Functional Programming in Java: A Practical Guide

A Practical Guide

Functional programming in Modern Java

Jitin Kayyala

This book is available at
https://leanpub.com/functionalprogramminginmodernjava

This version was published on 2026-01-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Part I — Why Functional Programming Matters in Java

Chapter 1. Why Java Developers Struggle with Complexity

For almost thirty years, Java's object oriented paradigm has worked well for the language.However, the combination of mutable state, dispersed responsibilities and implicit dependencies has made systems more challenging to understand.Differentiating between Accidental and Essential causes of complexity is necessary to understand why.

Accidental vs. Essential Complexity

Essential complexity is the inherent difficulty of the problem you're solving. If you're building a payment processing system, you genuinely need to handle currency conversion, fraud detection, and international regulations. These are irreducible.

Accidental complexity, by contrast, emerges from the tools and patterns we use to solve the problem. A payment processor doesn't require mutable state scattered across six objects connected through hidden references. It doesn't require deep inheritance hierarchies to model different payment types. These complexities are artifacts of our design choices, not the problem itself.

Consider a simple domain: customer orders. In a typical object-oriented design, you might structure it like this:

```java
public class Order {
    private String id;
    private Customer customer;
    private List<LineItem> items;
    private OrderStatus status;
    private BigDecimal totalPrice;

    public void addItem(LineItem item) {
        items.add(item);
        recalculateTotalPrice();
    }

    public void setStatus(OrderStatus newStatus) {
        status = newStatus;
    }

    private void recalculateTotalPrice() {
        totalPrice = items.stream()
            .map(item -> item.getPrice().multiply(BigDecimal.valueOf(item.getQuantity())))
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```

This design invites problems. If another part of the system reads `totalPrice` before `addItem` has finished, it sees stale data. If someone forgets to call `recalculateTotalPrice()` after modifying items directly, the total diverges from reality. The order knows how to modify itself, but nobody knows all the places that might modify it.

Mutability, Shared State, and Hidden Coupling

The root cause is mutability combined with shared references. When an object's state can change, every caller of that object must be aware of:

1. When the state might change
2. How other callers might have already changed it
3. Whether the change they're about to make contradicts earlier assumptions

This creates invisible coupling. A seemingly innocent change to one class can break behavior in systems far removed from the call site.

```java
// Somewhere in your code:
Order order = orderService.getOrder(123);
BigDecimal price = order.getTotalPrice(); // Gets $100

// Meanwhile, in another thread or callback:
order.addItem(luxuryItem);

// Your thread resumes:
payments.process(order, price); // Processing with stale price!
```

This isn't a bug in any single function—it's a structural problem created by mutable shared state.

Why "Just Add Setters" Stops Working

A common Java reflex is to encapsulate mutable state behind setters, believing this controls how objects change:

```java
public void setItems(List<LineItem> newItems) {
    this.items = newItems;
    recalculateTotalPrice();
}
```

This creates an illusion of control. The setter is one path to mutation, but the problem persists:

- Whoever holds a reference to the order can still call `addItem()` directly, bypassing your setter logic
- Multiple threads can interleave calls in unsafe ways
- Related fields (`totalPrice`, item count, validity rules) can fall out of sync
- Testing becomes a game of mocking all possible mutation sequences

Setters don't eliminate complexity; they redistribute it across your entire codebase.

FP as a Tool, Not a Religion

Functional programming offers a different approach: immutable data structures and pure functions that transform data without side effects. But this isn't a religious doctrine. Functional programming is a set of practical techniques for managing complexity.

The goal is not purity for its own sake. The goal is systems you can reason about, test thoroughly, and modify without fear. Functional programming is effective because:

1. **Immutability makes concurrency safe** – if data cannot change, there's no race condition
2. **Pure functions enable local reasoning** – you can understand a function by reading only that function
3. **Composition enables reuse** – you build complex behaviors by combining simple, well-understood pieces
4. **Testing becomes straightforward** – input □ function □ output, with no hidden state

Chapter 2. A Brief History of Functional Programming (Without the Math)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Why Functional Programming Didn't Catch On: A Multifaceted Answer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 3. Thinking in Data, Not Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part II — Core Functional Programming Concepts in Java

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 4. Immutability: The Foundation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 5. Pure Functions and Referential Transparency

Chapter 6. Functions as First-Class Citizens

Chapter 7. Composition Over Inheritance (For Real)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Composing Standard Functional Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part III — Functional Data Modeling in Modern Java

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 8. Records: Java's Missing Value Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 9. Sealed Types and Algebraic Thinking

Chapter 10. Null Is Not a Feature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part IV — Working with Collections Functionally

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 11. Streams: Power and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 12. Designing with Transformations, Not Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 13. Error Handling Without Exceptions Everywhere

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part V — Containing Side Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 14. Side Effects Are Inevitable—Chaos Is Optional

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 15. Dependency Injection, But Functional

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part VI — Introduction to Functional Optics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 16. The Real Problem: Updating Nested Immutable Data

Chapter 17. Lenses: Focused Reads and Writes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 18. Prisms: Working with Variants Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 19. Optionals, Lists, and Traversals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part VII — Functional Optics in Real Java Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 20. Implementing Optics with Records and Sealed Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 21. Integrating Optics into Existing Codebases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 22. Performance, Readability, and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Part VIII — Putting It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 23. A Real-World Case Study

Part IX — Beyond the Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 24. Libraries, Tools, and Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 25. How Far Should You Go?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Chapter 26. The Future of Functional Programming in Java

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/functionalprogramminginmodernjava.