Real Estate Listings App: Final Report

# Overview

This Java-based Real Estate Listings App is a console application designed to simulate basic property management features for users such as Admins, Agents, and Clients. The app demonstrates modern Java programming concepts including records, enums, pattern matching for sealed classes, streams, lambdas, regex validation, concurrency with ExecutorService, file I/O with NIO.2, and localization.

The application is built with **Java 21**. Although I did take into consideration adding **Java 22+ features** as suggested (e.g., unnamed patterns/variables), I did not use them in the final implementation due to compiler restrictions (e.g., usage of \_ as an identifier from Java 9 onwards).

**Key Java Features Used**

**1. Records**

**What:** Records are a special kind of class that function as immutable data carriers.

**Why:** Records reduce boilerplate code by automatically generating constructors, accessors, equals, hashCode, and toString. In this application, they represent core domain data like properties, users, and addresses. This enforces immutability and simplifies data handling, which is essential for reducing bugs and enhancing clarity.

**Used In:**

public record Address(String city, String country, String street, String zipCode) {}

public record Property(...)

public record User(String id, String name, Role role, String email, String password) {}

Records were used to represent Address, Property, and User data structures in a concise, immutable form.

**2. Enums**

**What:** Enums define a fixed set of constants.

**Why:** Using enums ensures data integrity when categorizing properties or user roles. This prevents errors from invalid string values and provides clarity when implementing role-based logic or property-type-specific functionality.

**Used In:**

public enum PropertyType { RENTAL, SALE, COMMERCIAL }

public enum Role { ADMIN, AGENT, CLIENT }

PropertyType and Role enums clearly define all possible categories and user roles.

**3. Sealed Interfaces & Pattern Matching**

**What:** Sealed interfaces limit which classes can implement them. Pattern matching simplifies the process of checking types and extracting data.

**Why:** Pattern matching eliminates the need for manual casting, and sealed interfaces ensure all listing types are known at compile-time. This leads to safer, more maintainable, and expressive code, which is crucial when branching logic based on property types.

**Used In:**

public sealed interface Listing permits RentalListing, SaleListing {}

switch (listing) {

case RentalListing r -> ...

case SaleListing s -> ...

}

This allows clean separation of logic for RentalListing and SaleListing and shows pattern matching in switch.

**4. Streams & Lambdas**

**What:** Streams process collections in a functional style. Lambdas are anonymous functions.

**Why:** **:** This pattern makes filtering and sorting data more readable and maintainable. It replaces nested loops and verbose code with elegant, declarative operations—critical in building a user-friendly property search experience.

**Used In:**

properties.stream()

.filter(p -> p.address().city().equalsIgnoreCase(city))

.sorted(Comparator.comparing(Property::price))

.forEach(System.out::println);

Used for filtering and sorting properties in a clean, functional manner.

**5. Regex for Validation**

**What:** Regex is used to define string patterns.

**Why:** This ensures email and password formats are secure and user-friendly. Validating input at this level prevents malformed data and enhances the user authentication process.

**Used In:**

Pattern.matches("^[\\w.-]+@[\\w.-]+\\.\\w{2,}$", email);

Pattern.matches("^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\\d).{6,}$", password);

Regex ensures only valid credentials are accepted for login.

**6. ExecutorService and Callables**

**What:** Used for running tasks concurrently.

**Why:** Simulating concurrent tasks helps emulate background services (e.g., loading messages, fetching leads) in a scalable way. It showcases Java’s concurrency features without complex thread management, vital for performance in real-world applications.

**Used In:**

ExecutorService service = Executors.newFixedThreadPool(3);

List<Callable<String>> tasks = List.of(...);

service.invokeAll(tasks);

Three concurrent tasks simulate workload (e.g., messages, recommendations, leads).

**7. NIO.2 File I/O**

**What:** Modern file input/output using java.nio.file.

**Why:** Writing data using Files.write and Path simplifies file operations and increases reliability. It enables exporting listings to a readable format, essential for reporting or data sharing.

**Used In:**

Path path = Paths.get("properties\_output.txt");

Files.write(path, properties.stream()

.map(p -> p.toString().replaceFirst("(\\d+\\.\\d{2}) EUR", currencyFormatter.format(p.price())))

.collect(Collectors.toList()));

Writes localized property listings to file using Path and Files.write().

**8. Localization**

**What:** Localization adapts the output format to user locale.

**Why:** Displaying prices in a localized format improves user experience across regions. It ensures currency symbols and formatting are contextually accurate for global users.

**Used In:**

Locale userLocale = Locale.getDefault();

NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance(userLocale);

Ensures currency is formatted based on user’s system locale.

**Conclusion**

My app effectively applies core Java 21 features to build a structured, modern, and maintainable console application. It balances object-oriented principles with functional programming patterns and demonstrates practical uses of streams, regex, sealed types, concurrency, file I/O, and localization. While Java 22+ features were not ultimately used due to syntax constraints, the app still demonstrates the core features of Java 21.