please generate 10 varied exercises around functional programming in java: sub topics: Lambdas, Method References, Optional, Stream, Functional Interfaces, Anonymous Classes should meet the following specifications: 1. one exercise should take about 40-50 minutes to complete (can be comprised of more than one clause 2. difficulty level should range from 6/10 to 8/10

**ChatGPT said:**

Here are 10 exercises covering different aspects of functional programming in Java, each designed to take about 40-50 minutes to complete. The difficulty ranges from **6/10 to 8/10**.

**1. Implement a Functional Data Processing Pipeline (Streams, Lambdas, Method References)**

**Difficulty: 7/10**  
**Task:**

* Create a Product class with fields: name, category, price, and rating.
* Create a list of Product objects.
* Implement a **data processing pipeline** that:
  1. Filters products with a rating above 4.
  2. Sorts them by price (ascending).
  3. Extracts only the names of the top 5 products.
  4. Converts them to uppercase.
  5. Collects the names into a **comma-separated string**.
* Use **method references** where possible.

**2. Custom Functional Interface for Data Transformation**

**Difficulty: 6.5/10**  
**Task:**

* Define a functional interface DataTransformer<T, R> with a method R transform(T input).
* Implement three lambda expressions using this interface:
  1. Converts a String to its **reverse**.
  2. Converts a List<Integer> to a **single concatenated String**.
  3. Converts a Double value to **a percentage format**.
* Use a generic method applyTransformation that applies any DataTransformer to an input.

**3. Advanced Stream Operations on Nested Collections**

**Difficulty: 7.5/10**  
**Task:**

* Create a Department class with String name and List<Employee> employees.
* Create an Employee class with String name, double salary, and boolean remoteWorker.
* Implement a **stream-based method** to:
  1. Find the department with the highest number of remote workers.
  2. Increase all salaries in that department by 10%.
  3. Return a sorted list of employee names (alphabetically) from that department.

**4. Implementing a Memoization Function (Lambdas, Functional Interfaces)**

**Difficulty: 8/10**  
**Task:**

* Create a **memoization function** that caches the results of expensive calculations.
* Define a functional interface Memoizer<T, R> that takes a function and stores results.
* Use it to memoize a Fibonacci function:

java

CopyEdit

int fib(int n)

* Ensure the function **only computes new values once** and retrieves cached results for repeated calls.

**5. Chaining Optional for Robust Data Retrieval**

**Difficulty: 6.5/10**  
**Task:**

* Create a User class with an Optional<Address> address.
* The Address class contains an Optional<String> city.
* Write a function that takes a User object and returns the city **using Optional chaining**.
* If any value is missing, return "Unknown City" instead.

**6. Implementing a Comparator Using Method References & Lambdas**

**Difficulty: 7/10**  
**Task:**

* Create a Book class with title, author, publicationYear, and price.
* Implement a Comparator<Book> in **three different ways**:
  1. Using a **lambda expression**.
  2. Using a **method reference**.
  3. Using an **anonymous class**.
* Sort a list of books by **price** (ascending), and then by **publication year** (descending).

**7. Functional Factory Method (Anonymous Classes, Functional Interfaces)**

**Difficulty: 7.5/10**  
**Task:**

* Define a Shape interface with void draw().
* Implement a ShapeFactory that returns different Shape instances:
  1. Using **anonymous classes**.
  2. Using **lambdas**.
* The factory should produce **Circle, Rectangle, and Triangle** instances dynamically.

**8. Parallel Stream Processing for Performance Optimization**

**Difficulty: 8/10**  
**Task:**

* Generate **1 million random integers** between 1 and 1000.
* Implement two different sum calculations:
  1. Using **sequential streams**.
  2. Using **parallel streams**.
* Measure and compare the execution times.
* Write a short analysis: **When should parallel streams be avoided?**

**9. Higher-Order Function for Retry Mechanism**

**Difficulty: 7.5/10**  
**Task:**

* Implement a **higher-order function** retry(Function<T, R> function, int attempts).
* The function should:
  1. Execute the given function.
  2. Retry up to attempts times if an exception occurs.
  3. If all attempts fail, return a **default fallback value**.
* Test it with a function that randomly fails **30% of the time**.

**10. Functional Approach to Event Handling System**

**Difficulty: 8/10**  
**Task:**

* Create an Event class with name, priority, and Runnable action.
* Implement an EventDispatcher class that:
  1. **Registers** events (using a Map<String, List<Event>>).
  2. Allows filtering events by **priority**.
  3. Executes all events using **method references**.
* Ensure the system supports **dynamic event addition & execution order based on priority**.

These exercises cover various aspects of **functional programming** in Java with **real-world applicability**. Would you like a GitHub-style structure to organize them? 🚀