verify functionality in a generalized way without knowing the type in advance.

### 🟥 Summary of Value Added by Generics

| **Feature** | **Without Generics** | **With Generics** |
| --- | --- | --- |
| Code Reuse | ❌ Copy-pasted logic for each type | ✅ Shared logic via UserRepository<T> |
| Safety | ❌ Runtime type casting | ✅ Compile-time type enforcement |
| Extensibility | ❌ Requires more boilerplate | ✅ Easily extend to new types |
| Testing | ❌ Separate test flows | ✅ One generalized testing structure |

This generic design improves not just this assignment but lays the foundation for a **robust, scalable, and future-proof architecture** — especially valuable if MEDTRACK is extended with new user roles or persistence layers like JDBC.

## 7 YOUR CODE

Unless your facilitator arranges another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request another transmission process (e.g., github).

## 8 INSTRUCTOR’S EVALUATION



## Appendix 1 (will be read as-needed only—add more as necessary)

AI Results:  
Propose improvements for sections **2.1**, **6.1**, and **a start to 6.2**. using ChatGPT

## 🔍 Assignment 3 Implementation Guide (For AI + Student Collaboration)

### ✅ 2.1 Introduce a Generic User Repository (NEW REQUIREMENT)

To achieve an **excellent implementation** of this requirement, implement the following:  
🟥 **Prompt 1:**  
"How can I refactor two separate repositories (Patient, Doctor) into a generic class using Java generics?"  
💬 **Value Added:**  
ChatGPT helped me design UserRepository<T extends User> with methods like addUser() and findById(). I refined the output to enforce type bounds and remove unnecessary instanceof checks from my original implementation.

🟥 **Prompt 2:**  
"What’s a clean way to read a CSV file into a generic type with a lambda parser?"  
💬 **Value Added:**  
I used ChatGPT to explore the use of CsvLoader<T> with a CsvParser<T> interface. I rewrote the original loop-based file loader into a Stream-based, type-safe, generic class, which now handles both Patient and Doctor files using a single loader.

🟥 **Prompt 3:**  
“How to design integration tests that validate a full booking lifecycle in a CLI Java app?”  
💬 **Value Added:**  
Based on ChatGPT’s structure suggestion, I wrote AppointmentFlowTest, which simulates real booking and verifies file output, reloading, and state persistence. I adjusted test order and added file cleanup between runs.

🟥 **Prompt 4:**  
"How to prevent tests from modifying production data in Java file I/O?"  
💬 **Value Added:**  
ChatGPT recommended test-specific file paths and cleanup using @BeforeEach. I integrated this into the usecases tests and added logic to clear appointments.txt, patients.csv, and doctors.csv when needed.

**Results:**

**Goals and Properties of an Excellent Solution:**

* ✅ Define a generic class UserRepository<T> with the bound T extends User to ensure type safety.
* ✅ Internally use a List<T> to store elements, supporting methods such as:
  + addUser(T user)
  + findById(String id): Optional<T>
  + getAllUsers(): List<T>
* ✅ Replace the previous hardcoded List<Patient> and List<Doctor> fields in UserRegistry with instances of UserRepository<Patient> and UserRepository<Doctor>.
* ✅ Ensure external methods in UserRegistry (like registerPatient() and getAllUsers()) remain unchanged so FacadeService and the CLI can continue functioning without modification.
* ✅ Show that the generic structure can support future subtypes like Admin or Nurse with minimal code changes.
* ✅ Include unit tests that verify patients and doctors are added, stored, and retrieved correctly using the generic repository.

**Suggestions for Going Beyond Minimum:**

* Show a comparison of "before vs. after" (how the code looked pre-generics vs. post-refactor).
* Include a diagram or fragment showing UserRepository<T> instantiated as UserRepository<Patient> and UserRepository<Doctor>.

### ✅ 6.1 Class Model Fragment Showing Generic Class(es)

To meet the expectations for this section:

**Key Elements for an Excellent Solution:**

* ✅ Add new class boxes for:
  + UserRepository<T>
  + CsvLoader<T>
  + CsvParser<T> (a @FunctionalInterface)
* ✅ Use red font or annotations to indicate which classes are generic and how the type parameter <T> is bounded or used.
* ✅ Show that:
  + UserRegistry now holds UserRepository<Patient> and UserRepository<Doctor>.
  + loadUsersFromFile() uses CsvParser<T> and Consumer<T> as generic functional parameters.
* ✅ Reflect how FacadeService interacts only with UserRegistry, while UserRegistry manages all generic logic.
* ✅ Include methods like addUser(), getAllUsers(), and load(String filename, CsvParser<T>) in the diagram for clarity.

**Suggestions to Go Further:**

* Illustrate polymorphism (e.g., getRoleInfo() called on User) to emphasize dynamic behavior.
* Show a generic method dispatching flow (e.g., how FacadeService → UserRegistry → UserRepository<T> works for a Patient).

### ✅ 6.2 Code, Input, and Output Showing Generics and Collections

This part is **in progress**, but an excellent submission would include:

#### A. Code Example 1 — Generic User Repository with Collections

* ✅ Define UserRepository<T extends User> using List<T> users = new ArrayList<>()
* ✅ Implement addUser(), getAllUsers(), and findById()
* ✅ Show it used in UserRegistry for both patients and doctors

#### B. Code Example 2 — Generic CSV Loader with Parser

* ✅ Implement CsvLoader<T> with:
* public List<T> load(String filename, CsvParser<T> parser)
* ✅ Use Java Streams and Collectors.toList() to load users from a CSV file

#### C. Test Input/Output Example

* ✅ Include sample patients.csv:
* P8000 | Maria Gomez | HealthPlus
* P8001 | John Doe | BlueCross
* ✅ Write a JUnit test that:
  + Loads the CSV using loadUsersFromCsv(...)
  + Asserts the loaded data contains P8000
  + Verifies that the repository stores patients correctly

#### D. Output/Console (Optional)

* ✅ Show output like:
* ✅ Loaded 2 patients from patients.csv

**Suggestions to Elevate the Work:**

* Add error handling for malformed CSV lines
* Show use of both patients and doctors through the same generic CSV loader
* Include a failing test (e.g., missing field) and describe how it's handled gracefully