Java Functional Programming

From Lambdas to Reactive Streams

readbytes.github.io

2025-07-06

This page is intentionally left blank.

Contents

1 Introduction to Functional Programming in Java **11**
- 1.1 What is Functional Programming? . . . 11
- 1.2 Benefits and Use Cases . . . 12
- 1.3 Overview of Functional Programming in Java . . . 13

2 Java 8 Functional Programming Basics **15**
- 2.1 Lambda Expressions: Syntax and Usage . . . 15
  - 2.1.1 Basic Syntax . . . 15
  - 2.1.2 Examples of Syntax Variations . . . 15
  - 2.1.3 Type Inference . . . 16
  - 2.1.4 Variable Capture: Effectively Final . . . 16
- 2.2 Functional Interfaces: `Function`, `Predicate`, `Consumer`, `Supplier` . . . 17
  - 2.2.1 `FunctionT, R` . . . 17
  - 2.2.2 `PredicateT` . . . 17
  - 2.2.3 `ConsumerT` . . . 18
  - 2.2.4 `SupplierT` . . . 18
  - 2.2.5 Real-World Usage Scenario . . . 18
- 2.3 Method References and Constructor References . . . 19
  - 2.3.1 Syntax Overview . . . 19
  - 2.3.2 Static Method Reference . . . 19
  - 2.3.3 Instance Method of a Particular Object . . . 20
  - 2.3.4 Instance Method of an Arbitrary Object of a Particular Type . . . 20
  - 2.3.5 Constructor Reference . . . 20
- 2.4 Example: Simple Calculator Using Lambdas . . . 21
  - 2.4.1 Highlights: . . . 22

3 Working with Functional Interfaces **24**
- 3.1 Creating Custom Functional Interfaces . . . 24
  - 3.1.1 The `@FunctionalInterface` Annotation . . . 24
  - 3.1.2 Defining a Custom Functional Interface . . . 24
  - 3.1.3 Example 1: Using the Custom Interface with a Lambda . . . 25
  - 3.1.4 Example 2: A No-Argument Functional Interface . . . 25
  - 3.1.5 Why Use Custom Functional Interfaces? . . . 26
- 3.2 Default and Static Methods in Interfaces . . . 26
  - 3.2.1 Default Methods . . . 26
  - 3.2.2 Static Methods . . . 27
  - 3.2.3 Combining Both . . . 27
- 3.3 Composing Functions and Predicates . . . 28
  - 3.3.1 Function Composition . . . 28
  - 3.3.2 Predicate Composition . . . 29
  - 3.3.3 Example 3: Function Predicate in a Real Use Case . . . 30
  - 3.3.4 Why Composition Matters . . . 30

3.4 Example: Filtering and Transforming Collections . . . 30
3.4.1 Full Example . . . 31
3.4.2 Expected Output . . . 32
3.4.3 Explanation . . . 32
3.4.4 Possible Variations . . . 32

4 Using Streams for Data Processing 34
4.1 Introduction to Streams API . . . 34
4.1.1 Streams vs. Collections . . . 34
4.1.2 Functional Programming with Streams . . . 34
4.1.3 Imperative vs. Declarative Example . . . 35
4.1.4 Conclusion . . . 35
4.2 Creating Streams: from Collections, Arrays, and I/O . . . 35
4.2.1 Streams from Collections . . . 36
4.2.2 Streams from Arrays . . . 36
4.2.3 Streams from I/O Sources . . . 37
4.2.4 Summary . . . 37
4.3 Intermediate Operations: `map`, `filter`, `sorted`, `distinct` . . . 37
4.3.1 `map()` Transforming Elements . . . 38
4.3.2 `filter()` Selecting Elements by Condition . . . 38
4.3.3 `sorted()` Ordering Elements . . . 38
4.3.4 `distinct()` Removing Duplicates . . . 39
4.3.5 Chaining Intermediate Operations . . . 39
4.3.6 Conclusion . . . 40
4.4 Terminal Operations: `forEach`, `collect`, `reduce` . . . 40
4.4.1 `forEach()` Iteration and Side Effects . . . 41
4.4.2 `collect()` Gathering Results . . . 41
4.4.3 `reduce()` Aggregating Values . . . 42
4.4.4 Summary . . . 43
4.5 Example: Processing Employee Data with Streams . . . 43
4.5.1 Employee Class and Sample Data . . . 44
4.5.2 Explanation . . . 45
4.5.3 Expected Output . . . 45

5 Advanced Stream Operations 47
5.1 Parallel Streams and Performance . . . 47
5.1.1 How Parallel Streams Work . . . 47
5.1.2 Potential Benefits . . . 47
5.1.3 Common Pitfalls and Considerations . . . 47
5.1.4 When to Use Parallel Streams . . . 48
5.1.5 Measuring Performance: Sequential vs. Parallel . . . 48
5.1.6 Conclusion . . . 49
5.2 Short-circuiting Operations: `limit`, `findFirst`, `anyMatch` . . . 49
5.2.1 `limit(long maxSize)` . . . 49
5.2.2 `findFirst()` . . . 49

5.2.3 `anyMatch(PredicateT)` . . . 50
5.2.4 Summary . . . 50
5.2.5 Why Use Short-circuiting? . . . 50
5.3 FlatMap for Nested Data . . . 51
5.3.1 Difference Between `map` and `flatMap` . . . 51
5.3.2 Example 1: Flattening a List of Lists . . . 51
5.3.3 Example 2: Extracting Nested Fields from Objects . . . 52
5.3.4 When to Use `flatMap` . . . 53
5.3.5 Summary . . . 53
5.4 Example: Processing Nested Collections . . . 54
5.4.1 Sample Data and Full Example: . . . 55
5.4.2 Expected Output: . . . 57
5.4.3 Summary . . . 57

6 Optional and Handling Nulls Functionally **59**
6.1 Using `Optional` to Avoid NullPointerExceptions . . . 59
6.1.1 Philosophy behind `Optional` . . . 60
6.2 Common `Optional` Operations . . . 60
6.2.1 Common Pitfalls . . . 62
6.2.2 Summary Table of Key Methods . . . 62
6.3 Example: Safely Navigating Complex Object Graphs . . . 63
6.3.1 Explanation: . . . 66
6.3.2 Benefits: . . . 67

7 Functional Error Handling **69**
7.1 Using `Either`, `Try` Patterns (Custom Implementations) . . . 69
7.1.1 Conceptual Model . . . 69
7.1.2 Simple Custom Implementation of `Either` . . . 69
7.1.3 Simple Custom Implementation of `Try` . . . 72
7.1.4 Benefits of `Either` and `Try` . . . 74
7.1.5 Summary . . . 74
7.2 Function Composition with Error Handling . . . 74
7.2.1 The challenge: chaining computations that might fail . . . 75
7.2.2 Using `Either` for composition with `flatMap` . . . 75
7.2.3 Example: Composing functions with `Either` . . . 75
7.2.4 Using `Try` for safe composition . . . 76
7.2.5 Recovering from errors . . . 77
7.2.6 Summary . . . 77
7.3 Example: Validating User Input Functionally . . . 78
7.3.1 Explanation . . . 82
7.3.2 Sample Output . . . 82

8 Functional Design Patterns **85**
8.1 Strategy Pattern with Lambdas . . . 85
8.1.1 Summary . . . 88

8.2 Command Pattern using Functional Interfaces . . . 89
8.2.1 Summary . . . 91
8.3 Observer Pattern with Functional Callbacks . . . 92
8.3.1 Summary . . . 94
8.4 Example: Event Handling in GUI Applications . . . 94
8.4.1 Explanation . . . 95
8.4.2 Why functional event handling? . . . 96

9 Higher-Order Functions and Currying 98
9.1 Defining and Using Higher-Order Functions . . . 98
9.1.1 Summary . . . 99
9.2 Partial Application and Currying in Java . . . 99
9.2.1 Summary . . . 101
9.3 Example: Building Reusable Validation Functions . . . 101
9.3.1 Explanation . . . 102
9.3.2 Benefits . . . 103

10 Lazy Evaluation and Infinite Streams 105
10.1 Understanding Lazy vs. Eager Evaluation . . . 105
10.1.1 Summary . . . 107
10.2 Creating and Using Infinite Streams . . . 107
10.2.1 Summary . . . 109
10.3 Example: Generating Fibonacci Numbers Lazily . . . 109
10.3.1 Key Takeaways . . . 110

11 Monads and Functional Data Structures 112
11.1 Understanding Monads in Java Context . . . 112
11.1.1 Summary . . . 113
11.2 Using Optional as a Monad . . . 113
11.2.1 Summary . . . 115
11.3 Introduction to Functional Collections . . . 115
11.3.1 Summary . . . 116
11.4 Example: Chaining Operations Safely with Monads . . . 117
11.4.1 Example: Chaining Operations Safely with Monads . . . 117
11.4.2 Explanation . . . 118
11.4.3 Benefits . . . 118

12 Concurrency and Functional Programming 120
12.1 Using CompletableFuture with Functional Style . . . 120
12.1.1 Functional Methods in `CompletableFuture` . . . 120
12.1.2 Example: Building a Functional Asynchronous Workflow . . . 120
12.1.3 Explanation . . . 121
12.1.4 Benefits of Functional Style . . . 122
12.1.5 Summary . . . 122
12.2 Parallel Stream Pitfalls and Best Practices . . . 122

12.2.1 Common Pitfalls of Parallel Streams . . . 122
12.2.2 Best Practices for Using Parallel Streams . . . 124
12.2.3 Summary . . . 124
12.3 Reactive Programming Overview with Functional Concepts . . . 124
12.3.1 Core Concepts in Reactive Programming . . . 124
12.3.2 Functional Principles in Reactive Programming . . . 125
12.3.3 Reactive Libraries in Java . . . 125
12.3.4 Summary . . . 125
12.4 Example: Asynchronous Data Fetching . . . 126
12.4.1 Scenario . . . 126
12.4.2 Code Example (Using `CompletableFuture`) . . . 126
12.4.3 Explanation . . . 127
12.4.4 Benefits of Functional Asynchronous Composition . . . 127

13 Functional Programming in Collections and APIs 129
13.1 Using Streams for Collection Manipulation . . . 129
13.1.1 Why Streams? . . . 129
13.1.2 Core Stream Operations . . . 129
13.1.3 Benefits of Functional Collection Processing . . . 130
13.1.4 Summary . . . 131
13.2 Functional Interfaces in Java APIs (e.g., `Comparator`, `Runnable`) . . . 131
13.2.1 `Comparator<T>` Functional Sorting . . . 131
13.2.2 `Runnable` Deferred Execution . . . 132
13.2.3 `Callable<V>` Asynchronous Computation with Return . . . 132
13.2.4 `Supplier<T>` Deferred Value Generation . . . 132
13.2.5 Summary . . . 133
13.3 Example: Sorting and Grouping Complex Data . . . 133
13.3.1 Code Example . . . 133
13.3.2 Explanation . . . 135
13.3.3 Benefits of Functional Style . . . 135

14 Functional Programming in Data Processing 137
14.1 Processing Files and I/O Functionally . . . 137
14.1.1 Reading Files with `Files.lines()` . . . 137
14.1.2 Example: Reading and Processing a Text File . . . 137
14.1.3 Key Points . . . 138
14.1.4 Handling Exceptions Functionally . . . 138
14.1.5 Aggregating and Collecting . . . 138
14.1.6 Summary . . . 139
14.2 Parsing and Transforming JSON Data . . . 139
14.2.1 Functional JSON Parsing with Jackson . . . 139
14.2.2 Example: Mapping JSON Array to Objects and Transforming . . . 139
14.2.3 Functional Transformations on JSON Trees . . . 140
14.2.4 Using Gson with Functional Patterns . . . 140
14.2.5 Summary . . . 141

14.3 Example: Building a CSV Parser with Streams . . . 141
14.3.1 Runnable CSV Parser Example . . . 141
14.3.2 Sample `people.csv` File . . . 142
14.3.3 Explanation . . . 143
14.3.4 Benefits of this Functional Approach . . . 143

15 Building Domain-Specific Languages (DSLs) 145
15.1 What is a DSL? . . . 145
15.1.1 Why Use DSLs? . . . 145
15.1.2 Internal vs External DSLs . . . 145
15.1.3 Characteristics of Good DSLs . . . 145
15.1.4 Simple Illustrative Example . . . 146
15.1.5 Summary . . . 146
15.2 Using Lambdas to Create Fluent APIs . . . 146
15.2.1 Fluent APIs and DSLs . . . 147
15.2.2 How Lambdas Help . . . 147
15.2.3 Builder Pattern and Method Chaining . . . 148
15.2.4 Function Composition . . . 149
15.2.5 Summary . . . 149
15.3 Example: Building a Simple Query DSL . . . 150
15.3.1 Defining the DSL . . . 150
15.3.2 Explanation . . . 151
15.3.3 Benefits of this DSL Approach . . . 151

16 Testing Functional Code 154
16.1 Unit Testing Lambdas and Functional Interfaces . . . 154
16.1.1 Challenges in Testing Lambdas . . . 154
16.1.2 Best Practices for Testing Functional Code . . . 154
16.1.3 Testing Simple Functional Interfaces . . . 154
16.1.4 Verifying Behavior with Mocks . . . 156
16.1.5 Summary . . . 156
16.2 Using Mocks and Stubs in Functional Context . . . 156
16.2.1 Why Use Mocks and Stubs? . . . 157
16.2.2 Using Mockito to Mock Functional Interfaces . . . 157
16.2.3 Strategies for Pure and Predictable Tests . . . 158
16.2.4 Summary . . . 159
16.3 Example: Testing Stream Pipelines . . . 159
16.3.1 Example: Stream Pipeline to Process Employee Names . . . 159
16.3.2 Explanation . . . 160
16.3.3 Summary . . . 161

17 Appendix 163
17.1 Common Functional Interfaces and Usage Examples . . . 163
17.1.1 Choosing the Right Interface . . . 164
17.1.2 Summary . . . 165

17.2 Writing Efficient Functional Java Code 165
17.2.1 Summary 166
17.3 Common Pitfalls and How to Avoid Them 167
17.3.1 Summary 168

Chapter 1.

Introduction to Functional Programming in Java

1. What is Functional Programming?
2. Benefits and Use Cases
3. Overview of Functional Programming in Java

1 Introduction to Functional Programming in Java

1.1 What is Functional Programming?

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. Unlike traditional imperative programming, where you write step-by-step instructions to change the program's state, functional programming focuses on what to compute rather than how to compute it.

At its core, functional programming relies on a few key principles that set it apart:

1. **Immutability** In functional programming, data is immutable, meaning once created, it cannot be changed. Instead of modifying existing data, you create new data structures that reflect the changes. Think of it like a frozen ice cube—once frozen, it doesn't melt or change shape. If you need a different shape, you make a new ice cube. This immutability helps prevent bugs caused by unexpected side effects, making programs easier to understand and reason about.

2. **First-Class and Higher-Order Functions** Functions are treated as first-class citizens, which means they can be assigned to variables, passed as arguments to other functions, and returned from functions. This flexibility allows you to build programs by combining small, reusable functions. For example, just like you can pass ingredients to a recipe or use a tool for multiple tasks, functions can be passed around and composed to perform complex operations.

3. **Pure Functions** Pure functions always produce the same output for the same input and do not cause side effects (like modifying a global variable or printing to the console). This predictability makes pure functions easier to test, debug, and parallelize. Imagine a vending machine: pressing a button (input) always delivers the same snack (output) without altering anything else inside the machine.

To better understand the difference between functional and other paradigms, consider how you might update a bank account balance:

- In an **imperative** style, you might explicitly instruct the program to subtract an amount from the current balance stored in memory.
- In an **object-oriented** style, you would send a message (method call) to the account object to update its internal state.
- In a **functional** style, you would create a new account balance based on the old balance and the amount, without modifying the original balance directly.

This approach reduces complexity by eliminating hidden changes and shared state, making functional programs more predictable.

Functional programming is gaining popularity in Java and many other languages because it helps write code that is concise, modular, and easier to reason about. As you explore this book, you will see how these concepts unlock powerful ways to build robust and maintainable

applications.

1.2 Benefits and Use Cases

Functional programming offers several compelling advantages that make it increasingly popular in modern software development.

One of the primary benefits is **easier reasoning about code**. Because functional programs rely on pure functions—functions that always produce the same output given the same input and have no side effects—developers can understand and predict behavior without worrying about hidden changes in state. This predictability reduces bugs and makes debugging simpler. Imagine you're troubleshooting a calculation in a spreadsheet: if the formula always produces the same result, you can isolate problems faster.

Another key advantage is **better modularity and composability**. Functional programming encourages breaking problems down into small, reusable functions that can be composed together like building blocks. This modular approach makes it easier to develop, test, and maintain code. For example, in a data processing pipeline, you might chain together functions to filter, transform, and aggregate data without rewriting logic or creating complex conditional flows.

Functional programming also **improves concurrency support**. Since functional code avoids mutable shared state, it naturally eliminates many issues related to concurrent modification and race conditions. This makes writing multi-threaded or parallel programs safer and more straightforward. For instance, when processing large datasets, you can use parallel streams to speed up execution without worrying about synchronization bugs.

Use Case: Data Processing Pipelines Imagine an application that analyzes large volumes of customer transactions. Using functional programming, you can build a pipeline that filters suspicious transactions, maps transactions to relevant summary objects, and reduces the data to compute totals—all expressed in a clear, concise sequence of functions. This pipeline can easily be parallelized, improving performance on multi-core processors.

Use Case: Event-Driven Systems In event-driven architectures, functional programming helps by representing event handlers as pure functions or functional callbacks. This leads to more predictable and testable event processing, reducing side effects that often cause bugs in complex systems.

Overall, functional programming helps developers write clean, reliable, and scalable code, particularly well suited to modern challenges like big data, real-time applications, and distributed systems. As you progress in this book, you'll discover how these benefits translate into practical coding patterns in Java.

1.3 Overview of Functional Programming in Java

Java introduced functional programming features relatively late compared to some other languages, with major additions arriving in **Java 8** (released in 2014). Prior to this, Java was primarily an object-oriented language, relying heavily on classes and imperative constructs. However, the growing need for more expressive, concise, and parallelizable code led to the integration of functional programming concepts into the language.

The most significant additions that enable functional programming in Java are:

- **Lambda Expressions**: These provide a clear and concise way to represent anonymous functions (functions without names). Lambdas simplify the creation of small function objects, replacing verbose anonymous inner classes used in earlier Java versions.
- **Functional Interfaces**: Java defines a set of interfaces with a single abstract method, such as `Function<T, R>`, `Predicate<T>`, `Consumer<T>`, and `Supplier<T>`. These interfaces serve as targets for lambda expressions and method references, allowing functional-style programming while maintaining strong type safety.
- **Streams API**: Introduced as a powerful tool for processing collections in a functional manner, streams enable chaining of operations like filtering, mapping, and reducing. Streams support lazy evaluation and can be easily parallelized to improve performance.
- **Method References**: A syntactic shortcut that allows existing methods or constructors to be used where a lambda expression is expected, improving code readability.

These features seamlessly integrate with existing Java syntax and libraries, making it possible to adopt functional programming gradually without abandoning object-oriented design.

Here is a simple example of a lambda expression that filters a list of strings to include only those starting with the letter "J":

```
List<String> names = Arrays.asList("John", "Alice", "Jack", "Bob");
List<String> jNames = names.stream()
                           .filter(name -> name.startsWith("J"))
                           .collect(Collectors.toList());

System.out.println(jNames); // Output: [John, Jack]
```

In this example, the lambda `name -> name.startsWith("J")` defines a predicate used by the `filter` operation. This concise syntax demonstrates how functional programming enhances readability and expressiveness in Java.

As you move through this book, you will explore these features in greater depth and learn how to harness the power of functional programming in Java effectively.

Chapter 2.

Java 8 Functional Programming Basics

1. Lambda Expressions: Syntax and Usage
2. Functional Interfaces: `Function`, `Predicate`, `Consumer`, `Supplier`
3. Method References and Constructor References
4. Example: Simple Calculator Using Lambdas

2 Java 8 Functional Programming Basics

2.1 Lambda Expressions: Syntax and Usage

Lambda expressions, introduced in Java 8, provide a concise way to represent anonymous functions—blocks of code that can be passed around and executed later. They simplify what used to require verbose anonymous inner classes, making your code cleaner and easier to read.

2.1.1 Basic Syntax

A lambda expression consists of three parts:

- **Parameters**: The input parameters, similar to method parameters.
- **Arrow token**: `->` separates parameters from the body.
- **Body**: The code to execute, which can be a single expression or a block of statements.

General form:

```
(parameters) -> expression
```

or

```
(parameters) -> { statements; }
```

2.1.2 Examples of Syntax Variations

1. **No parameters** If the lambda takes no arguments, use empty parentheses:

```
Runnable r = () -> System.out.println("Hello, world!");
r.run();
```

This prints `Hello, world!`. Here, the lambda implements the `Runnable` interface's single method `run()`.

2. **Single parameter without parentheses** If there is exactly one parameter, parentheses can be omitted:

```
Consumer<String> printer = name -> System.out.println("Name: " + name);
printer.accept("Alice");
```

This prints `Name: Alice`.

3. **Multiple parameters** If there are multiple parameters, parentheses are required:

```
BiFunction<Integer, Integer, Integer> adder = (a, b) -> a + b;
System.out.println(adder.apply(3, 5)); // Outputs 8
```

4. **Block body with multiple statements** When the body has more than one statement, use braces `{}` and explicit `return` if needed:

```
Function<Integer, String> converter = num -> {
    int doubled = num * 2;
    return "Result: " + doubled;
};
System.out.println(converter.apply(4)); // Outputs "Result: 8"
```

2.1.3 Type Inference

Java can usually infer parameter types from context, so explicit types are optional:

```
Predicate<String> isEmpty = s -> s.isEmpty();
```

But you can specify types if you prefer:

```
Predicate<String> isEmpty = (String s) -> s.isEmpty();
```

2.1.4 Variable Capture: Effectively Final

Lambdas can access variables from their enclosing scope, but only if those variables are **effectively final**—meaning their value does not change after assignment. This restriction prevents unexpected side effects and keeps lambdas predictable.

Example:

```
int factor = 2;
Function<Integer, Integer> multiplier = x -> x * factor;
System.out.println(multiplier.apply(5)); // Outputs 10
```

Here, `factor` is captured by the lambda. Trying to modify `factor` after the lambda is defined will cause a compilation error.

By using lambda expressions, you transform bulky anonymous classes into clean, expressive code blocks. They are a fundamental building block of Java's functional programming capabilities.

2.2 Functional Interfaces: `Function`, `Predicate`, `Consumer`, `Supplier`

Java 8 introduced a set of standard **functional interfaces** in the `java.util.function` package. These interfaces define single abstract methods and serve as targets for lambda expressions and method references. The four core interfaces—`Function`, `Predicate`, `Consumer`, and `Supplier`—cover the most common functional programming use cases: transforming data, filtering, performing side effects, and supplying values.

2.2.1 `Function<T, R>`

A `Function` takes an input of type `T` and produces a result of type `R`. It represents a transformation or mapping operation.

- **Method:** `R apply(T t)`

Example: Mapping Strings to their lengths

```
Function<String, Integer> lengthFunction = s -> s.length();
List<String> names = List.of("Alice", "Bob", "Charlie");

List<Integer> lengths = names.stream()
                             .map(lengthFunction)
                             .toList();

System.out.println(lengths); // Output: [5, 3, 7]
```

Here, the `Function` maps each string to its length.

2.2.2 `Predicate<T>`

A `Predicate` tests a condition on an input of type `T` and returns a boolean indicating if the input matches the condition.

- **Method:** `boolean test(T t)`

Example: Filtering even numbers

```
Predicate<Integer> isEven = num -> num % 2 == 0;
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

List<Integer> evens = numbers.stream()
                             .filter(isEven)
                             .toList();

System.out.println(evens); // Output: [2, 4, 6]
```

This `Predicate` filters the stream to include only even numbers.

2.2.3 `ConsumerT`

A `Consumer` performs an action on an input of type `T` but does **not** return a result. It's typically used for side effects like printing or modifying external state.

- **Method:** `void accept(T t)`

Example: Printing each element

```
Consumer<String> printer = s -> System.out.println("Name: " + s);
List<String> fruits = List.of("Apple", "Banana", "Cherry");

fruits.forEach(printer);
```

This prints each fruit name prefixed by "Name:".

2.2.4 `SupplierT`

A `Supplier` produces or supplies a value of type `T` without taking any input. It's useful for lazy value generation or deferred computation.

- **Method:** `T get()`

Example: Supplying a random number

```
Supplier<Double> randomSupplier = () -> Math.random();

System.out.println("Random number: " + randomSupplier.get());
System.out.println("Random number: " + randomSupplier.get());
```

Each call to `get()` generates a new random number.

2.2.5 Real-World Usage Scenario

Imagine processing a list of orders. You could use:

- A `Function<Order, Double>` to extract the total price.
- A `Predicate<Order>` to filter only completed orders.
- A `Consumer<Order>` to log each order's details.
- A `Supplier<Order>` to generate sample test orders on demand.

Together, these interfaces let you build expressive, modular, and reusable pipelines that handle data transformation, filtering, side effects, and deferred generation elegantly.

These core functional interfaces are foundational to Java's functional programming style and integrate seamlessly with streams and other APIs. Mastering their use will empower you to write concise and powerful code.

2.3 Method References and Constructor References

Method references are a shorthand syntax in Java for writing lambda expressions that simply call an existing method. They improve code readability by eliminating boilerplate when the lambda's body is just a method call. Introduced in Java 8, method references are closely related to lambdas and can be used wherever a lambda expression is expected.

2.3.1 Syntax Overview

The general syntax of a method reference is:

`ClassName::methodName`

This replaces a lambda like `x -> ClassName.methodName(x)` when the method call matches the functional interface's signature.

There are **four main types** of method references:

2.3.2 Static Method Reference

Syntax: `ClassName::staticMethod`

Equivalent Lambda: `x -> ClassName.staticMethod(x)`

Example:

```
Function<String, Integer> parseInt = Integer::parseInt;
System.out.println(parseInt.apply("42")); // Output: 42
```

This is equivalent to:

```
Function<String, Integer> parseInt = s -> Integer.parseInt(s);
```

2.3.3 Instance Method of a Particular Object

Syntax: `instance::instanceMethod`

Example:

```
Consumer<String> printer = System.out::println;
printer.accept("Hello, method reference!"); // Output: Hello, method reference!
```

Equivalent lambda:

```
Consumer<String> printer = s -> System.out.println(s);
```

2.3.4 Instance Method of an Arbitrary Object of a Particular Type

Syntax: `ClassName::instanceMethod`

Used when the instance is provided at runtime.

Example:

```
List<String> names = List.of("bob", "alice", "carol");
names.sort(String::compareToIgnoreCase);
System.out.println(names); // Output: [alice, bob, carol]
```

Equivalent lambda:

```
names.sort((a, b) -> a.compareToIgnoreCase(b));
```

2.3.5 Constructor Reference

Syntax: `ClassName::new`

Used when you want to instantiate a class using a lambda.

Example:

```
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> myList = listSupplier.get();
myList.add("Item");
System.out.println(myList); // Output: [Item]
```

Equivalent lambda:

```
Supplier<List<String>> listSupplier = () -> new ArrayList<>();
```

By replacing simple lambdas with method references, you can make your code cleaner and easier to understand. As you use more functional constructs in Java, method references will become a natural tool for writing expressive and concise code.

2.4 Example: Simple Calculator Using Lambdas

In this section, we'll build a simple calculator using Java's functional programming features. We'll use **lambda expressions**, **functional interfaces**, and **method references** to model arithmetic operations like addition, subtraction, multiplication, and division. The calculator will allow dynamic selection of operations by passing functions as parameters.

This approach demonstrates how functional programming promotes flexibility and clean separation of logic by treating operations as **first-class functions**.

Here is a complete, runnable Java program:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.function.BiFunction;

public class LambdaCalculator {

    public static void main(String[] args) {
        // Define arithmetic operations using lambdas and method references
        Map<String, BiFunction<Double, Double, Double>> operations = new HashMap<>();

        // Using lambdas
        operations.put("+", (a, b) -> a + b);
        operations.put("-", (a, b) -> a - b);
        // Using method reference for multiplication
        operations.put("*", LambdaCalculator::multiply);
        // Division with lambda and error check
        operations.put("/", (a, b) -> {
            if (b == 0) {
                throw new ArithmeticException("Cannot divide by zero.");
            }
            return a / b;
        });

        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        double x = scanner.nextDouble();

        System.out.print("Enter operator (+, -, *, /): ");
        String op = scanner.next();

        System.out.print("Enter second number: ");
        double y = scanner.nextDouble();
```

```
        BiFunction<Double, Double, Double> operation = operations.get(op);

        if (operation != null) {
            try {
                double result = operation.apply(x, y);
                System.out.println("Result: " + result);
            } catch (ArithmeticException ex) {
                System.out.println("Error: " + ex.getMessage());
            }
        } else {
            System.out.println("Unsupported operation: " + op);
        }

        scanner.close();
    }

    // Method to use as a method reference
    public static double multiply(double a, double b) {
        return a * b;
    }
}
```

2.4.1 Highlights:

- **`BiFunction<Double, Double, Double>`** is used to represent operations that take two numbers and return a result.
- Operations are stored in a `Map`, allowing selection based on user input.
- **Method reference** (`LambdaCalculator::multiply`) is used for multiplication.
- Division includes an inline lambda with error handling for division by zero.
- This design keeps logic modular and easily extendable (e.g., you could add exponentiation with one line).

This example shows how lambdas and functional interfaces can simplify even classic programming tasks like building a calculator.

Chapter 3.

Working with Functional Interfaces

1. Creating Custom Functional Interfaces
2. Default and Static Methods in Interfaces
3. Composing Functions and Predicates
4. Example: Filtering and Transforming Collections

3 Working with Functional Interfaces

3.1 Creating Custom Functional Interfaces

While Java 8 provides many built-in functional interfaces in the `java.util.function` package (like `Function`, `Predicate`, `Consumer`, and `Supplier`), there are times when these don't match your specific use case. In such cases, you can create your own **custom functional interfaces** tailored to your needs.

A **functional interface** is simply an interface with a **single abstract method (SAM)**. These interfaces can be used as targets for lambda expressions and method references, enabling clean and expressive code.

3.1.1 The `@FunctionalInterface` Annotation

Java provides the `@FunctionalInterface` annotation to explicitly mark an interface as functional. This annotation is optional but recommended—it tells the compiler to enforce that the interface contains only one abstract method. If you accidentally add a second abstract method, the compiler will raise an error, helping prevent mistakes.

3.1.2 Defining a Custom Functional Interface

Let's create a functional interface for a custom operation that takes two integers and returns a string:

```
@FunctionalInterface
interface IntToStringOperation {
    String apply(int a, int b);

    // Optional: default method
    default void printExample() {
        System.out.println("This interface converts two ints into a string.");
    }

    // Optional: static method
    static void showInfo() {
        System.out.println("IntToStringOperation is a custom functional interface.");
    }
}
```

This interface has one abstract method, `apply`, and optional `default` and `static` methods for extended functionality.

3.1.3 Example 1: Using the Custom Interface with a Lambda

```
public class CustomFunctionalInterfaceDemo {
    public static void main(String[] args) {
        IntToStringOperation concat = (a, b) -> "Combined: " + a + b;
        System.out.println(concat.apply(10, 20)); // Output: Combined: 1020

        concat.printExample();
        IntToStringOperation.showInfo();
    }
}
```

Full runnable code:

```
@FunctionalInterface
interface IntToStringOperation {
    String apply(int a, int b);

    // Optional: default method
    default void printExample() {
        System.out.println("This interface converts two ints into a string.");
    }

    // Optional: static method
    static void showInfo() {
        System.out.println("IntToStringOperation is a custom functional interface.");
    }
}

public class CustomFunctionalInterfaceDemo {
    public static void main(String[] args) {
        IntToStringOperation concat = (a, b) -> "Combined: " + a + b;
        System.out.println(concat.apply(10, 20)); // Output: Combined: 1020

        concat.printExample(); // Default method
        IntToStringOperation.showInfo(); // Static method
    }
}
```

3.1.4 Example 2: A No-Argument Functional Interface

You can also define interfaces with no parameters:

```
@FunctionalInterface
interface MessageProvider {
    String getMessage();
}

public class MessageExample {
    public static void main(String[] args) {
        MessageProvider provider = () -> "Hello from a custom interface!";
```

```
        System.out.println(provider.getMessage());
    }
}
```

3.1.5 Why Use Custom Functional Interfaces?

- When you need method signatures not covered by built-in interfaces (e.g., more than two parameters or specific return types).
- When naming improves code readability (`MessageProvider` is clearer than `Supplier<String>`).
- When you want to include domain-specific behavior via default or static helper methods.

Creating your own functional interfaces allows for expressive, type-safe, and reusable functional constructs tailored to your application's needs.

3.2 Default and Static Methods in Interfaces

Prior to Java 8, interfaces could only contain abstract methods, meaning every implementing class had to provide the method's behavior. Java 8 introduced **default** and **static methods** in interfaces, bringing more flexibility and power—especially in the context of functional programming.

3.2.1 Default Methods

A **default method** provides a method implementation directly within the interface using the `default` keyword. This allows interfaces to evolve without breaking existing implementations. Default methods are especially useful in **functional interfaces**, where they can offer reusable behaviors or compose functions.

Example 1: Default Method for Composition

```
@FunctionalInterface
interface Formatter {
    String format(String input);

    // Compose uppercase formatting with prefix
    default Formatter withPrefix(String prefix) {
        return (s) -> prefix + format(s);
    }
}
```

```java
public class DefaultMethodExample {
    public static void main(String[] args) {
        Formatter upperCase = s -> s.toUpperCase();
        Formatter withGreeting = upperCase.withPrefix("Hello, ");

        System.out.println(withGreeting.format("world")); // Output: Hello, WORLD
    }
}
```

In this example, `withPrefix` is a default method that returns a new composed `Formatter`.

3.2.2 Static Methods

Static methods in interfaces are utility methods related to the interface's behavior. These can be called without an instance and are often used as **factory** or **helper methods**.

Example 2: Static Factory Method in Functional Interface

```java
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b);

    static MathOperation multiply() {
        return (a, b) -> a * b;
    }
}

public class StaticMethodExample {
    public static void main(String[] args) {
        MathOperation op = MathOperation.multiply();
        System.out.println(op.operate(4, 5)); // Output: 20
    }
}
```

3.2.3 Combining Both

Default and static methods work well together. You can create reusable, composable, and testable patterns using both.

Example 3: Combining Defaults and Statics

```java
@FunctionalInterface
interface Validator {
    boolean isValid(String value);

    default Validator and(Validator other) {
        return s -> this.isValid(s) && other.isValid(s);
    }
```

```
    static Validator notEmpty() {
        return s -> s != null && !s.isEmpty();
    }
}

public class ValidatorExample {
    public static void main(String[] args) {
        Validator validator = Validator.notEmpty().and(s -> s.length() >= 3);
        System.out.println(validator.isValid("abc")); // Output: true
        System.out.println(validator.isValid(""));    // Output: false
    }
}
```

Summary: Default methods support behavior sharing and composition without forcing all implementations to override them, while static methods offer reusable helpers. Together, they enrich functional interfaces with utility, composability, and backwards compatibility.

3.3 Composing Functions and Predicates

One of the key strengths of functional programming is the ability to **compose** small, reusable functions to build more complex behavior. Java's functional interfaces like `Function` and `Predicate` provide built-in methods such as `andThen`, `compose`, `and`, `or`, and `negate` to support this composition. These methods allow chaining and combining logic in a clear and expressive way.

3.3.1 Function Composition

The `Function<T, R>` interface provides two important methods:

- **`andThen(Function after)`**: Executes the current function first, then passes the result to the `after` function.
- **`compose(Function before)`**: Executes the `before` function first, then passes the result to the current function.

This is useful for building data transformation pipelines.

Example 1: Chaining Functions

```
import java.util.function.Function;

public class FunctionCompositionExample {
    public static void main(String[] args) {
        Function<String, String> trim = String::trim;
        Function<String, String> toUpper = String::toUpperCase;
        Function<String, Integer> length = String::length;
```

```
        // Compose: trim -> toUpper -> length
        Function<String, Integer> composed = trim.andThen(toUpper).andThen(length);

        System.out.println(composed.apply("  hello  ")); // Output: 5
    }
}
```

Here, the input string is trimmed, converted to uppercase, and its length is calculated—all using function composition.

3.3.2 Predicate Composition

The `Predicate<T>` interface includes:

- **`and(Predicate other)`**: True if both predicates are true.
- **`or(Predicate other)`**: True if either predicate is true.
- **`negate()`**: Inverts the result of the predicate.

These allow building complex conditions from simpler tests.

Example 2: Combining Predicates

```
import java.util.List;
import java.util.function.Predicate;

public class PredicateCombinationExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "", null, "Bob", "  ", "Charlie");

        Predicate<String> notNull = s -> s != null;
        Predicate<String> notEmpty = s -> !s.isEmpty();
        Predicate<String> notBlank = s -> !s.trim().isEmpty();

        // Combined predicate to filter valid names
        Predicate<String> isValid = notNull.and(notEmpty).and(notBlank);

        names.stream()
             .filter(isValid)
             .forEach(System.out::println); // Output: Alice, Bob, Charlie
    }
}
```

This example filters a list to remove `null`, empty, or blank strings using predicate composition.

3.3.3 Example 3: Function Predicate in a Real Use Case

```java
import java.util.function.Function;
import java.util.function.Predicate;

public class EmailValidation {
    public static void main(String[] args) {
        Function<String, String> normalize = email -> email.toLowerCase().trim();
        Predicate<String> containsAt = email -> email.contains("@");
        Predicate<String> validDomain = email -> email.endsWith(".com");

        String rawEmail = "  USER@Example.COM ";

        String cleaned = normalize.apply(rawEmail);
        boolean isValid = containsAt.and(validDomain).test(cleaned);

        System.out.println("Normalized: " + cleaned);     // Output: user@example.com
        System.out.println("Valid email: " + isValid);    // Output: true
    }
}
```

3.3.4 Why Composition Matters

Function and predicate composition allows you to:

- Write small, focused units of logic.
- Reuse and combine them flexibly.
- Avoid duplication and deeply nested conditionals.

By chaining behavior, your code becomes **declarative**, **expressive**, and **easy to test**—hallmarks of good functional programming in Java.

3.4 Example: Filtering and Transforming Collections

Functional programming in Java simplifies operations on collections by allowing us to declaratively express **filtering**, **mapping**, and **processing** using lambda expressions and functional interfaces. In this example, we'll work with a list of `Person` objects. We'll **filter** out only adults (age 18), and then **transform** each `Person` into a `String` greeting message.

We'll use:

- `Predicate<Person>` to filter.
- `Function<Person, String>` to map.
- Lambdas for concise expression.
- `Stream` API to process the list.

3.4.1 Full Example

```java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class CollectionFilterTransform {
    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("Alice", 22),
            new Person("Bob", 15),
            new Person("Charlie", 30),
            new Person("Daisy", 17)
        );

        // Predicate to filter adults
        Predicate<Person> isAdult = p -> p.getAge() >= 18;

        // Function to convert Person to greeting message
        Function<Person, String> toGreeting = p -> "Hello, " + p.getName() + "!";

        // Process: filter adults, transform to greeting strings
        List<String> greetings = people.stream()
            .filter(isAdult)
            .map(toGreeting)
            .collect(Collectors.toList());

        // Output the result
        greetings.forEach(System.out::println);
    }
}
```

3.4.2 Expected Output

```
Hello, Alice!
Hello, Charlie!
```

3.4.3 Explanation

- The `Predicate<Person>` filters out anyone under 18.
- The `Function<Person, String>` maps a `Person` to a `String` greeting.
- The use of lambda expressions (`p -> p.getAge() >= 18`) makes the logic concise.
- The stream pipeline `filter().map().collect()` processes the list declaratively.

3.4.4 Possible Variations

- Sort the filtered list using `Comparator.comparing(Person::getAge)`.
- Use a `BiFunction` to customize greetings based on age.
- Add a second filter (e.g., names starting with "A") using `.filter(p -> p.getName().startsWith("A"))`.

This example highlights how Java's functional programming model transforms verbose loops into clean, readable, and composable operations using predicates, functions, and the stream API.

Chapter 4.

Using Streams for Data Processing

1. Introduction to Streams API
2. Creating Streams: from Collections, Arrays, and I/O
3. Intermediate Operations: `map`, `filter`, `sorted`, `distinct`
4. Terminal Operations: `forEach`, `collect`, `reduce`
5. Example: Processing Employee Data with Streams

4 Using Streams for Data Processing

4.1 Introduction to Streams API

The **Streams API**, introduced in Java 8, is one of the most powerful additions to the language and a cornerstone of functional programming in Java. It provides a high-level, declarative way to process sequences of data—whether from collections, arrays, or I/O sources—using a fluent and composable pipeline model.

At its core, a **stream** is a sequence of elements that supports various operations to compute results. Unlike collections, which store data in memory, streams are designed for **computation**, not storage. This distinction makes streams ideal for chaining operations like filtering, mapping, and reducing in a functional style.

4.1.1 Streams vs. Collections

Traditional collections like `List` or `Set` are **data structures**, designed to hold and access elements. In contrast, a **stream** is a **view or pipeline** on a data source that can be processed in a functional way.

Key differences:

- **Collections** are eager; streams are **lazy**—operations are not executed until a terminal operation (like `collect()` or `forEach()`) is invoked.
- **Collections** are mutable by default; streams favor **immutability**, avoiding side effects.
- **Collections** require explicit iteration; streams support **declarative** syntax with automatic iteration.

4.1.2 Functional Programming with Streams

Streams support a wide range of **functional operations**, such as:

- **Filtering** elements with `filter(Predicate)`
- **Transforming** elements with `map(Function)`
- **Reducing** elements to a single result with `reduce()`
- **Collecting** results into collections or strings with `collect()`

Each operation in a stream pipeline returns a new stream, enabling **composition** of multiple operations in a readable and fluent way.

Streams also support **lazy evaluation**, meaning intermediate operations are only performed when a terminal operation is triggered. This results in more efficient processing, especially with large datasets or infinite streams.

4.1.3 Imperative vs. Declarative Example

Let's compare traditional iteration with a stream-based approach:

Imperative (loop-based):

```
List<String> names = List.of("Alice", "Bob", "Charlie", "David");
List<String> result = new ArrayList<>();

for (String name : names) {
    if (name.startsWith("A")) {
        result.add(name.toUpperCase());
    }
}
System.out.println(result); // Output: [ALICE]
```

Declarative (stream-based):

```
List<String> result = names.stream()
    .filter(name -> name.startsWith("A"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(result); // Output: [ALICE]
```

The stream version is **shorter, clearer, and more expressive**. It describes *what* to do, not *how* to do it.

4.1.4 Conclusion

The Streams API revolutionizes how Java developers process data. It encourages a functional approach that is clean, composable, and scalable. With support for parallelism, lazy evaluation, and a rich set of operations, streams form the backbone of modern, functional Java programming.

4.2 Creating Streams: from Collections, Arrays, and I/O

Streams in Java can be created from a variety of data sources including **collections**, **arrays**, and **I/O sources**. This flexibility allows developers to apply functional-style operations across different types of data consistently and efficiently.

4.2.1 Streams from Collections

Most commonly, streams are created from Java collections like `List`, `Set`, or `Map`.

- **Sequential Stream**: Processes elements one at a time in order.
- **Parallel Stream**: Processes elements concurrently using multiple threads (internally via the ForkJoinPool).

Example – Sequential Stream:

```
List<String> names = List.of("Alice", "Bob", "Charlie");
names.stream()
     .filter(n -> n.length() > 3)
     .forEach(System.out::println);
```

Example – Parallel Stream:

```
names.parallelStream()
     .map(String::toUpperCase)
     .forEach(System.out::println); // May print in non-deterministic order
```

Use **parallel streams** when operations are independent, CPU-intensive, and performance gain outweighs the overhead of multithreading. Be cautious with side effects and shared mutable state.

4.2.2 Streams from Arrays

Java provides utility methods to create streams from arrays via the `Arrays` class.

Example – Stream from Array:

```
int[] numbers = {1, 2, 3, 4, 5};
int sum = Arrays.stream(numbers)
                .filter(n -> n % 2 == 0)
                .sum(); // Output: 6
System.out.println("Sum of evens: " + sum);
```

For object arrays, you can use:

```
String[] fruits = {"apple", "banana", "cherry"};
Stream<String> fruitStream = Arrays.stream(fruits);
fruitStream.forEach(System.out::println);
```

4.2.3 Streams from I/O Sources

The `java.nio.file.Files` class provides powerful stream-based methods to read files line by line.

Example – Stream from File:

```
import java.nio.file.*;
import java.io.IOException;

public class FileStreamExample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("data.txt");
        Files.lines(path)
             .filter(line -> !line.isBlank())
             .map(String::trim)
             .forEach(System.out::println);
    }
}
```

This approach reads a file lazily and efficiently—ideal for processing large files without loading the entire content into memory.

4.2.4 Summary

Source Type	Method
Collections	`stream()`, `parallelStream()`
Arrays	`Arrays.stream(array)`
Files (text)	`Files.lines(Path)`

By understanding how to create streams from these sources, you can effectively harness the power of Java's functional programming features across various data contexts. This sets the foundation for writing clean, declarative, and efficient data processing pipelines.

4.3 Intermediate Operations: `map`, `filter`, `sorted`, `distinct`

In the Java Streams API, **intermediate operations** are used to build up a pipeline of transformations. These operations do not produce a final result immediately; instead, they return a new stream, allowing you to chain multiple operations together. Actual processing occurs **only when a terminal operation** (like `collect()` or `forEach()`) is invoked, making intermediate operations **lazy**.

This laziness enables efficient data processing, as elements are only evaluated as needed, and

in some cases, not at all (e.g., when using `limit()` or `findFirst()`).

Let's explore four essential intermediate operations: `map`, `filter`, `sorted`, and `distinct`.

4.3.1 `map()` Transforming Elements

The `map(Function<T, R>)` method transforms each element in a stream from type `T` to type `R`. It is ideal for converting data.

Example: Convert a list of names to uppercase

```
List<String> names = List.of("alice", "bob", "charlie");
List<String> upper = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(upper); // Output: [ALICE, BOB, CHARLIE]
```

4.3.2 `filter()` Selecting Elements by Condition

The `filter(Predicate<T>)` method includes only those elements that match a given condition.

Example: Keep only even numbers

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
List<Integer> evens = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());

System.out.println(evens); // Output: [2, 4, 6]
```

4.3.3 `sorted()` Ordering Elements

The `sorted()` method sorts the stream's elements. It uses natural ordering by default, but you can supply a custom `Comparator`.

Example: Sort strings by length

```
List<String> words = List.of("banana", "apple", "kiwi");
List<String> sorted = words.stream()
    .sorted(Comparator.comparingInt(String::length))
    .collect(Collectors.toList());
```

```
System.out.println(sorted); // Output: [kiwi, apple, banana]
```

4.3.4 `distinct()` Removing Duplicates

The `distinct()` method removes duplicate elements based on `equals()`.

Example: Remove duplicate integers

```
List<Integer> nums = List.of(1, 2, 2, 3, 3, 3, 4);
List<Integer> unique = nums.stream()
    .distinct()
    .collect(Collectors.toList());

System.out.println(unique); // Output: [1, 2, 3, 4]
```

4.3.5 Chaining Intermediate Operations

Intermediate operations can be chained fluently to create powerful and readable data pipelines.

Example: Transform and filter names

```
List<String> names = List.of("Alice", "Bob", "Alex", "Charlie");
List<String> result = names.stream()
    .filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());

System.out.println(result); // Output: [ALEX, ALICE]
```

Full runnable code:

```
import java.util.*;
import java.util.stream.*;

public class StreamOperationsDemo {

    public static void main(String[] args) {
        // map(): Convert names to uppercase
        List<String> names = List.of("alice", "bob", "charlie");
        List<String> upper = names.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());
        System.out.println("Uppercase names: " + upper); // [ALICE, BOB, CHARLIE]

        // filter(): Keep only even numbers
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

```
        List<Integer> evens = numbers.stream()
            .filter(n -> n % 2 == 0)
            .collect(Collectors.toList());
        System.out.println("Even numbers: " + evens); // [2, 4, 6]

        // sorted(): Sort strings by length
        List<String> words = List.of("banana", "apple", "kiwi");
        List<String> sorted = words.stream()
            .sorted(Comparator.comparingInt(String::length))
            .collect(Collectors.toList());
        System.out.println("Sorted by length: " + sorted); // [kiwi, apple, banana]

        // distinct(): Remove duplicate integers
        List<Integer> nums = List.of(1, 2, 2, 3, 3, 3, 4);
        List<Integer> unique = nums.stream()
            .distinct()
            .collect(Collectors.toList());
        System.out.println("Unique values: " + unique); // [1, 2, 3, 4]

        // Chaining: Filter names that start with "A", uppercase, sort
        List<String> mixedNames = List.of("Alice", "Bob", "Alex", "Charlie");
        List<String> result = mixedNames.stream()
            .filter(n -> n.startsWith("A"))
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.toList());
        System.out.println("Filtered and transformed: " + result); // [ALEX, ALICE]
    }
}
```

4.3.6 Conclusion

Intermediate operations are **composable**, **lazy**, and **pure**—they transform or filter data without modifying the underlying source. Their lazy nature allows the Java runtime to optimize stream pipelines and avoid unnecessary computations. By combining `map`, `filter`, `sorted`, and `distinct`, you can write expressive, functional-style code to process data cleanly and efficiently.

4.4 Terminal Operations: `forEach`, `collect`, `reduce`

In Java's Streams API, **terminal operations** are the final step in a stream pipeline. They **trigger execution** of all intermediate operations and produce a result (e.g., a collection, a single value) or cause a side effect (e.g., printing values). After a terminal operation is invoked, the stream can no longer be used.

In this section, we explore three of the most commonly used terminal operations: `forEach`, `collect`, and `reduce`.

4.4.1 `forEach()` Iteration and Side Effects

The `forEach(Consumer<? super T> action)` method performs an action for each element in the stream. It is often used for logging, printing, or performing side effects.

Example: Printing items

```
List<String> names = List.of("Alice", "Bob", "Charlie");

names.stream()
     .filter(name -> name.length() > 3)
     .forEach(System.out::println); // Output: Alice, Charlie
```

> **Note:** `forEach` should be avoided for modifying external state, especially in parallel streams, as it may lead to unpredictable results.

4.4.2 `collect()` Gathering Results

The `collect()` method performs a **mutable reduction** of elements into a collection, string, map, or other structure. It uses the **Collectors** utility class to define the type of accumulation.

Common Collectors:

- **`Collectors.toList()`** — gathers elements into a `List`
- **`Collectors.toSet()`** — gathers elements into a `Set`
- **`Collectors.joining()`** — concatenates strings
- **`Collectors.groupingBy()`** — groups elements by a classifier function

Example: Collect to list

```
List<String> upper = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(upper); // Output: [ALICE, BOB, CHARLIE]
```

Example: Grouping by string length

```
Map<Integer, List<String>> grouped = names.stream()
    .collect(Collectors.groupingBy(String::length));

System.out.println(grouped);
// Output: {3=[Bob], 5=[Alice], 7=[Charlie]}
```

Example: Join names into a string

```
String joined = names.stream()
    .collect(Collectors.joining(", "));
```

```
System.out.println(joined); // Output: Alice, Bob, Charlie
```

The `collect()` method is powerful and flexible for producing results in many shapes.

4.4.3 `reduce()` Aggregating Values

The `reduce()` method combines stream elements into a **single result** using an **accumulator** function.

There are three common forms:

- `reduce(identity, accumulator)`
- `reduce(accumulator)` – returns `Optional<T>`
- `reduce(identity, accumulator, combiner)` – used for parallel reduction

Example: Sum of numbers

```
List<Integer> nums = List.of(1, 2, 3, 4, 5);
int sum = nums.stream()
    .reduce(0, Integer::sum); // identity: 0, accumulator: sum

System.out.println("Sum: " + sum); // Output: Sum: 15
```

Example: Find the longest name

```
Optional<String> longest = names.stream()
    .reduce((a, b) -> a.length() > b.length() ? a : b);

longest.ifPresent(System.out::println); // Output: Charlie
```

Full runnable code:

```
import java.util.*;
import java.util.stream.*;

public class StreamTerminalOperationsDemo {

    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");

        // forEach(): print names longer than 3 characters
        System.out.println("Names with length > 3:");
        names.stream()
             .filter(name -> name.length() > 3)
             .forEach(System.out::println); // Alice, Charlie

        // collect(): map to uppercase and collect to list
        List<String> upper = names.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());
```

```
        System.out.println("\nUppercase list: " + upper); // [ALICE, BOB, CHARLIE]

        // collect(): group by string length
        Map<Integer, List<String>> grouped = names.stream()
            .collect(Collectors.groupingBy(String::length));
        System.out.println("\nGrouped by length: " + grouped);
        // Example output: {3=[Bob], 5=[Alice], 7=[Charlie]}

        // collect(): join names into a string
        String joined = names.stream()
            .collect(Collectors.joining(", "));
        System.out.println("\nJoined names: " + joined); // Alice, Bob, Charlie

        // reduce(): sum of numbers
        List<Integer> nums = List.of(1, 2, 3, 4, 5);
        int sum = nums.stream()
            .reduce(0, Integer::sum);
        System.out.println("\nSum of numbers: " + sum); // 15

        // reduce(): find the longest name
        Optional<String> longest = names.stream()
            .reduce((a, b) -> a.length() > b.length() ? a : b);
        longest.ifPresent(name -> System.out.println("\nLongest name: " + name)); // Charlie
    }
}
```

4.4.4 Summary

Operation	Purpose
`forEach`	Perform side effects on each element
`collect`	Accumulate elements into containers
`reduce`	Aggregate elements into a single value

Terminal operations finalize stream processing and allow you to extract meaningful results from transformed data. By using `forEach`, `collect`, and `reduce`, you can build expressive and efficient pipelines that transform data from raw sequences into structured outcomes.

4.5 Example: Processing Employee Data with Streams

This example demonstrates how to use Java Streams to process a list of employee objects. We will filter employees by department and salary, map their data to extract names and emails, sort them by salary, and collect the results into new lists.

4.5.1 Employee Class and Sample Data

```
import java.util.*;
import java.util.stream.*;

class Employee {
    private String name;
    private String email;
    private String department;
    private double salary;

    public Employee(String name, String email, String department, double salary) {
        this.name = name;
        this.email = email;
        this.department = department;
        this.salary = salary;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public String getDepartment() { return department; }
    public double getSalary() { return salary; }

    @Override
    public String toString() {
        return name + " (" + department + "), $" + salary;
    }
}

public class EmployeeStreamExample {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
            new Employee("Alice", "alice@example.com", "Engineering", 95000),
            new Employee("Bob", "bob@example.com", "Sales", 55000),
            new Employee("Charlie", "charlie@example.com", "Engineering", 105000),
            new Employee("Diana", "diana@example.com", "Marketing", 60000),
            new Employee("Evan", "evan@example.com", "Engineering", 75000)
        );

        // Filter Engineering employees with salary > 80000
        List<Employee> highEarners = employees.stream()
            .filter(e -> e.getDepartment().equals("Engineering"))
            .filter(e -> e.getSalary() > 80000)
            .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
            .collect(Collectors.toList());

        System.out.println("High earning Engineers:");
        highEarners.forEach(System.out::println);

        // Extract email addresses of Marketing employees
        List<String> marketingEmails = employees.stream()
            .filter(e -> e.getDepartment().equals("Marketing"))
            .map(Employee::getEmail)
            .collect(Collectors.toList());

        System.out.println("\nMarketing team emails:");
        marketingEmails.forEach(System.out::println);
```

```
    }
}
```

4.5.2 Explanation

- **Filtering:** We use `.filter()` twice to narrow down employees to those in "Engineering" earning more than $80,000.
- **Sorting:** `.sorted()` orders the filtered employees by salary in descending order using a method reference to `Employee::getSalary`.
- **Mapping:** `.map(Employee::getEmail)` extracts email addresses of marketing employees.
- **Collecting:** `.collect(Collectors.toList())` gathers the processed elements into lists.
- **Method references and lambdas:** The example combines concise lambda expressions (e.g., for filtering) and method references (e.g., for mapping and sorting) for clarity.

4.5.3 Expected Output

```
High earning Engineers:
Charlie (Engineering), $105000.0
Alice (Engineering), $95000.0

Marketing team emails:
diana@example.com
```

This example highlights the expressiveness of stream pipelines for data processing, showcasing filtering, transformation, sorting, and collecting with clear and readable code.

Chapter 5.

Advanced Stream Operations

1. Parallel Streams and Performance
2. Short-circuiting Operations: `limit`, `findFirst`, `anyMatch`
3. FlatMap for Nested Data
4. Example: Processing Nested Collections

5 Advanced Stream Operations

5.1 Parallel Streams and Performance

Java's Streams API offers a powerful feature called **parallel streams**, which enables processing data concurrently using multiple CPU cores. This is achieved by splitting the stream's data into multiple chunks, processing them in parallel threads, and then combining the results. Parallel streams can dramatically speed up data-intensive operations on large datasets, making them a valuable tool for performance optimization.

5.1.1 How Parallel Streams Work

When you create a parallel stream (via `.parallelStream()` or `.stream().parallel()`), the framework uses the **ForkJoinPool** to distribute tasks across available CPU cores. Each thread processes a portion of the data independently, and the results are merged at the end.

This approach contrasts with **sequential streams**, where processing happens on a single thread, executing operations one element at a time in order.

5.1.2 Potential Benefits

- **Improved performance**: For CPU-bound, large datasets, parallel streams can reduce processing time by leveraging multiple cores.
- **Simplified concurrency**: Developers can write functional, declarative code without explicitly managing threads, locks, or synchronization.

5.1.3 Common Pitfalls and Considerations

- **Thread-safety**: Operations must be **stateless and side-effect free** to avoid data races or inconsistent results. Avoid modifying shared mutable data during stream processing.
- **Order preservation**: Some stream operations (like `forEach`) do not guarantee order in parallel streams, which might be problematic if the order matters.
- **Overhead**: For small datasets, the cost of managing multiple threads and merging results may exceed benefits, making parallel streams slower than sequential ones.
- **Blocking operations**: Parallel streams are less effective when operations block or wait (e.g., I/O), as this can stall threads and reduce concurrency benefits.

5.1.4 When to Use Parallel Streams

- Use parallel streams when:
  - You are processing large collections or datasets.
  - Operations are CPU-intensive and independent.
  - Results do not depend on element order, or order can be controlled.
- Avoid parallel streams when:
  - The dataset is small.
  - Operations have side effects or require synchronization.
  - You rely heavily on order-sensitive results without proper care.

5.1.5 Measuring Performance: Sequential vs. Parallel

Here's a simple benchmark comparing sequential and parallel streams for summing a large range of numbers:

```
import java.util.stream.IntStream;

public class ParallelStreamBenchmark {
    public static void main(String[] args) {
        int max = 10_000_000;

        // Sequential sum
        long start = System.currentTimeMillis();
        long seqSum = IntStream.rangeClosed(1, max)
                               .sum();
        long end = System.currentTimeMillis();
        System.out.println("Sequential sum: " + seqSum + " in " + (end - start) + " ms");

        // Parallel sum
        start = System.currentTimeMillis();
        long parSum = IntStream.rangeClosed(1, max)
                               .parallel()
                               .sum();
        end = System.currentTimeMillis();
        System.out.println("Parallel sum: " + parSum + " in " + (end - start) + " ms");
    }
}
```

Typical output:

```
Sequential sum: 50000005000000 in 150 ms
Parallel sum: 50000005000000 in 50 ms
```

The parallel version often executes faster on multi-core machines, but results may vary depending on CPU, JVM optimizations, and system load.

5.1.6 Conclusion

Parallel streams are a convenient way to utilize multiple CPU cores and improve performance for large-scale data processing. However, understanding when to use them and avoiding common pitfalls like side effects and order-dependence is crucial. Always **measure and profile** your application to ensure parallel streams deliver the desired performance benefits.

5.2 Short-circuiting Operations: `limit`, `findFirst`, `anyMatch`

Short-circuiting operations in Java Streams are powerful tools that can **terminate the stream pipeline early** once a certain condition is met, improving efficiency by avoiding unnecessary processing. These operations help save time and resources, especially when dealing with large or potentially infinite data sources.

5.2.1 `limit(long maxSize)`

The `limit` operation **restricts the stream to process only the first `maxSize` elements**, ignoring the rest. This is especially useful for implementing **pagination** or sampling.

Example: Pagination with `limit`

```
List<String> names = List.of("Alice", "Bob", "Charlie", "Diana", "Evan");

List<String> firstTwo = names.stream()
                             .limit(2)
                             .collect(Collectors.toList());

System.out.println(firstTwo); // Output: [Alice, Bob]
```

5.2.2 `findFirst()`

The `findFirst` operation **retrieves the first element** in the stream that matches the criteria (if any). It returns an `Optional<T>`, so it handles the case where no element matches safely.

Example: Find the first long name

```
List<String> names = List.of("Bob", "Alice", "Charlie", "Diana");

Optional<String> firstLongName = names.stream()
                                      .filter(name -> name.length() > 5)
```

```
                                        .findFirst();

firstLongName.ifPresent(System.out::println); // Output: Charlie
```

`findFirst` is useful for early termination in searches where only one match is needed.

5.2.3 `anyMatch(PredicateT)`

The `anyMatch` operation checks **whether any element in the stream matches the given predicate**. It returns `true` immediately when a match is found, or `false` if none matches.

Example: Quick check for a condition

```
List<Integer> numbers = List.of(1, 3, 5, 8, 9);

boolean hasEven = numbers.stream()
                         .anyMatch(n -> n % 2 == 0);

System.out.println(hasEven); // Output: true (because of 8)
```

`anyMatch` is ideal for **quickly verifying the existence of elements** that meet a condition, which can significantly reduce processing time on large datasets.

5.2.4 Summary

Operation	Purpose	Result Type	Use Case Example
`limit(n)`	Process only first `n` elements	Stream	Pagination or sampling
`findFirst`	Get first matching element	`Optional<T>`	Early search termination
`anyMatch`	Check if any element matches	`boolean`	Quick condition checks

5.2.5 Why Use Short-circuiting?

Short-circuiting saves computation by **stopping the pipeline as soon as the result is known**, which is especially beneficial for large or infinite streams.

These operations demonstrate the flexibility and efficiency of the Streams API, allowing you to build performant data-processing pipelines that terminate early when possible.

5.3 FlatMap for Nested Data

When working with nested data structures—such as lists of lists or collections inside objects—the `flatMap` operation becomes essential. It **flattens** multiple levels of nested streams into a single continuous stream, simplifying processing.

5.3.1 Difference Between `map` and `flatMap`

- `map` transforms each element into another element (or stream) but **preserves the nesting**.
- `flatMap` transforms each element into a stream and then **flattens those streams into one stream**.

In short: `map` produces a stream of streams when the mapping function returns a stream, while `flatMap` merges those streams into a single stream.

5.3.2 Example 1: Flattening a List of Lists

Suppose you have a list of lists of integers:

```
List<List<Integer>> listOfLists = List.of(
    List.of(1, 2, 3),
    List.of(4, 5),
    List.of(6, 7, 8, 9)
);
```

Using `map` would produce a stream of streams:

```
Stream<Stream<Integer>> mapped = listOfLists.stream()
    .map(list -> list.stream());

mapped.forEach(s -> s.forEach(System.out::println));
// Prints all numbers but requires nested loops
```

Using `flatMap` flattens this into a single stream of integers:

```
List<Integer> flattened = listOfLists.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());

System.out.println(flattened);
// Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.3.3 Example 2: Extracting Nested Fields from Objects

Imagine you have a `Person` class where each person has a list of phone numbers:

```
class Person {
    private String name;
    private List<String> phoneNumbers;

    // Constructor, getters omitted for brevity
}

List<Person> people = List.of(
    new Person("Alice", List.of("123-456", "234-567")),
    new Person("Bob", List.of("345-678")),
    new Person("Charlie", List.of("456-789", "567-890"))
);
```

To get a flat list of all phone numbers:

```
List<String> allNumbers = people.stream()
    .flatMap(person -> person.getPhoneNumbers().stream())
    .collect(Collectors.toList());

System.out.println(allNumbers);
// Output: [123-456, 234-567, 345-678, 456-789, 567-890]
```

Full runnable code:

```
import java.util.*;
import java.util.stream.*;

class Person {
    private String name;
    private List<String> phoneNumbers;

    public Person(String name, List<String> phoneNumbers) {
        this.name = name;
        this.phoneNumbers = phoneNumbers;
    }

    public String getName() {
        return name;
    }

    public List<String> getPhoneNumbers() {
        return phoneNumbers;
    }
}

public class MapFlatMapExample {
    public static void main(String[] args) {
        // Example 1: Flattening a List of Lists
        List<List<Integer>> listOfLists = List.of(
            List.of(1, 2, 3),
            List.of(4, 5),
            List.of(6, 7, 8, 9)
```

```
        );

        System.out.println("Using map (Stream<Stream<Integer>>):");
        Stream<Stream<Integer>> mapped = listOfLists.stream()
            .map(list -> list.stream());
        mapped.forEach(stream -> stream.forEach(System.out::println)); // nested iteration

        System.out.println("\nUsing flatMap (Stream<Integer>):");
        List<Integer> flattened = listOfLists.stream()
            .flatMap(List::stream)
            .collect(Collectors.toList());
        System.out.println("Flattened list: " + flattened); // [1, 2, 3, 4, 5, 6, 7, 8, 9]

        // Example 2: Extracting Nested Fields from Objects
        List<Person> people = List.of(
            new Person("Alice", List.of("123-456", "234-567")),
            new Person("Bob", List.of("345-678")),
            new Person("Charlie", List.of("456-789", "567-890"))
        );

        List<String> allNumbers = people.stream()
            .flatMap(person -> person.getPhoneNumbers().stream())
            .collect(Collectors.toList());

        System.out.println("\nAll phone numbers: " + allNumbers);
        // Output: [123-456, 234-567, 345-678, 456-789, 567-890]
    }
}
```

5.3.4 When to Use `flatMap`

- When you have **nested collections** (e.g., list of lists) and want to process all elements as a single sequence.
- When extracting **nested fields or relationships** from objects.
- When parsing **complex data structures**, like JSON arrays inside arrays.

5.3.5 Summary

Operation	Result
`map`	Stream of streams (nested)
`flatMap`	Flattened, single-level stream

By mastering `flatMap`, you can handle deeply nested or complex data structures cleanly and efficiently, unlocking more powerful data processing patterns in Java's functional programming paradigm.

5.4 Example: Processing Nested Collections

When working with nested collections—such as a list of departments each containing a list of employees—handling the data can become verbose and cumbersome using traditional loops. The `flatMap` operation in Java Streams simplifies this by flattening nested streams into a single stream, allowing seamless processing of deeply nested data.

Let's consider a practical example: we have a `List<Department>`, where each `Department` holds a list of `Employee` objects. Our goal is to find all employees with a salary greater than $75,000 across all departments.

Here is how we might model the classes:

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name + " ($" + salary + ")";
    }
}

class Department {
    String name;
    List<Employee> employees;

    Department(String name, List<Employee> employees) {
        this.name = name;
        this.employees = employees;
    }
}
```

Traditional nested loops approach:

```
List<Employee> highEarners = new ArrayList<>();
for (Department dept : departments) {
    for (Employee emp : dept.employees) {
        if (emp.salary > 75000) {
            highEarners.add(emp);
        }
    }
}
```

While this works, it quickly becomes bulky as complexity grows.

Using Streams and `flatMap`:

```
List<Employee> highEarners = departments.stream()
    // Flatten the stream of departments into a stream of employees
    .flatMap(dept -> dept.employees.stream())
    // Filter employees with salary > 75,000
    .filter(emp -> emp.salary > 75000)
    // Collect the results into a list
    .collect(Collectors.toList());
```

This approach is concise, readable, and expressive. `flatMap` replaces the nested iteration by producing one continuous stream of employees from all departments, so you can apply filters and other operations directly.

5.4.1 Sample Data and Full Example:

```
public class NestedCollectionsExample {
    public static void main(String[] args) {
        List<Department> departments = Arrays.asList(
            new Department("Engineering", Arrays.asList(
                new Employee("Alice", 90000),
                new Employee("Bob", 60000),
                new Employee("Charlie", 80000)
            )),
            new Department("HR", Arrays.asList(
                new Employee("Diana", 70000),
                new Employee("Evan", 85000)
            )),
            new Department("Sales", Arrays.asList(
                new Employee("Fiona", 72000),
                new Employee("George", 78000)
            ))
        );

        List<Employee> highEarners = departments.stream()
            .flatMap(dept -> dept.employees.stream())
            .filter(emp -> emp.salary > 75000)
            .collect(Collectors.toList());

        System.out.println("Employees with salary > $75,000:");
        highEarners.forEach(System.out::println);
    }
}
```

Full runnable code:

```
import java.util.*;
import java.util.stream.Collectors;

class Employee {
```

```
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name + " ($" + salary + ")";
    }
}

class Department {
    String name;
    List<Employee> employees;

    Department(String name, List<Employee> employees) {
        this.name = name;
        this.employees = employees;
    }
}

public class NestedCollectionsExample {
    public static void main(String[] args) {
        List<Department> departments = Arrays.asList(
            new Department("Engineering", Arrays.asList(
                new Employee("Alice", 90000),
                new Employee("Bob", 60000),
                new Employee("Charlie", 80000)
            )),
            new Department("HR", Arrays.asList(
                new Employee("Diana", 70000),
                new Employee("Evan", 85000)
            )),
            new Department("Sales", Arrays.asList(
                new Employee("Fiona", 72000),
                new Employee("George", 78000)
            ))
        );

        List<Employee> highEarners = departments.stream()
            .flatMap(dept -> dept.employees.stream())
            .filter(emp -> emp.salary > 75000)
            .collect(Collectors.toList());

        System.out.println("Employees with salary > $75,000:");
        highEarners.forEach(System.out::println);
    }
}
```

5.4.2 Expected Output:

```
Employees with salary > $75,000:
Alice ($90000.0)
Charlie ($80000.0)
Evan ($85000.0)
George ($78000.0)
```

5.4.3 Summary

- The nested `List<Department>` to `List<Employee>` transformation is streamlined by `flatMap`.
- Without `flatMap`, nested loops are needed to iterate through departments and employees.
- `flatMap` "flattens" the nested lists into a single stream, enabling operations like `filter` to work across all employees.
- This leads to cleaner, more maintainable, and declarative code when dealing with nested data structures.

Chapter 6.

Optional and Handling Nulls Functionally

1. Using `Optional` to Avoid NullPointerExceptions
2. Common `Optional` Operations
3. Example: Safely Navigating Complex Object Graphs

6 Optional and Handling Nulls Functionally

6.1 Using `Optional` to Avoid NullPointerExceptions

One of the most common and frustrating problems in Java development has been dealing with `null` values, which can lead to unexpected and hard-to-debug `NullPointerException`s (NPEs). Traditionally, Java programmers had to scatter explicit null checks throughout their code to avoid these exceptions, making the code verbose, error-prone, and harder to maintain.

To address this, Java 8 introduced the `Optional<T>` class—a container object which may or may not contain a non-null value. The idea behind `Optional` is to **make the possibility of absence explicit** in the type system, encouraging developers to handle the "no value" case in a deliberate and clear way, instead of ignoring it or risking a null dereference.

Why was `Optional` introduced?

Before `Optional`, a method returning a reference type often returned `null` to indicate "no result". But callers had no compile-time guarantee that a value might be absent—they had to remember to check for `null`. Forgetting to do so resulted in runtime NPEs, which are common bugs in Java applications.

`Optional` makes it explicit that the value might be missing. This shifts the responsibility from silently dealing with `null` to explicitly handling presence or absence, improving code readability and safety.

Creating `Optional` instances

- **Empty `Optional`**: Represents absence of a value.
  ```
  Optional<String> emptyOpt = Optional.empty();
  ```
- **Non-null value**: Wrap a guaranteed non-null value.
  ```
  Optional<String> name = Optional.of("Alice");
  ```
- **Nullable value**: Wrap a value that might be `null`. If it is, returns an empty `Optional`.
  ```
  Optional<String> nullableName = Optional.ofNullable(possibleNullName);
  ```

Checking for presence and accessing the value

You can check if a value is present with `isPresent()`:

```
if (name.isPresent()) {
    System.out.println("Name is " + name.get());
}
```

But a more idiomatic and safer approach uses methods like `ifPresent()`, `orElse()`, or `orElseGet()` to handle default values or conditional execution without calling `get()` directly.

Contrasting Traditional Null Handling vs. `Optional`

Traditional null handling:

```
String getEmployeeName(Employee emp) {
    if (emp != null && emp.getName() != null) {
        return emp.getName();
    } else {
        return "Unknown";
    }
}
```

This code is cluttered with null checks and is error-prone if you forget any.

Using `Optional`:

```
Optional<String> getEmployeeNameOptional(Employee emp) {
    return Optional.ofNullable(emp)
                   .map(Employee::getName);
}

// Usage
String name = getEmployeeNameOptional(emp).orElse("Unknown");
```

Here, `Optional` clearly communicates that the name might be absent, and the caller can specify a default or alternative action without explicit null checks.

6.1.1 Philosophy behind `Optional`

The design goal of `Optional` is to **prevent the null-related errors by encouraging explicit, functional-style handling of absent values**. Rather than "defensive programming" with scattered null checks, you adopt a declarative style that clarifies intent, making the code safer and easier to follow.

`Optional` is not intended to replace every nullable reference, especially not in fields or collections, but rather to be used primarily as method return types to signal optional values clearly.

In summary, `Optional` is a powerful tool that elevates the handling of potentially missing values from an implicit, error-prone practice into an explicit, expressive part of your Java programs, helping you avoid `NullPointerException`s and write cleaner, more robust code.

6.2 Common `Optional` Operations

The `Optional` API provides a rich set of methods that allow you to safely and expressively work with values that may or may not be present. These methods help you avoid explicit

null checks and write fluent, readable code. Below, we explore some of the most commonly used operations on `Optional` and show how they can be combined to handle optional values effectively.

`isPresent()` and `ifPresent()`

- **`isPresent()`**: Returns `true` if a value is present, otherwise `false`.
- **`ifPresent(Consumer<? super T> action)`**: Executes the given action if a value is present.

```
Optional<String> optName = Optional.of("Alice");

if (optName.isPresent()) {
    System.out.println("Name is: " + optName.get());
}

// Preferred:
optName.ifPresent(name -> System.out.println("Name is: " + name));
```

`ifPresent` is often preferred over `isPresent` + `get()` because it avoids explicit calls to `get()` and improves safety.

Providing Default Values: `orElse()`, `orElseGet()`, and `orElseThrow()`

- **`orElse(T other)`**: Returns the value if present; otherwise returns the provided default.
- **`orElseGet(Supplier<? extends T> other)`**: Similar to `orElse`, but the default is computed lazily using a supplier.
- **`orElseThrow()`**: Returns the value if present, otherwise throws `NoSuchElementException`.
- **`orElseThrow(Supplier<? extends X> exceptionSupplier)`**: Throws a custom exception if no value is present.

```
Optional<String> emptyOpt = Optional.empty();

// orElse returns default eagerly
String name1 = emptyOpt.orElse(getDefaultName()); // getDefaultName() is called regardless

// orElseGet calls supplier lazily
String name2 = emptyOpt.orElseGet(() -> getDefaultName()); // called only if needed

// orElseThrow throws exception if absent
String name3 = optName.orElseThrow(() -> new IllegalStateException("Name missing"));
```

Best practice: Use `orElseGet` when generating the default is expensive or has side effects, to avoid unnecessary computation.

Transforming Values: `map()` and `flatMap()`

- **`map(Function<? super T, ? extends U>)`**: Applies a function to the value if present, wrapping the result back in an `Optional`.
- **`flatMap(Function<? super T, Optional<U>>)`**: Similar to `map`, but the function returns an `Optional` itself, so it avoids nested `Optional<Optional<U>>`.

```java
Optional<String> nameOpt = Optional.of("Alice");

// Convert to uppercase if present
Optional<String> upperName = nameOpt.map(String::toUpperCase);

// Chaining example with flatMap
Optional<String> trimmedName = nameOpt
    .map(String::trim)
    .flatMap(s -> s.isEmpty() ? Optional.empty() : Optional.of(s));

System.out.println(upperName.orElse("No name"));    // Outputs: ALICE
System.out.println(trimmedName.orElse("Empty name")); // Outputs: Alice
```

`map()` is ideal for simple transformations. Use `flatMap()` when the mapping function itself returns an `Optional`, avoiding nested optionals.

Chaining Operations for Fluent and Safe Processing

The true power of `Optional` is visible when chaining multiple operations:

```java
Optional<String> result = Optional.ofNullable(getUser())
    .map(User::getAddress)
    .map(Address::getCity)
    .filter(city -> !city.isEmpty())
    .map(String::toUpperCase);

result.ifPresent(city -> System.out.println("City: " + city));
```

This sequence safely navigates a potentially null-laden object graph without explicit null checks.

6.2.1 Common Pitfalls

- **Avoid calling `get()` without checking presence:** This defeats the purpose of `Optional` and leads to exceptions.
- **Don't use `Optional` for fields or collections:** It's designed primarily for method return types.
- **Beware eager evaluation in `orElse()`:** Use `orElseGet()` if the default value is expensive to compute.
- **Don't misuse `flatMap` when `map` suffices:** `flatMap` expects a function returning `Optional`; otherwise, use `map`.

6.2.2 Summary Table of Key Methods

Method	Description	Returns
`isPresent()`	Checks presence	`boolean`
`ifPresent()`	Performs action if value present	`void`
`orElse()`	Returns value or default eagerly	`T`
`orElseGet()`	Returns value or default lazily	`T`
`orElseThrow()`	Returns value or throws exception	`T`
`map()`	Transforms value if present	`Optional<U>`
`flatMap()`	Transforms value producing another Optional	`Optional<U>`

By leveraging these `Optional` operations, you can handle potentially absent values in a fluent, declarative style—reducing bugs and improving code readability while avoiding cumbersome null checks.

6.3 Example: Safely Navigating Complex Object Graphs

In real-world applications, you often encounter deeply nested object graphs where any intermediate node can be `null`. For example, a `User` may have an `Address`, which may have a `City`, which may have a `ZipCode`. Navigating this hierarchy with traditional null checks quickly becomes unwieldy and error-prone.

Using `Optional`, you can safely traverse this graph without cluttered nested `if` statements, handling missing data gracefully and providing sensible defaults when necessary.

Modeling the classes:

```
class ZipCode {
    private String code;

    ZipCode(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }
}

class City {
    private ZipCode zipCode;

    City(ZipCode zipCode) {
        this.zipCode = zipCode;
    }

    public ZipCode getZipCode() {
        return zipCode;
```

```
    }
}

class Address {
    private City city;

    Address(City city) {
        this.city = city;
    }

    public City getCity() {
        return city;
    }
}

class User {
    private Address address;

    User(Address address) {
        this.address = address;
    }

    public Address getAddress() {
        return address;
    }
}
```

Traditional null checks to get zipcode:

```
String getZipCodeTraditional(User user) {
    if (user != null) {
        Address address = user.getAddress();
        if (address != null) {
            City city = address.getCity();
            if (city != null) {
                ZipCode zip = city.getZipCode();
                if (zip != null) {
                    return zip.getCode();
                }
            }
        }
    }
    return "00000";  // Default zipcode
}
```

This nested structure is verbose and hard to maintain.

Using `Optional` to safely navigate and extract the zipcode:

```
import java.util.Optional;

public class OptionalNavigationExample {

    public static void main(String[] args) {
        // Sample users with different levels of missing data
```

```
        User user1 = new User(new Address(new City(new ZipCode("12345"))));
        User user2 = new User(new Address(new City(null))); // ZipCode missing
        User user3 = new User(null);                        // Address missing
        User user4 = null;                                  // User is null

        System.out.println(getZipCode(user1)); // Outputs: 12345
        System.out.println(getZipCode(user2)); // Outputs: 00000 (default)
        System.out.println(getZipCode(user3)); // Outputs: 00000 (default)
        System.out.println(getZipCode(user4)); // Outputs: 00000 (default)
    }

    static String getZipCode(User user) {
        return Optional.ofNullable(user)
                .map(User::getAddress)
                .map(Address::getCity)
                .map(City::getZipCode)
                .map(ZipCode::getCode)
                .orElse("00000");  // Provide default zipcode if any level is missing
    }
}
```

Full runnable code:

```
import java.util.Optional;

class ZipCode {
    private String code;

    ZipCode(String code) {
        this.code = code;
    }

    public String getCode() {
        return code;
    }
}

class City {
    private ZipCode zipCode;

    City(ZipCode zipCode) {
        this.zipCode = zipCode;
    }

    public ZipCode getZipCode() {
        return zipCode;
    }
}

class Address {
    private City city;

    Address(City city) {
        this.city = city;
    }

    public City getCity() {
```

```
        return city;
    }
}

class User {
    private Address address;

    User(Address address) {
        this.address = address;
    }

    public Address getAddress() {
        return address;
    }
}

public class OptionalNavigationExample {

    public static void main(String[] args) {
        // Sample users with different levels of missing data
        User user1 = new User(new Address(new City(new ZipCode("12345"))));
        User user2 = new User(new Address(new City(null))); // ZipCode missing
        User user3 = new User(null);                        // Address missing
        User user4 = null;                                  // User is null

        System.out.println(getZipCode(user1)); // Outputs: 12345
        System.out.println(getZipCode(user2)); // Outputs: 00000 (default)
        System.out.println(getZipCode(user3)); // Outputs: 00000 (default)
        System.out.println(getZipCode(user4)); // Outputs: 00000 (default)
    }

    static String getZipCode(User user) {
        return Optional.ofNullable(user)
                .map(User::getAddress)
                .map(Address::getCity)
                .map(City::getZipCode)
                .map(ZipCode::getCode)
                .orElse("00000");  // Default value if any level is missing
    }
}
```

6.3.1 Explanation:

- `Optional.ofNullable(user)` starts the chain, wrapping a possibly null `User`.
- Each subsequent `map()` call safely extracts the next nested object if present.
- If at any step the value is `null`, the chain short-circuits to an empty `Optional`.
- Finally, `orElse("00000")` provides a default zipcode if any nested value is missing.

6.3.2 Benefits:

- **Readability:** The chain clearly expresses the navigation path without nested null checks.
- **Safety:** No risk of `NullPointerException`.
- **Conciseness:** Much less boilerplate compared to traditional null handling.
- **Graceful fallback:** Defaults are easily provided with `orElse`.

This pattern elegantly solves the challenge of safely extracting deeply nested values in Java, promoting a functional and declarative style that is easier to read, write, and maintain.

Chapter 7.

Functional Error Handling

1. Using `Either`, `Try` Patterns (Custom Implementations)
2. Function Composition with Error Handling
3. Example: Validating User Input Functionally

7 Functional Error Handling

7.1 Using `Either`, `Try` Patterns (Custom Implementations)

Traditional error handling in Java primarily relies on exceptions and sometimes returning `null` to signal failure. While exceptions are powerful, they come with several drawbacks: they disrupt normal control flow, are often costly to create, and encourage imperative "try-catch" blocks scattered throughout the code. Returning `null` to indicate errors can lead to subtle bugs and `NullPointerException`s if not handled carefully.

Functional programming offers more expressive alternatives to model computations that may fail, avoiding exceptions as control flow and providing better composability. Two common patterns inspired by functional languages are **`Either`** and **`Try`** monads. They encapsulate the result of a computation that can succeed or fail, making error handling explicit and fluent.

7.1.1 Conceptual Model

- **`Either<L, R>`** represents a value of two possible types:
  - **Left (L)** usually indicates failure or an error.
  - **Right (R)** represents a successful result.

This design explicitly separates success from failure, forcing the programmer to handle both cases without exceptions.

- **`Try<T>`** models computations that can throw exceptions:
  - It wraps either a successful result (`Success`) or an exception (`Failure`), allowing chaining operations that may fail without immediate exception throwing.

7.1.2 Simple Custom Implementation of `Either`

```
public abstract class Either<L, R> {

    public abstract boolean isRight();
    public abstract boolean isLeft();
    public abstract L getLeft();
    public abstract R getRight();

    public static <L, R> Either<L, R> right(R value) {
        return new Right<>(value);
    }

    public static <L, R> Either<L, R> left(L value) {
```

```
        return new Left<>(value);
    }

    private static final class Right<L, R> extends Either<L, R> {
        private final R value;

        Right(R value) { this.value = value; }
        public boolean isRight() { return true; }
        public boolean isLeft() { return false; }
        public L getLeft() { throw new IllegalStateException("No left value"); }
        public R getRight() { return value; }
    }

    private static final class Left<L, R> extends Either<L, R> {
        private final L value;

        Left(L value) { this.value = value; }
        public boolean isRight() { return false; }
        public boolean isLeft() { return true; }
        public L getLeft() { return value; }
        public R getRight() { throw new IllegalStateException("No right value"); }
    }
}
```

Usage example:

```
Either<String, Integer> parseInt(String s) {
    try {
        return Either.right(Integer.parseInt(s));
    } catch (NumberFormatException e) {
        return Either.left("Invalid number: " + s);
    }
}

Either<String, Integer> result = parseInt("123");
if (result.isRight()) {
    System.out.println("Parsed value: " + result.getRight());
} else {
    System.out.println("Error: " + result.getLeft());
}
```

Full runnable code:

```
public abstract class Either<L, R> {

    public abstract boolean isRight();
    public abstract boolean isLeft();
    public abstract L getLeft();
    public abstract R getRight();

    public static <L, R> Either<L, R> right(R value) {
        return new Right<>(value);
    }

    public static <L, R> Either<L, R> left(L value) {
        return new Left<>(value);
```

```
    }

    private static final class Right<L, R> extends Either<L, R> {
        private final R value;

        Right(R value) { this.value = value; }

        public boolean isRight() { return true; }
        public boolean isLeft() { return false; }
        public L getLeft() { throw new IllegalStateException("No left value"); }
        public R getRight() { return value; }
    }

    private static final class Left<L, R> extends Either<L, R> {
        private final L value;

        Left(L value) { this.value = value; }

        public boolean isRight() { return false; }
        public boolean isLeft() { return true; }
        public L getLeft() { return value; }
        public R getRight() { throw new IllegalStateException("No right value"); }
    }

    // Usage example in main:
    public static void main(String[] args) {
        Either<String, Integer> result1 = parseInt("123");
        printResult(result1);

        Either<String, Integer> result2 = parseInt("abc");
        printResult(result2);
    }

    static Either<String, Integer> parseInt(String s) {
        try {
            return Either.right(Integer.parseInt(s));
        } catch (NumberFormatException e) {
            return Either.left("Invalid number: " + s);
        }
    }

    static void printResult(Either<String, Integer> result) {
        if (result.isRight()) {
            System.out.println("Parsed value: " + result.getRight());
        } else {
            System.out.println("Error: " + result.getLeft());
        }
    }
}
```

7.1.3 Simple Custom Implementation of Try

```
public abstract class Try<T> {

    public abstract boolean isSuccess();
    public abstract T get() throws Exception;
    public abstract Exception getException();

    public static <T> Try<T> of(CheckedSupplier<T> supplier) {
        try {
            return new Success<>(supplier.get());
        } catch (Exception e) {
            return new Failure<>(e);
        }
    }

    public interface CheckedSupplier<T> {
        T get() throws Exception;
    }

    private static final class Success<T> extends Try<T> {
        private final T value;
        Success(T value) { this.value = value; }
        public boolean isSuccess() { return true; }
        public T get() { return value; }
        public Exception getException() { return null; }
    }

    private static final class Failure<T> extends Try<T> {
        private final Exception exception;
        Failure(Exception exception) { this.exception = exception; }
        public boolean isSuccess() { return false; }
        public T get() throws Exception { throw exception; }
        public Exception getException() { return exception; }
    }
}
```

Usage example:

```
Try<Integer> result = Try.of(() -> Integer.parseInt("abc"));

if (result.isSuccess()) {
    System.out.println("Parsed: " + result.get());
} else {
    System.out.println("Failed with: " + result.getException().getMessage());
}
```

Full runnable code:

```
public class TryExample {

    public static void main(String[] args) {
        Try<Integer> result = Try.of(() -> Integer.parseInt("abc"));

        if (result.isSuccess()) {
```

```
            try {
                System.out.println("Parsed: " + result.get());
            } catch (Exception e) {
                // This shouldn't happen because we already checked isSuccess()
            }
        } else {
            System.out.println("Failed with: " + result.getException().getMessage());
        }
    }

    public static abstract class Try<T> {

        public abstract boolean isSuccess();
        public abstract T get() throws Exception;
        public abstract Exception getException();

        public static <T> Try<T> of(CheckedSupplier<T> supplier) {
            try {
                return new Success<>(supplier.get());
            } catch (Exception e) {
                return new Failure<>(e);
            }
        }

        public interface CheckedSupplier<T> {
            T get() throws Exception;
        }

        private static final class Success<T> extends Try<T> {
            private final T value;

            Success(T value) {
                this.value = value;
            }

            public boolean isSuccess() {
                return true;
            }

            public T get() {
                return value;
            }

            public Exception getException() {
                return null;
            }
        }

        private static final class Failure<T> extends Try<T> {
            private final Exception exception;

            Failure(Exception exception) {
                this.exception = exception;
            }

            public boolean isSuccess() {
                return false;
            }
```

```
            public T get() throws Exception {
                throw exception;
            }

            public Exception getException() {
                return exception;
            }
        }
    }
}
```

7.1.4 Benefits of `Either` and `Try`

- **Explicit error handling:** Errors are part of the type, making handling mandatory and visible.
- **No exceptions for control flow:** Unlike exceptions, these types model failure as a normal value.
- **Better composition:** You can chain operations, map over success values, or flatMap to compose dependent computations, improving readability.
- **Separation of concerns:** Error information is preserved and can be propagated or transformed cleanly.

7.1.5 Summary

By adopting `Either` and `Try` patterns in Java, you gain a more functional and robust way to handle errors that avoids the pitfalls of exceptions and nulls. These patterns encourage explicit error propagation, reduce boilerplate try-catch blocks, and foster composable, maintainable code that clearly distinguishes success from failure.

7.2 Function Composition with Error Handling

In functional programming, composing small, reusable functions is a key principle. However, when these functions can fail—returning errors instead of plain values—composition becomes more challenging. Traditional exception handling often breaks the flow and requires verbose try-catch blocks, making the code harder to read and maintain.

Using functional error handling constructs like `Either` or `Try` (introduced in the previous section) enables elegant composition by representing computations that may succeed or fail explicitly. This allows chaining operations safely, propagating errors automatically, and preserving error context without throwing exceptions.

7.2.1 The challenge: chaining computations that might fail

Suppose you have functions that return `Either<L, R>` or `Try<T>`. Composing them requires passing the successful result from one function to the next while short-circuiting the pipeline if an error occurs.

7.2.2 Using `Either` for composition with `flatMap`

Recall our custom `Either<L, R>` implementation from the previous section. To compose functions, we add a `flatMap` method:

```
public abstract class Either<L, R> {
    // ... existing methods ...

    public abstract <U> Either<L, U> flatMap(Function<? super R, Either<L, U>> mapper);

    // Inside Right:
    private static final class Right<L, R> extends Either<L, R> {
        // ...
        public <U> Either<L, U> flatMap(Function<? super R, Either<L, U>> mapper) {
            return mapper.apply(value);
        }
    }

    // Inside Left:
    private static final class Left<L, R> extends Either<L, R> {
        // ...
        public <U> Either<L, U> flatMap(Function<? super R, Either<L, U>> mapper) {
            return (Either<L, U>) this; // propagate the error without calling mapper
        }
    }
}
```

7.2.3 Example: Composing functions with `Either`

```
Either<String, Integer> parse(String s) {
    try {
        return Either.right(Integer.parseInt(s));
    } catch (NumberFormatException e) {
        return Either.left("Invalid number: " + s);
    }
}

Either<String, Integer> reciprocal(int x) {
    if (x == 0) {
        return Either.left("Division by zero");
    } else {
```

```
        return Either.right(1 / x);
    }
}

public void composeExample() {
    Either<String, Integer> result = parse("10")
        .flatMap(this::reciprocal);

    result.isRight()
        ? System.out.println("Result: " + result.getRight())
        : System.out.println("Error: " + result.getLeft());
}
```

Here, if `parse` fails, the error propagates, and `reciprocal` is never called, short-circuiting the chain. If both succeed, the final result is printed.

7.2.4 Using `Try` for safe composition

Similarly, a `Try` type can have a `flatMap` method to chain computations:

```
public abstract class Try<T> {
    // ... existing methods ...

    public abstract <U> Try<U> flatMap(Function<? super T, Try<U>> mapper);

    // Success implementation:
    private static final class Success<T> extends Try<T> {
        // ...
        public <U> Try<U> flatMap(Function<? super T, Try<U>> mapper) {
            try {
                return mapper.apply(value);
            } catch (Exception e) {
                return new Failure<>(e);
            }
        }
    }

    // Failure implementation:
    private static final class Failure<T> extends Try<T> {
        // ...
        public <U> Try<U> flatMap(Function<? super T, Try<U>> mapper) {
            return (Try<U>) this;
        }
    }
}
```

Example usage:

```
Try<Integer> parseTry(String s) {
    return Try.of(() -> Integer.parseInt(s));
}
```

```
Try<Integer> reciprocalTry(int x) {
    return x == 0
        ? new Try.Failure<>(new ArithmeticException("Division by zero"))
        : new Try.Success<>(1 / x);
}

public void tryComposeExample() {
    Try<Integer> result = parseTry("0")
        .flatMap(this::reciprocalTry);

    if (result.isSuccess()) {
        System.out.println("Result: " + result.get());
    } else {
        System.out.println("Failed: " + result.getException().getMessage());
    }
}
```

7.2.5 Recovering from errors

Both `Either` and `Try` support recovery methods:

- **`Either`** can provide fallback values via a method like `orElse` or `fold`.
- **`Try`** typically has `recover` or `recoverWith` to replace failures with a default or alternate success.

Example recovery with `Try`:

```
Try<Integer> safeResult = parseTry("abc")
    .flatMap(this::reciprocalTry)
    .recover(ex -> 0); // fallback to 0 on failure
```

7.2.6 Summary

- Composing functions that may fail is simplified by `Either` and `Try` monads.
- `flatMap` (or equivalent) lets you chain computations, automatically short-circuiting on errors.
- Errors are explicit values, preserving error context without exceptions.
- You can recover from failures gracefully, maintaining fluent pipelines.
- This approach encourages clean, readable, and maintainable error handling in functional Java code.

By adopting these patterns, Java developers gain powerful tools for robust error propagation and composition, moving beyond the pitfalls of exceptions and nulls.

7.3 Example: Validating User Input Functionally

Validating user input is a common task that often involves multiple checks on different fields. Traditional validation approaches either throw exceptions on the first failure or accumulate errors in an ad hoc manner. Functional error handling patterns like `Either` allow us to build composable validation pipelines that clearly separate success and failure cases, enabling us to collect all errors without exceptions.

In this example, we'll create a simple user input validator that checks a username and age, accumulating errors if either validation fails. We'll use a custom `Either` type that supports error accumulation in a `List<String>` on the left side.

Custom `Either` supporting error accumulation

```
import java.util.*;
import java.util.function.Function;

abstract class Either<L, R> {
    public abstract boolean isRight();
    public abstract R getRight();
    public abstract L getLeft();

    // Factory methods
    public static <L, R> Either<L, R> right(R value) {
        return new Right<>(value);
    }

    public static <L, R> Either<L, R> left(L value) {
        return new Left<>(value);
    }

    // Combine two Eithers accumulating errors (assuming L is List<String>)
    public Either<List<String>, R> combine(Either<List<String>, R> other,
                                           Function<R, R> combineFunc) {
        if (this.isRight() && other.isRight()) {
            return right(combineFunc.apply(other.getRight()));
        } else {
            List<String> errors = new ArrayList<>();
            if (this.isLeft()) errors.addAll((List<String>)this.getLeft());
            if (other.isLeft()) errors.addAll((List<String>)other.getLeft());
            return left(errors);
        }
    }

    // Inner classes
    private static final class Right<L, R> extends Either<L, R> {
        private final R value;
        Right(R value) { this.value = value; }
        public boolean isRight() { return true; }
        public R getRight() { return value; }
        public L getLeft() { throw new NoSuchElementException("No Left value"); }
    }

    private static final class Left<L, R> extends Either<L, R> {
        private final L value;
```

```
        Left(L value) { this.value = value; }
        public boolean isRight() { return false; }
        public R getRight() { throw new NoSuchElementException("No Right value"); }
        public L getLeft() { return value; }
    }
}
```

User and validation functions

```
class User {
    final String username;
    final int age;

    User(String username, int age) {
        this.username = username;
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{username='" + username + "', age=" + age + "}";
    }
}
```

```
public class UserValidationExample {

    static Either<List<String>, String> validateUsername(String username) {
        List<String> errors = new ArrayList<>();
        if (username == null || username.isEmpty()) {
            errors.add("Username cannot be empty");
        }
        if (username != null && username.length() < 3) {
            errors.add("Username must be at least 3 characters");
        }
        return errors.isEmpty() ? Either.right(username) : Either.left(errors);
    }

    static Either<List<String>, Integer> validateAge(int age) {
        if (age < 18) {
            return Either.left(Collections.singletonList("Age must be at least 18"));
        }
        return Either.right(age);
    }

    static Either<List<String>, User> validateUser(String username, int age) {
        Either<List<String>, String> validUsername = validateUsername(username);
        Either<List<String>, Integer> validAge = validateAge(age);

        if (validUsername.isRight() && validAge.isRight()) {
            return Either.right(new User(validUsername.getRight(), validAge.getRight()));
        }

        // Accumulate errors from both validations
        List<String> allErrors = new ArrayList<>();
        if (validUsername.isLeft()) allErrors.addAll(validUsername.getLeft());
```

```
        if (validAge.isLeft()) allErrors.addAll(validAge.getLeft());

        return Either.left(allErrors);
    }

    public static void main(String[] args) {
        // Test cases
        Either<List<String>, User> user1 = validateUser("Al", 20);
        Either<List<String>, User> user2 = validateUser("Alice", 17);
        Either<List<String>, User> user3 = validateUser("", 15);
        Either<List<String>, User> user4 = validateUser("Bob", 25);

        printResult(user1);
        printResult(user2);
        printResult(user3);
        printResult(user4);
    }

    static void printResult(Either<List<String>, User> result) {
        if (result.isRight()) {
            System.out.println("Validation succeeded: " + result.getRight());
        } else {
            System.out.println("Validation failed with errors:");
            result.getLeft().forEach(err -> System.out.println(" - " + err));
        }
    }
}
```

Full runnable code:

```
import java.util.*;
import java.util.function.Function;

// Either abstraction
abstract class Either<L, R> {
    public abstract boolean isRight();
    public abstract R getRight();
    public abstract L getLeft();

    public static <L, R> Either<L, R> right(R value) {
        return new Right<>(value);
    }

    public static <L, R> Either<L, R> left(L value) {
        return new Left<>(value);
    }

    private static final class Right<L, R> extends Either<L, R> {
        private final R value;
        Right(R value) { this.value = value; }
        public boolean isRight() { return true; }
        public R getRight() { return value; }
        public L getLeft() { throw new NoSuchElementException("No Left value"); }
    }

    private static final class Left<L, R> extends Either<L, R> {
        private final L value;
```

```
        Left(L value) { this.value = value; }
        public boolean isRight() { return false; }
        public R getRight() { throw new NoSuchElementException("No Right value"); }
        public L getLeft() { return value; }
    }
}

// User class
class User {
    final String username;
    final int age;

    User(String username, int age) {
        this.username = username;
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{username='" + username + "', age=" + age + "}";
    }
}

// Validation and demo logic
public class UserValidationExample {

    static Either<List<String>, String> validateUsername(String username) {
        List<String> errors = new ArrayList<>();
        if (username == null || username.isEmpty()) {
            errors.add("Username cannot be empty");
        }
        if (username != null && username.length() < 3) {
            errors.add("Username must be at least 3 characters");
        }
        return errors.isEmpty() ? Either.right(username) : Either.left(errors);
    }

    static Either<List<String>, Integer> validateAge(int age) {
        if (age < 18) {
            return Either.left(Collections.singletonList("Age must be at least 18"));
        }
        return Either.right(age);
    }

    static Either<List<String>, User> validateUser(String username, int age) {
        Either<List<String>, String> validUsername = validateUsername(username);
        Either<List<String>, Integer> validAge = validateAge(age);

        if (validUsername.isRight() && validAge.isRight()) {
            return Either.right(new User(validUsername.getRight(), validAge.getRight()));
        }

        List<String> allErrors = new ArrayList<>();
        if (!validUsername.isRight()) allErrors.addAll(validUsername.getLeft());
        if (!validAge.isRight()) allErrors.addAll(validAge.getLeft());

        return Either.left(allErrors);
    }
```

```
    public static void main(String[] args) {
        List<Either<List<String>, User>> testCases = Arrays.asList(
            validateUser("Al", 20),
            validateUser("Alice", 17),
            validateUser("", 15),
            validateUser("Bob", 25)
        );

        for (Either<List<String>, User> result : testCases) {
            printResult(result);
        }
    }

    static void printResult(Either<List<String>, User> result) {
        if (result.isRight()) {
            System.out.println("YES Validation succeeded: " + result.getRight());
        } else {
            System.out.println("NO Validation failed with errors:");
            result.getLeft().forEach(err -> System.out.println(" - " + err));
        }
    }
}
```

7.3.1 Explanation

- **Validation functions** return `Either<List<String>, T>` to encapsulate multiple errors or a valid value.
- The `validateUser` method combines results, accumulating all errors rather than failing fast.
- The `main` method demonstrates multiple input scenarios with outputs showing success or error accumulation.
- This functional style avoids exceptions and promotes clean error propagation and composition.

7.3.2 Sample Output

```
Validation failed with errors:
 - Username must be at least 3 characters

Validation failed with errors:
 - Age must be at least 18

Validation failed with errors:
 - Username cannot be empty
```

```
 - Age must be at least 18

Validation succeeded: User{username='Bob', age=25}
```

This example demonstrates how functional error handling with `Either` enables expressive, composable validation pipelines that accumulate errors cleanly and improve code readability and robustness.

Chapter 8.

Functional Design Patterns

1. Strategy Pattern with Lambdas
2. Command Pattern using Functional Interfaces
3. Observer Pattern with Functional Callbacks
4. Example: Event Handling in GUI Applications

8 Functional Design Patterns

8.1 Strategy Pattern with Lambdas

The **Strategy Pattern** is a classic behavioral design pattern that enables selecting an algorithm's behavior at runtime. Traditionally, this involves defining a family of algorithms, encapsulating each one in its own class, and making them interchangeable via a common interface. This allows the client to choose or switch the algorithm without changing the context code.

Traditional Strategy Pattern Structure

Typically, you create an interface representing the strategy, e.g.,

```
interface SortingStrategy {
    void sort(int[] array);
}
```

Then provide multiple implementations:

```
class BubbleSort implements SortingStrategy {
    public void sort(int[] array) {
        // Bubble sort implementation
    }
}

class QuickSort implements SortingStrategy {
    public void sort(int[] array) {
        // Quick sort implementation
    }
}
```

The client code uses the strategy via the interface:

```
class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}
```

While effective, this pattern involves boilerplate classes and verbosity.

Simplifying with Lambdas and Functional Interfaces

Java 8's introduction of **functional interfaces** and **lambdas** greatly simplifies the Strategy pattern. Instead of creating multiple concrete classes, you can define a single functional

interface and pass behavior as a lambda expression directly.

For example, the `SortingStrategy` interface can be replaced by a functional interface like:

```
@FunctionalInterface
interface SortingStrategy {
    void sort(int[] array);
}
```

Using lambdas, you can now easily provide different strategies inline without separate classes:

```
SortingStrategy bubbleSort = array -> {
    // Bubble sort logic
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = 0; j < array.length - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
};

SortingStrategy quickSort = array -> Arrays.sort(array); // Using built-in sort
```

The client can switch strategies effortlessly:

```
class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(int[] array) {
        strategy.sort(array);
    }
}
```

Usage:

```
public static void main(String[] args) {
    int[] data = {5, 2, 8, 3, 1};
    Sorter sorter = new Sorter(bubbleSort);

    sorter.sort(data);
    System.out.println("Bubble sorted: " + Arrays.toString(data));

    sorter.setStrategy(quickSort);
    int[] data2 = {9, 7, 4, 6};
```

```
    sorter.sort(data2);
    System.out.println("Quick sorted: " + Arrays.toString(data2));
}
```

Full runnable code:

```
import java.util.Arrays;

// Define functional interface for sorting strategies
@FunctionalInterface
interface SortingStrategy {
    void sort(int[] array);
}

// Sorter class using a strategy
class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(int[] array) {
        strategy.sort(array);
    }
}

public class StrategyPatternWithLambda {
    public static void main(String[] args) {
        // Lambda-based bubble sort strategy
        SortingStrategy bubbleSort = array -> {
            for (int i = 0; i < array.length - 1; i++) {
                for (int j = 0; j < array.length - 1 - i; j++) {
                    if (array[j] > array[j + 1]) {
                        int temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                    }
                }
            }
        };

        // Lambda-based quick sort strategy using built-in method
        SortingStrategy quickSort = Arrays::sort;

        // Using bubble sort
        int[] data = {5, 2, 8, 3, 1};
        Sorter sorter = new Sorter(bubbleSort);
        sorter.sort(data);
        System.out.println("Bubble sorted: " + Arrays.toString(data));

        // Using quick sort
        int[] data2 = {9, 7, 4, 6};
```

```
        sorter.setStrategy(quickSort);
        sorter.sort(data2);
        System.out.println("Quick sorted: " + Arrays.toString(data2));
    }
}
```

Other Practical Examples

Discount Strategy

```
@FunctionalInterface
interface DiscountStrategy {
    double applyDiscount(double price);
}

DiscountStrategy noDiscount = price -> price;
DiscountStrategy seasonalDiscount = price -> price * 0.9;
DiscountStrategy clearanceDiscount = price -> price * 0.5;

double originalPrice = 100.0;
System.out.println("Seasonal price: " + seasonalDiscount.applyDiscount(originalPrice));
```

Logging Behavior

```
@FunctionalInterface
interface Logger {
    void log(String message);
}

Logger consoleLogger = msg -> System.out.println("[Console] " + msg);
Logger fileLogger = msg -> System.out.println("[File] " + msg); // Simplified

consoleLogger.log("App started");
fileLogger.log("File saved");
```

8.1.1 Summary

By using Java lambdas and functional interfaces, the Strategy pattern becomes lightweight and flexible. Instead of multiple boilerplate classes, behaviors are expressed concisely as lambdas, making your code more readable and maintainable while retaining full expressiveness and runtime flexibility. This functional approach suits many use cases like sorting, discounts, logging, and more.

8.2 Command Pattern using Functional Interfaces

The **Command Pattern** is a behavioral design pattern that encapsulates a request or an operation as an object. This allows you to parameterize clients with different requests, queue or log operations, and support undoable actions. The core idea is to separate the *invoker* (who triggers the command) from the *executor* (the code that performs the action), promoting loose coupling and greater flexibility.

Traditional Command Pattern

Traditionally, you define a `Command` interface:

```
public interface Command {
    void execute();
}
```

Each operation implements this interface as a concrete class:

```
class TurnOnLightCommand implements Command {
    private Light light;

    public TurnOnLightCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}
```

The invoker holds a reference to the `Command` and calls `execute()` without knowing the details.

Using Functional Interfaces and Lambdas

Java's **functional interfaces** like `Runnable`, `Consumer<T>`, or custom ones perfectly fit the Command pattern's single-method interface. Lambdas enable writing commands concisely without boilerplate classes.

For example, a simple command can be represented as a `Runnable` lambda:

```
Runnable turnOnLight = () -> System.out.println("Light turned ON");
Runnable turnOffLight = () -> System.out.println("Light turned OFF");
```

An invoker class might simply store and execute these commands:

```
class RemoteControl {
    private Runnable command;

    public void setCommand(Runnable command) {
        this.command = command;
    }
```

```
    public void pressButton() {
        if (command != null) {
            command.run();
        }
    }
}
```

Usage:

```
public static void main(String[] args) {
    RemoteControl remote = new RemoteControl();

    remote.setCommand(turnOnLight);
    remote.pressButton();  // Output: Light turned ON

    remote.setCommand(turnOffLight);
    remote.pressButton();  // Output: Light turned OFF
}
```

Full runnable code:

```
public class CommandPatternWithLambdas {

    public static void main(String[] args) {
        // Define commands as lambdas
        Runnable turnOnLight = () -> System.out.println("Light turned ON");
        Runnable turnOffLight = () -> System.out.println("Light turned OFF");

        // Create the invoker
        RemoteControl remote = new RemoteControl();

        // Use the ON command
        remote.setCommand(turnOnLight);
        remote.pressButton();  // Output: Light turned ON

        // Use the OFF command
        remote.setCommand(turnOffLight);
        remote.pressButton();  // Output: Light turned OFF
    }

    // Invoker class
    static class RemoteControl {
        private Runnable command;

        public void setCommand(Runnable command) {
            this.command = command;
        }

        public void pressButton() {
            if (command != null) {
                command.run();
            }
        }
    }
}
```

Advantages of Functional Command Pattern

- **Deferred execution:** Commands can be created and passed around without immediate execution, enabling flexible scheduling or queuing.
- **Undo functionality:** Commands can capture the state needed to reverse their actions.
- **Macro commands:** Multiple commands can be combined into one to execute a workflow.

Composing Macro Commands

Using `Runnable`'s default method `andThen`, you can compose commands:

```
Runnable startEngine = () -> System.out.println("Engine started");
Runnable playMusic = () -> System.out.println("Music playing");

Runnable morningRoutine = startEngine.andThen(playMusic);

morningRoutine.run();
// Output:
// Engine started
// Music playing
```

Using Custom Functional Interfaces with Parameters

Sometimes commands require parameters or return results. Define a custom interface:

```
@FunctionalInterface
interface CommandWithArg<T> {
    void execute(T arg);
}

CommandWithArg<String> printMessage = msg -> System.out.println("Message: " + msg);

printMessage.execute("Hello, Command Pattern!");
```

8.2.1 Summary

Java functional interfaces simplify the Command pattern by eliminating boilerplate command classes, making commands easy to create, pass, and compose as lambdas. This approach supports deferred execution, flexible workflows, and features like undo or macros, all with concise and readable code. It's a natural fit for event handling, task scheduling, and any context where actions need to be encapsulated and executed flexibly.

8.3 Observer Pattern with Functional Callbacks

The **Observer Pattern** is a behavioral design pattern that establishes a one-to-many relationship between objects: when the *subject* changes its state, all its *observers* are notified and updated automatically. This pattern is fundamental in event-driven programming, GUIs, and reactive systems where components react asynchronously to events or data changes.

Traditional Observer Pattern

In classic Java, the pattern often involves interfaces like `Observer` and `Observable` or listener interfaces such as `PropertyChangeListener`. Observers implement these interfaces and register themselves with the subject to receive updates.

For example, using `PropertyChangeListener`:

```
PropertyChangeSupport support = new PropertyChangeSupport(this);

support.addPropertyChangeListener(evt -> {
    System.out.println("Property " + evt.getPropertyName() + " changed to " + evt.getNewValue());
});
```

Functional Programming Simplifies Observers

Java 8 introduced lambdas and method references, which simplify observers dramatically by letting you register behavior inline without implementing entire classes. Observers become *functional callbacks*—just functions invoked on event occurrence.

Instead of writing verbose listener classes, you register a lambda that reacts to events:

```
subject.addListener(event -> System.out.println("Event received: " + event));
```

This promotes more readable, concise, and maintainable code.

Example: Custom Event System with Functional Callbacks

Let's implement a simple observable subject that accepts functional observers.

```
import java.util.*;
import java.util.function.Consumer;

class EventSource<T> {
    private final List<Consumer<T>> observers = new ArrayList<>();

    public void subscribe(Consumer<T> observer) {
        observers.add(observer);
    }

    public void unsubscribe(Consumer<T> observer) {
        observers.remove(observer);
    }

    public void notifyObservers(T event) {
        for (Consumer<T> observer : observers) {
```

```
            observer.accept(event);
        }
    }
}
```

Usage:

```
public class ObserverExample {
    public static void main(String[] args) {
        EventSource<String> eventSource = new EventSource<>();

        // Register observers with lambdas
        eventSource.subscribe(event -> System.out.println("Observer 1 received: " + event));
        eventSource.subscribe(event -> System.out.println("Observer 2 received: " + event.toUpperCase()));

        eventSource.notifyObservers("Hello Observers!");

        // Output:
        // Observer 1 received: Hello Observers!
        // Observer 2 received: HELLO OBSERVERS!
    }
}
```

Full runnable code:

```
import java.util.*;
import java.util.function.Consumer;

// Observable subject
class EventSource<T> {
    private final List<Consumer<T>> observers = new ArrayList<>();

    public void subscribe(Consumer<T> observer) {
        observers.add(observer);
    }

    public void unsubscribe(Consumer<T> observer) {
        observers.remove(observer);
    }

    public void notifyObservers(T event) {
        for (Consumer<T> observer : observers) {
            observer.accept(event);
        }
    }
}

// Demo with functional observers
public class ObserverExample {
    public static void main(String[] args) {
        EventSource<String> eventSource = new EventSource<>();

        // Register observers using lambdas
        eventSource.subscribe(event -> System.out.println("Observer 1 received: " + event));
        eventSource.subscribe(event -> System.out.println("Observer 2 received: " + event.toUpperCase()));
```

```
        eventSource.notifyObservers("Hello Observers!");
    }
}
```

Real-World Relevance

- **GUI applications:** Button clicks, mouse movements, and other UI events are handled by observers registered as callbacks.
- **Event-driven architectures:** Services subscribe to events and react asynchronously, decoupling producers and consumers.
- **Reactive programming:** Streams of data changes trigger chained callbacks or operators, enabling responsive, declarative data flows.

Managing Subscriptions Easily

Because observers are functions, you can store and remove them effortlessly. This is especially useful in long-lived systems where listeners must be unsubscribed to prevent memory leaks:

```
Consumer<String> observer = event -> System.out.println("Received event: " + event);
eventSource.subscribe(observer);

// Later, unsubscribe if needed
eventSource.unsubscribe(observer);
```

8.3.1 Summary

Functional callbacks transform the Observer pattern into a lightweight, flexible mechanism for event handling. By leveraging lambdas and method references, Java developers can implement event subscription and notification without boilerplate listener classes, making event-driven code more concise, readable, and maintainable. This functional approach aligns perfectly with GUI frameworks, reactive streams, and modern event-driven architectures.

8.4 Example: Event Handling in GUI Applications

Java's GUI frameworks, such as **Swing**, rely heavily on event-driven programming where components notify listeners of user actions like button clicks or text input changes. With Java 8 and later, **functional interfaces** and **lambdas** simplify event handling by replacing verbose anonymous classes with concise, readable functional callbacks.

Below is a runnable example demonstrating a simple Swing GUI that uses lambdas to handle button clicks and window closing events.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FunctionalSwingExample {

    public static void main(String[] args) {
        // Create main frame (window)
        JFrame frame = new JFrame("Functional Event Handling Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        // Create a button
        JButton clickButton = new JButton("Click Me");

        // Create a label to show click count
        JLabel label = new JLabel("Button not clicked yet");

        // Use lambda for button click event handler (ActionListener)
        clickButton.addActionListener(e -> {
            String currentText = label.getText();
            int count = 0;
            if (currentText.contains("clicked")) {
                count = Integer.parseInt(currentText.replaceAll("\\D+", "")) + 1;
            }
            label.setText("Button clicked " + count + " times");
        });

        // Use lambda for window close event (WindowListener)
        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.out.println("Window is closing. Bye!");
            }
        });

        // Layout components vertically
        frame.setLayout(new FlowLayout());
        frame.add(clickButton);
        frame.add(label);

        // Show the window
        frame.setVisible(true);
    }
}
```

8.4.1 Explanation

- **Button click:** The `addActionListener` method accepts a functional interface `ActionListener` (single method `actionPerformed`). Using a lambda expression, the code becomes concise without anonymous inner classes.
- **Window closing:** The `WindowListener` interface has multiple methods, so we use the adapter class `WindowAdapter` and override only the relevant method (`windowClosing`)

with a lambda-friendly style (anonymous subclass).
- **Label update:** We read the label's text, parse the current count, increment it, and update the label—all inside the lambda callback, demonstrating inline event logic.

8.4.2 Why functional event handling?

- **Conciseness:** Lambdas replace bulky anonymous classes, improving readability.
- **Expressiveness:** Event logic lives close to where events are registered.
- **Maintainability:** Easier to refactor and follow compared to nested classes.

This example illustrates how functional design patterns mesh naturally with GUI event handling, making your Java desktop apps more elegant and easier to maintain. Lambdas help keep event-driven code clear and focused, boosting developer productivity in GUI development.

Chapter 9.

Higher-Order Functions and Currying

1. Defining and Using Higher-Order Functions
2. Partial Application and Currying in Java
3. Example: Building Reusable Validation Functions

9 Higher-Order Functions and Currying

9.1 Defining and Using Higher-Order Functions

In functional programming, **higher-order functions (HOFs)** are functions that can either take other functions as parameters or return functions as results—or both. This capability enables powerful abstraction and code reuse by letting you build flexible, customizable behavior without duplicating code.

Why Higher-Order Functions Matter

Traditional programming often involves writing many similar functions with small variations. Higher-order functions allow you to **capture these variations as functions themselves** and pass them around, letting you compose complex behavior from simpler building blocks.

They:

- **Encapsulate behavior:** You can inject custom logic via function arguments.
- **Enable reuse:** Write generic algorithms that work with different operations.
- **Support function composition:** Chain and combine behaviors elegantly.

Java Support for Higher-Order Functions

Since Java 8, the introduction of **functional interfaces** (interfaces with a single abstract method) and **lambdas** makes working with functions as first-class values possible. Common functional interfaces like `Function<T,R>`, `Predicate<T>`, and `Consumer<T>` facilitate passing behavior around.

Example 1: Function as Parameter

Suppose you want a generic method to transform strings using different operations:

```
import java.util.function.Function;

public class HigherOrderExample {

    // Higher-order function: takes a Function<String,String> as argument
    public static String transformString(String input, Function<String, String> transformer) {
        return transformer.apply(input);
    }

    public static void main(String[] args) {
        String result1 = transformString("hello", s -> s.toUpperCase());
        String result2 = transformString("functional", s -> s + " programming");

        System.out.println(result1); // Output: HELLO
        System.out.println(result2); // Output: functional programming
    }
}
```

Here, `transformString` is a higher-order function that accepts a function to customize how the string is transformed.

Example 2: Function Returning a Function

Higher-order functions can also return functions. This pattern is useful for creating **configurable behavior**:

```
import java.util.function.Function;

public class FunctionReturningFunction {

    // Returns a function that adds a fixed prefix to input strings
    public static Function<String, String> prefixer(String prefix) {
        return s -> prefix + s;
    }

    public static void main(String[] args) {
        Function<String, String> helloPrefixer = prefixer("Hello, ");
        Function<String, String> errorPrefixer = prefixer("Error: ");

        System.out.println(helloPrefixer.apply("Alice"));  // Output: Hello, Alice
        System.out.println(errorPrefixer.apply("File not found")); // Output: Error: File not found
    }
}
```

The `prefixer` method returns a new function customized with the provided prefix, demonstrating how functions can be dynamically created and reused.

9.1.1 Summary

Higher-order functions are fundamental to writing flexible, expressive, and reusable code in functional programming. Java's functional interfaces and lambdas provide solid support for defining and using these functions. Whether by passing functions as arguments or returning them as results, higher-order functions help encapsulate behavior, reduce duplication, and enable powerful abstractions that improve code quality and clarity.

9.2 Partial Application and Currying in Java

Partial application and **currying** are powerful functional programming techniques that allow you to create new functions by pre-filling some arguments of existing functions. Both improve code reuse and flexibility by enabling you to configure functions incrementally.

What is Partial Application?

Partial application is the process of fixing a few arguments of a multi-parameter function, producing a new function that takes the remaining arguments. For example, given a function with three parameters, partial application with the first argument fixed results in a function with two parameters.

Use case: When you know some inputs ahead of time, you can create specialized versions of generic functions without rewriting them.

What is Currying?

Currying transforms a function that takes multiple arguments into a sequence of functions, each with a single argument. For instance, a function `(a, b, c) -> r` becomes `a -> (b -> (c -> r))`.

Difference: Currying always produces unary functions nested inside each other, while partial application can fix any subset of arguments without fully nesting.

Why Use These Techniques?

- They promote **code modularity** and **reusability**.
- They help **customize functions incrementally**.
- Currying enables elegant **function composition** and pipelines.

Javas Limitations and Workarounds

Java does not have native syntax for currying or partial application, but with functional interfaces and lambdas, we can emulate these patterns.

Example: Currying in Java

```
import java.util.function.Function;

public class CurryingExample {

    // Curried function: takes 'a' and returns function taking 'b' then 'c'
    static Function<Integer, Function<Integer, Function<Integer, Integer>>> curriedAdd =
        a -> b -> c -> a + b + c;

    public static void main(String[] args) {
        // Use the curried function step by step
        Function<Integer, Function<Integer, Integer>> add5 = curriedAdd.apply(5);
        Function<Integer, Integer> add5And10 = add5.apply(10);
        int result = add5And10.apply(3); // 5 + 10 + 3 = 18

        System.out.println(result); // Output: 18

        // Or all at once
        System.out.println(curriedAdd.apply(1).apply(2).apply(3)); // Output: 6
    }
}
```

Here, `curriedAdd` is a function that takes an integer and returns a function expecting the next integer, and so on, until all three inputs are provided.

Example: Partial Application in Java

We can partially apply a function by fixing one or more arguments:

```java
import java.util.function.BiFunction;
import java.util.function.Function;

public class PartialApplicationExample {

    // Regular two-argument function
    static BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;

    // Partial application: fix the first argument
    static Function<Integer, Integer> multiplyBy5 = b -> multiply.apply(5, b);

    public static void main(String[] args) {
        System.out.println(multiply.apply(3, 4)); // 12
        System.out.println(multiplyBy5.apply(4)); // 20
    }
}
```

The `multiplyBy5` function fixes `a` as 5 and returns a new function that only requires the second argument.

9.2.1 Summary

- **Partial application** fixes some arguments of a function to create a specialized function with fewer parameters.
- **Currying** transforms a multi-parameter function into a chain of single-parameter functions.
- Java lacks built-in currying syntax but supports these concepts through functional interfaces and lambdas.
- Emulating currying and partial application increases code reuse, customization, and functional composition, making your Java code more modular and expressive.

9.3 Example: Building Reusable Validation Functions

Functional programming encourages building small, reusable functions that can be combined to solve complex problems. Validation is a perfect example: by creating **reusable, composable validators**, you can easily customize and extend validation logic without duplication.

Using Higher-Order Functions and Currying for Validation

Let's create a generic validation function that accepts a predicate and an error message. By using **partial application**, we can fix the error message upfront and later supply the input to validate.

```java
import java.util.function.Function;
import java.util.function.Predicate;
```

```java
import java.util.ArrayList;
import java.util.List;

public class ValidationExample {

    // Validator type: takes input T, returns list of error messages (empty if valid)
    @FunctionalInterface
    interface Validator<T> {
        List<String> validate(T t);

        // Combines two validators
        default Validator<T> and(Validator<T> other) {
            return t -> {
                List<String> errors = new ArrayList<>(this.validate(t));
                errors.addAll(other.validate(t));
                return errors;
            };
        }
    }

    // Higher-order function to create a validator from a predicate and an error message
    static <T> Function<String, Validator<T>> createValidator(Predicate<T> predicate) {
        return errorMessage -> t -> predicate.test(t) ? List.of() : List.of(errorMessage);
    }

    public static void main(String[] args) {
        // Explicit type parameters help with type inference
        Validator<String> notEmpty = ValidationExample.<String>createValidator(
            s -> s != null && !s.isEmpty()).apply("Must not be empty");

        Validator<String> minLength5 = ValidationExample.<String>createValidator(
            s -> s.length() >= 5).apply("Must be at least 5 characters");

        // Compose validators using 'and' method
        Validator<String> usernameValidator = notEmpty.and(minLength5);

        // Test inputs
        String input1 = "";
        String input2 = "Java";
        String input3 = "Lambda";

        System.out.println("Input1 errors: " + usernameValidator.validate(input1));
        System.out.println("Input2 errors: " + usernameValidator.validate(input2));
        System.out.println("Input3 errors: " + usernameValidator.validate(input3));
    }
}
```

9.3.1 Explanation

- `Validator<T>` represents a function that takes input `T` and returns a list of error messages.
- `createValidator` is a **higher-order function** that takes a predicate and returns a function which, when given an error message, produces a validator.

- This enables **partial application**: fixing the predicate once, then supplying different error messages to create specific rules.
- The `and` method combines validators by merging their error lists, allowing modular composition of rules.
- The example shows validation for usernames that must be non-empty and at least 5 characters long, with clear reusable components.

9.3.2 Benefits

- **Code reuse:** Define common validation patterns once and customize with different error messages.
- **Composability:** Combine simple validators to express complex validation logic cleanly.
- **Clarity:** Validation rules are explicit, declarative, and easy to maintain.

This approach leads to more modular, readable, and flexible validation code, leveraging functional programming concepts effectively in Java.

Chapter 10.

Lazy Evaluation and Infinite Streams

1. Understanding Lazy vs. Eager Evaluation
2. Creating and Using Infinite Streams
3. Example: Generating Fibonacci Numbers Lazily

10 Lazy Evaluation and Infinite Streams

10.1 Understanding Lazy vs. Eager Evaluation

In programming, **evaluation strategy** determines when expressions or computations are executed. Two common strategies are **eager** (or strict) and **lazy** evaluation.

Eager Evaluation

In eager evaluation, expressions are computed **immediately** when they are encountered. This means all data processing happens upfront, regardless of whether the results are actually used later.

Example with eager evaluation using collections:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class EagerExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Eagerly process the collection
        List<Integer> doubled = numbers.stream()
                                       .map(n -> {
                                           System.out.println("Doubling " + n);
                                           return n * 2;
                                       })
                                       .collect(Collectors.toList()); // Terminal operation triggers pr

        System.out.println(doubled);
    }
}
```

Here, the entire list is processed immediately when `collect()` is called, doubling each number and printing the operation for every element.

Lazy Evaluation

Lazy evaluation **defers computation** until the results are actually needed. This is a key concept in functional programming, enabling efficient use of CPU and memory by avoiding unnecessary work.

Java's **Streams API** supports lazy evaluation for intermediate operations like `map()`, `filter()`, or `sorted()`. These operations build a pipeline but do not process elements until a **terminal operation** (e.g., `forEach()`, `collect()`, `findFirst()`) is invoked.

Example with lazy evaluation:

```
import java.util.Arrays;
import java.util.List;
```

```java
public class LazyExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        numbers.stream()
               .map(n -> {
                   System.out.println("Mapping " + n);
                   return n * 2;
               })
               .filter(n -> {
                   System.out.println("Filtering " + n);
                   return n > 5;
               })
               .forEach(n -> System.out.println("Consumed " + n));
    }
}
```

Output shows that mapping and filtering happen **only as needed**, per element:

```
Mapping 1
Filtering 2
Mapping 2
Filtering 4
Mapping 3
Filtering 6
Consumed 6
Mapping 4
Filtering 8
Consumed 8
Mapping 5
Filtering 10
Consumed 10
```

Notice the operations interleave and stop early if possible, avoiding processing the entire collection upfront.

Why Does This Matter?

- **Performance:** Lazy evaluation avoids unnecessary computation by processing only what's needed, which is critical for large or infinite data sources.
- **Memory Usage:** Eager evaluation may hold entire intermediate results in memory, while lazy evaluation processes elements one-by-one, reducing memory footprint.
- **Infinite Data:** Lazy evaluation allows working with infinite streams since values are generated on demand, impossible with eager processing.

10.1.1 Summary

- **Eager evaluation** computes immediately, processing entire collections upfront.
- **Lazy evaluation** delays computation until results are requested, enabling efficiency.
- Java's Streams combine both: intermediate operations are lazy, terminal operations trigger processing.
- Understanding this distinction helps you write performant, resource-friendly functional code that scales well to large or infinite data.

10.2 Creating and Using Infinite Streams

Infinite streams are streams without a predefined end—they can generate an unbounded sequence of elements. Unlike collections with fixed size, infinite streams rely on **lazy evaluation** to produce elements on demand, making it possible to work with potentially limitless data safely and efficiently.

How Java Supports Infinite Streams

Java's Streams API provides two primary methods for creating infinite streams:

- **`Stream.iterate(seed, UnaryOperator)`**: Generates an infinite stream by repeatedly applying a function to the previous element.
- **`Stream.generate(Supplier)`**: Produces an infinite stream by repeatedly calling a supplier function that generates elements independently.

Because of lazy evaluation, no elements are computed until the stream is consumed through a terminal operation.

Example 1: Using `Stream.iterate`

Generate an infinite sequence of natural numbers starting at 1:

```
import java.util.stream.Stream;

public class InfiniteStreamIterate {
    public static void main(String[] args) {
        Stream<Integer> naturalNumbers = Stream.iterate(1, n -> n + 1);

        // Limit to first 10 elements to avoid infinite processing
        naturalNumbers.limit(10)
                      .forEach(System.out::println);
    }
}
```

Output:

```
1
```

```
2
3
4
5
6
7
8
9
10
```

Here, `iterate` builds an infinite stream, but `limit(10)` safely restricts the output to the first 10 elements, preventing unbounded computation.

Example 2: Using `Stream.generate`

Generate an infinite stream of random numbers:

```java
import java.util.Random;
import java.util.stream.Stream;

public class InfiniteStreamGenerate {
    public static void main(String[] args) {
        Random random = new Random();

        Stream<Double> randomNumbers = Stream.generate(random::nextDouble);

        randomNumbers.limit(5)
                     .forEach(System.out::println);
    }
}
```

This creates an endless stream of random doubles, but only 5 are printed due to `limit`.

Why Laziness is Crucial

Without **lazy evaluation**, infinite streams would cause immediate non-termination or out-of-memory errors, since all elements would be computed at once. Java streams defer element generation until a terminal operation requests them, enabling infinite streams to be practical and safe.

Common Use Cases for Infinite Streams

- **Generating numeric sequences:** Natural numbers, Fibonacci numbers, primes, etc.
- **Random or pseudo-random data:** For simulations or testing.
- **On-demand data processing:** Event streams, sensor data, or user input streams.
- **Reactive and asynchronous programming:** Handling potentially unbounded event flows elegantly.

10.2.1 Summary

Java's Streams API makes infinite streams accessible via `Stream.iterate` and `Stream.generate`. Thanks to lazy evaluation, these streams produce elements only as needed, allowing safe processing with short-circuiting operations like `limit()`. Infinite streams empower powerful functional programming patterns to model unbounded or on-demand data sources flexibly and efficiently.

10.3 Example: Generating Fibonacci Numbers Lazily

The Fibonacci sequence is a classic example of an infinite numeric series where each number is the sum of the two preceding ones, starting from 0 and 1:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Using Java's **infinite streams**, we can generate this sequence **lazily**—producing values only when requested, avoiding unnecessary computation or memory usage.

Lazy Fibonacci Stream with `Stream.iterate`

Here's a runnable example generating Fibonacci numbers lazily:

```
import java.util.stream.Stream;

public class LazyFibonacci {

    public static void main(String[] args) {
        // Stream.iterate takes a seed (initial pair) and a function to produce the next pair
        Stream<long[]> fibStream = Stream.iterate(
            new long[]{0, 1},                  // Initial pair: fib(0)=0, fib(1)=1
            pair -> new long[]{pair[1], pair[0] + pair[1]}  // Next pair: [fib(n), fib(n+1)]
        );

        // Extract only the first element of each pair (the current Fibonacci number),
        // limit to first 15 numbers to avoid infinite output
        fibStream
            .limit(15)
            .map(pair -> pair[0])
            .forEach(System.out::println);
    }
}
```

Explanation

- The stream's **seed** is a two-element array representing the first two Fibonacci numbers `[0, 1]`.
- The **unary operator** function generates the next pair by shifting the previous pair:

  - `pair[1]` becomes the new first element.
  - `pair[0] + pair[1]` becomes the new second element.
- This way, each step produces a pair representing consecutive Fibonacci numbers.
- The stream is infinite because `Stream.iterate` will keep generating pairs indefinitely.
- We use `.limit(15)` to safely restrict output to the first 15 Fibonacci numbers.
- `.map(pair -> pair[0])` extracts the current Fibonacci number from each pair.

Output

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

10.3.1 Key Takeaways

- The Fibonacci sequence is modeled as a stream of pairs to maintain state without mutable variables.
- **Lazy evaluation** means the Fibonacci numbers are calculated only when requested, one by one.
- This approach scales to very large sequences efficiently because it never computes or stores more elements than necessary.
- Using `.limit()` protects against infinite looping or excessive memory use, demonstrating safe consumption of infinite streams.

This pattern illustrates the power of Java streams combined with functional programming concepts to elegantly and efficiently generate infinite sequences on demand.

Chapter 11.

Monads and Functional Data Structures

1. Understanding Monads in Java Context
2. Using Optional as a Monad
3. Introduction to Functional Collections
4. Example: Chaining Operations Safely with Monads

11 Monads and Functional Data Structures

11.1 Understanding Monads in Java Context

Monads are an important concept in functional programming that help us **manage side effects** and **chain computations** cleanly and safely. Although monads originated in languages like Haskell, their principles can be understood and applied in Java, even without native monad syntax.

What is a Monad?

At its core, a **monad** is a design pattern that wraps a value and provides a way to:

- **Chain operations** on that value without unwrapping it manually each time.
- **Handle side effects or computations that may fail, be absent, or produce multiple results.**

Think of a monad as a container with some value inside, along with rules for how to apply functions to that value while preserving the container's structure and context.

Why Are Monads Important?

In traditional programming, handling operations that might fail (e.g., null values, exceptions) or have side effects often results in scattered checks or try-catch blocks, making code verbose and error-prone.

Monads **encapsulate** these concerns:

- They allow **safe chaining** of operations, automatically managing cases like absence or failure.
- They keep the code **clean and expressive** by abstracting boilerplate logic.
- They encourage **composability** of functions, making programs modular and easier to reason about.

Monad Components in Java

In Java, you can think of a monad as any class that provides:

1. **A way to wrap a value** — typically a static factory or constructor. For example, `Optional.of(value)` wraps a value into an `Optional`.
2. **A method to apply a function and flatten the result** — commonly called `flatMap` or `bind`, which lets you chain operations that themselves return wrapped values.

`Optional`, `Stream`, and even `CompletableFuture` follow this monadic pattern.

Monad Laws (Brief Overview)

To qualify as a monad, a type must follow three laws ensuring predictable behavior:

1. **Left identity:** Wrapping a value and then applying a function is the same as just applying the function.

2. **Right identity:** Applying a wrapping function to a monad doesn't change the monad.
3. **Associativity:** Chaining multiple functions yields the same result regardless of how operations are nested.

These laws guarantee **consistency** in chaining operations and help prevent subtle bugs.

Example: Using `Optional` as a Monad

Consider chaining methods to extract and transform nested data safely:

```
Optional<String> name = Optional.of("Alice");

Optional<String> result = name
    .flatMap(n -> Optional.of(n.toUpperCase()))
    .flatMap(n -> Optional.of(n + " Smith"));

result.ifPresent(System.out::println);  // Output: ALICE Smith
```

Here, `flatMap` lets you chain transformations without worrying about nulls or wrapping/unwrapping values explicitly. If any step returns an empty `Optional`, the entire chain short-circuits safely.

11.1.1 Summary

- A **monad** is a pattern for wrapping values and chaining computations while handling context like failure, absence, or side effects.
- Java doesn't have native monad syntax but supports monadic programming through classes like `Optional` and `Stream`.
- Monads follow laws that ensure their chaining behavior is predictable and consistent.
- Using monads helps you write safer, cleaner, and more modular code by abstracting error handling and side effects from business logic.

Understanding monads equips you with a powerful tool to structure functional programs effectively in Java's ecosystem.

11.2 Using Optional as a Monad

Java's `Optional` is more than just a null-avoidance utility—it also exhibits **monadic behavior**. In functional programming, a **monad** is a pattern that wraps values and provides a consistent way to **chain operations** while handling effects like absence, failure, or context. `Optional` achieves this by safely encapsulating a potentially absent value and offering methods like `map()` and `flatMap()` to transform or chain further operations.

Why Treat `Optional` as a Monad?

Traditionally, dealing with `null` in Java required verbose conditional checks:

```
String name = getUser();
if (name != null) {
    String upper = name.toUpperCase();
    if (upper != null) {
        // Do something
    }
}
```

This kind of nesting is error-prone and hard to maintain. `Optional` helps you **eliminate nested conditionals** and make data transformations **composable**.

Chaining with `map` and `flatMap`

- `map(Function<T, R>)` transforms the wrapped value if present, returning a new `Optional<R>`.
- `flatMap(Function<T, Optional<R>>)` is used when the function itself returns an `Optional`, avoiding nested optionals.

Runnable Example: Composing Optional Operations

```
import java.util.Optional;

public class OptionalMonadExample {

    public static void main(String[] args) {
        Optional<String> username = Optional.of("alice");

        Optional<String> result = username
            .map(String::toUpperCase)                    // Optional["ALICE"]
            .flatMap(OptionalMonadExample::addLastName)  // Optional["ALICE SMITH"]
            .map(OptionalMonadExample::wrapInBrackets);  // Optional["[ALICE SMITH]"]

        result.ifPresent(System.out::println); // Output: [ALICE SMITH]
    }

    // Simulates a function that returns an Optional
    static Optional<String> addLastName(String name) {
        return Optional.of(name + " SMITH");
    }

    // Pure function that wraps a string
    static String wrapInBrackets(String input) {
        return "[" + input + "]";
    }
}
```

What Happens Under the Hood

- If any stage returns `Optional.empty()`, the entire chain short-circuits.
- You avoid `null` checks entirely, focusing only on the transformations.

- `flatMap()` prevents nested `Optional<Optional<T>>`, keeping the chain clean.

Benefits Over Traditional Null Checks

Traditional Approach	Optional Monad Approach
Verbose and repetitive	Concise and readable
Easy to forget null checks	Forces explicit handling of absence
Not composable	Easily composable with functions

11.2.1 Summary

Java's `Optional` fits the monad model by wrapping a value and providing `map()` and `flatMap()` for transformation and chaining. This allows developers to build robust and expressive pipelines that handle missing values safely—without the clutter of null checks. By treating `Optional` as a monad, your code becomes more **declarative**, **composable**, and **error-resistant**, embracing the functional programming style within the Java ecosystem.

11.3 Introduction to Functional Collections

In functional programming, **immutability** is a core principle. Data structures are not modified in place; instead, operations produce new versions of data with the desired changes. This leads to **predictable, thread-safe**, and **side-effect-free** code. Traditional Java collections like `ArrayList`, `HashMap`, and `HashSet` are **mutable**, which can conflict with functional paradigms that emphasize **referential transparency** and **pure functions**.

Why Mutable Collections Are Problematic

Mutable collections can introduce bugs, especially in concurrent or multi-threaded applications. For example, modifying a shared list in one function might unintentionally affect another function relying on the original state. This violates **functional purity**, where the output of a function should depend only on its inputs, not external mutable state.

Example of problematic code:

```
List<String> names = new ArrayList<>();
names.add("Alice");
modifyList(names); // might add/remove elements
```

You can't be sure what `names` contains after `modifyList()`—this unpredictability undermines code clarity and safety.

Functional Collections: Immutable by Design

Functional (or persistent) collections solve this by ensuring data structures cannot be altered once created. Instead of modifying an existing structure, you create a **new version** with the desired change, while sharing structure internally for efficiency.

These collections enable:

- **Thread safety** without locks.
- **Simplified reasoning** about program state.
- **No accidental side effects.**

Java Support and Third-Party Libraries

Java does not provide full persistent collections natively, but some options are available:

- **Java 9+ `List.of()`, `Set.of()`, `Map.of()`** — Immutable collections, but not persistent.
- **Guava (`ImmutableList`, `ImmutableMap`)** — Popular for read-only collections.
- **Vavr** — A functional programming library for Java with persistent collections (`List`, `Map`, `Set`, etc.).
- **FunctionalJava** and **Javaslang** (Vavr's predecessor) — Offer rich FP data types.

Example: Using Vavr Immutable List

```
import io.vavr.collection.List;

public class FunctionalListExample {
    public static void main(String[] args) {
        List<String> original = List.of("Java", "Scala", "Kotlin");

        // Create a new list with an added element
        List<String> updated = original.append("Clojure");

        System.out.println("Original: " + original); // [Java, Scala, Kotlin]
        System.out.println("Updated: " + updated);   // [Java, Scala, Kotlin, Clojure]
    }
}
```

Here, `original` remains unchanged—`updated` is a new list. This ensures **safe reuse of values** in concurrent or chained functional logic.

11.3.1 Summary

Functional collections uphold the principles of **immutability, purity, and referential transparency**. While standard Java collections are mutable by default, libraries like Vavr bring robust, persistent alternatives into the language. Adopting functional collections leads to **clearer, safer, and more maintainable** Java code, especially when writing programs in

a functional style.

11.4 Example: Chaining Operations Safely with Monads

11.4.1 Example: Chaining Operations Safely with Monads

Monads provide a consistent, safe way to chain computations, especially when dealing with operations that may fail or produce absent values. In Java, the `Optional` class acts as a monad, allowing developers to **compose operations** on values that might be missing—without resorting to verbose null checks or nested conditionals.

Let's walk through a practical example that demonstrates how to chain multiple operations safely using `Optional`.

Problem

We want to extract a user's ZIP code from a nested object structure. The user might not have an address, and the address might not have a ZIP code. Traditionally, this requires multiple null checks.

Traditional Approach (Verbose & Error-Prone)

```
String zip = null;
if (user != null) {
    Address addr = user.getAddress();
    if (addr != null) {
        zip = addr.getZipcode();
    }
}
```

Monadic Approach Using `Optional`

```
import java.util.Optional;

public class MonadChainingExample {

    // Nested domain classes
    static class User {
        private final Optional<Address> address;

        User(Address address) {
            this.address = Optional.ofNullable(address);
        }

        Optional<Address> getAddress() {
            return address;
        }
    }
```

```
    static class Address {
        private final Optional<String> zipcode;

        Address(String zipcode) {
            this.zipcode = Optional.ofNullable(zipcode);
        }

        Optional<String> getZipcode() {
            return zipcode;
        }
    }

    public static void main(String[] args) {
        // Sample data
        User userWithZip = new User(new Address("12345"));
        User userWithoutZip = new User(new Address(null));
        User userWithoutAddress = new User(null);

        // Chain operations using flatMap to avoid nested optionals
        System.out.println(getUserZip(userWithZip));     // Output: Optional[12345]
        System.out.println(getUserZip(userWithoutZip));  // Output: Optional.empty
        System.out.println(getUserZip(userWithoutAddress)); // Output: Optional.empty
    }

    // Chaining safely using monadic composition
    static Optional<String> getUserZip(User user) {
        return Optional.ofNullable(user)
                       .flatMap(User::getAddress)
                       .flatMap(Address::getZipcode);
    }
}
```

11.4.2 Explanation

- Each method (`getAddress`, `getZipcode`) returns an `Optional`, making it composable.
- `flatMap` is used instead of `map` to avoid nested `Optional<Optional<T>>`.
- If at any stage the data is absent (`null`), the chain safely short-circuits and returns `Optional.empty()`.

11.4.3 Benefits

- **No null checks**: Eliminates nested if-statements.
- **Readable and maintainable**: Clear intent and logic flow.
- **Safe by design**: Nulls are handled via the type system.

This example shows how treating `Optional` as a monad enhances safety, robustness, and clarity when navigating complex or uncertain data structures.

Chapter 12.

Concurrency and Functional Programming

1. Using CompletableFuture with Functional Style
2. Parallel Stream Pitfalls and Best Practices
3. Reactive Programming Overview with Functional Concepts
4. Example: Asynchronous Data Fetching

12 Concurrency and Functional Programming

12.1 Using CompletableFuture with Functional Style

In modern Java, the `CompletableFuture` class is a powerful tool for writing **asynchronous** and **non-blocking** programs. It allows tasks to run in the background and supports chaining operations without blocking the main thread. Combined with **lambdas** and **functional interfaces**, `CompletableFuture` enables a **declarative, functional style** of concurrency that is clean and readable.

Why Use `CompletableFuture`?

Before `CompletableFuture`, writing asynchronous code often involved manual thread management or callback hell using nested anonymous classes. `CompletableFuture` simplifies this by:

- Letting you write non-blocking chains of computations.
- Providing a fluent API for composing dependent tasks.
- Handling exceptions in a unified, composable way.

12.1.1 Functional Methods in `CompletableFuture`

Here are some core methods that enable a functional style:

- **`thenApply(Function<T, R>)`**: Transforms the result of a computation.
- **`thenCompose(Function<T, CompletableFuture<R>>)`**: Flattens nested futures, used for chaining dependent asynchronous calls.
- **`exceptionally(Function<Throwable, T>)`**: Handles exceptions and provides fallback values.
- **`thenAccept(Consumer<T>)`**: Consumes the result without returning anything.

These methods accept **functional interfaces**, which means you can pass **lambdas** or **method references** directly.

12.1.2 Example: Building a Functional Asynchronous Workflow

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;

public class FunctionalCompletableFuture {

    public static void main(String[] args) {
        CompletableFuture<String> future = fetchUserId()
```

```
            .thenCompose(FunctionalCompletableFuture::fetchUserDetails)     // flatMap equivalent
            .thenApply(data -> "User Info: " + data)                        // map equivalent
            .exceptionally(ex -> "Error: " + ex.getMessage());

        // Block to print the result (not recommended in production)
        System.out.println(future.join());
    }

    // Simulate an async method to get user ID
    static CompletableFuture<String> fetchUserId() {
        return CompletableFuture.supplyAsync(() -> {
            sleep(500);
            return "user123";
        });
    }

    // Simulate an async method to fetch user details using the ID
    static CompletableFuture<String> fetchUserDetails(String userId) {
        return CompletableFuture.supplyAsync(() -> {
            sleep(1000);
            if (userId.equals("user123")) {
                return "Name: Alice, Age: 30";
            } else {
                throw new RuntimeException("User not found");
            }
        });
    }

    // Utility sleep method
    static void sleep(int millis) {
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
        } catch (InterruptedException ignored) {}
    }
}
```

12.1.3 Explanation

- `fetchUserId()` returns a `CompletableFuture<String>`.
- `thenCompose(fetchUserDetails)` starts the second async call using the result from the first.
- `thenApply(...)` formats the final result.
- `exceptionally(...)` provides error handling without try-catch blocks.

The entire pipeline is non-blocking until `.join()` is called at the end to retrieve the result (typically avoided in production where callbacks or further chaining would be used instead).

12.1.4 Benefits of Functional Style

- **Declarative and readable**: Each transformation is explicit and logically ordered.
- **Non-blocking**: Async operations don't tie up threads unnecessarily.
- **Composable**: You can build complex workflows by chaining simple operations.
- **Error handling**: Centralized and clean, reducing scattered try-catch code.

12.1.5 Summary

`CompletableFuture` enables a **functional approach to concurrency** in Java. By using methods like `thenApply`, `thenCompose`, and `exceptionally`, developers can construct asynchronous pipelines that are efficient, readable, and robust. Embracing these functional patterns leads to more maintainable and scalable concurrent applications.

12.2 Parallel Stream Pitfalls and Best Practices

Java's **parallel streams** offer an easy way to perform data processing in parallel, potentially improving performance on multi-core systems. By invoking `.parallelStream()` or calling `.parallel()` on a stream, Java automatically handles thread distribution. However, parallel streams come with **pitfalls** that can lead to **incorrect results**, **unpredictable behavior**, or even **worse performance** than sequential streams when misused.

12.2.1 Common Pitfalls of Parallel Streams

Shared Mutable State

One of the most dangerous issues with parallel streams is using shared mutable state without proper synchronization.

Pitfall Example (unsafe):

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;

public class SharedStatePitfall {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();

        IntStream.range(0, 1000).parallel().forEach(list::add); // Not thread-safe!

        System.out.println("Size: " + list.size()); // Often < 1000
```

```
    }
}
```

This may print a size smaller than 1000 because `ArrayList` is not thread-safe. Concurrent modifications lead to data races and corrupted state.

Solution: Use thread-safe collectors or concurrent data structures.

```
List<Integer> list = IntStream.range(0, 1000)
    .parallel()
    .boxed()
    .collect(Collectors.toList()); // Internally uses thread-safe collection
```

Side Effects in Stream Operations

Functional programming discourages side effects, and they're especially problematic in parallel streams where the order and timing of execution are unpredictable.

Avoid code like this:

```
parallelStream.forEach(x -> doSomethingAndLog(x)); // Logging may interleave
```

Side effects (e.g., logging, I/O) can interfere with concurrency and are hard to debug.

Best Practice: Make operations pure—no external effects or shared state.

Unexpected Ordering

Parallel streams do not guarantee the order of execution unless you explicitly preserve it.

Example:

```
List.of("A", "B", "C", "D")
    .parallelStream()
    .forEach(System.out::print); // Output could be: CBAD, DCBA, etc.
```

Solution: Use `.forEachOrdered()` if order matters:

```
.parallelStream()
.forEachOrdered(System.out::print);
```

Poor Fit for Small or Simple Workloads

Parallelism introduces overhead. For small data sets or inexpensive operations, it can actually reduce performance.

Best Practice: Use parallel streams only for:

- CPU-intensive tasks.
- Large data sets (e.g., 10,000+ elements).
- Stateless, associative operations.

12.2.2 Best Practices for Using Parallel Streams

- YES Use **pure functions** with no side effects.
- YES Use **thread-safe collectors**, such as `Collectors.toList()`.
- YES Avoid accessing **shared mutable state**.
- YES Benchmark performance with real workloads—don't assume faster = better.
- YES Use `.forEachOrdered()` if output order matters.
- YES Use parallel streams in **read-heavy, CPU-bound** workloads.

12.2.3 Summary

Parallel streams provide a convenient abstraction for concurrent data processing, but they come with significant caveats. Misusing them can lead to **bugs**, **race conditions**, and **worse performance**. By adhering to functional principles—**immutability**, **statelessness**, and **thread-safety**—you can safely harness the power of parallelism in Java streams.

12.3 Reactive Programming Overview with Functional Concepts

Reactive programming is a paradigm built on the foundation of **functional programming**, designed for **asynchronous, non-blocking** handling of data streams. It focuses on **responding to events over time**—whether those events come from user actions, network responses, or sensor input—and handling them in a declarative, efficient, and resilient way.

12.3.1 Core Concepts in Reactive Programming

1. **Asynchronous Data Streams** Reactive systems model data as **streams** of events that can be observed and transformed. Instead of pulling data when needed, you **subscribe** to a stream and receive items as they become available. These streams can be finite (e.g., a file) or infinite (e.g., user input, network sockets).
2. **Observables and Observers** At the heart of reactive systems is the **observer pattern**, where an **observable** emits items and **observers** subscribe to receive them. This aligns with functional concepts such as higher-order functions, where callbacks (functions) are passed to handle emitted values.
3. **Backpressure** In real systems, data producers can be faster than consumers. **Backpressure** is a mechanism that allows subscribers to signal how much data they can handle, avoiding memory overload and crashes. It's a key concept in resilient reactive design.

4. **Event-Driven Architecture** Reactive applications are often **event-driven**, where business logic responds to asynchronous triggers like clicks, API results, or system changes. This decouples components, improves scalability, and fits naturally with lambda-based functional patterns.

12.3.2 Functional Principles in Reactive Programming

Reactive programming deeply embraces **functional principles**:

- **Immutability**: Each transformation on a stream results in a new stream without modifying the original.
- **Pure Functions**: Stream transformations avoid side effects, making behavior predictable and testable.
- **Higher-Order Functions**: Methods like `map`, `filter`, `flatMap`, and `reduce` are core to reactive pipelines, letting you express data flow clearly and declaratively.

These features resemble Java's Stream API, but reactive streams are **asynchronous and push-based**, whereas Java streams are **synchronous and pull-based**.

12.3.3 Reactive Libraries in Java

Several mature libraries bring reactive programming to Java:

- **RxJava** (from Netflix): One of the earliest and most feature-rich libraries, built around the `Observable` type and offering a wide range of operators for transforming and composing streams.
- **Project Reactor** (by Pivotal): The reactive foundation for Spring WebFlux, providing types like `Mono` (0–1 values) and `Flux` (0–N values) that support backpressure and functional chaining.
- **Akka Streams** and **Mutiny** are other libraries used in more specialized reactive systems.

12.3.4 Summary

Reactive programming is a **natural extension of functional programming** for handling asynchronous data and events. By leveraging **observables**, **backpressure**, and **declarative transformation operators**, it promotes clean, responsive, and resilient code. Java developers can adopt reactive paradigms through libraries like RxJava or Reactor, building on familiar functional concepts like lambdas, higher-order functions, and immutable streams.

12.4 Example: Asynchronous Data Fetching

Asynchronous data fetching is a common scenario in modern applications—retrieving information from a database, remote API, or external service. Java's `CompletableFuture` provides a clean, functional approach to handle this using **non-blocking**, **composable** operations. This section presents a **self-contained example** of composing asynchronous calls, handling errors, and applying transformations in a functional style.

12.4.1 Scenario

We want to simulate a process that:

1. Fetches a user ID asynchronously.
2. Uses that ID to fetch user details (like name and email).
3. Formats and returns the result.
4. Handles any potential errors gracefully.

12.4.2 Code Example (Using `CompletableFuture`)

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ThreadLocalRandom;

public class AsyncDataFetchExample {

    public static void main(String[] args) {
        fetchUserId()
            .thenCompose(AsyncDataFetchExample::fetchUserDetails) // Chain async fetch
            .thenApply(user -> "Fetched User: " + user)           // Format result
            .exceptionally(ex -> "Error occurred: " + ex.getMessage()) // Handle errors
            .thenAccept(System.out::println);                     // Print result

        // Prevent main thread from exiting early
        sleep(2000);
    }

    // Simulates asynchronous fetching of a user ID
    static CompletableFuture<String> fetchUserId() {
        return CompletableFuture.supplyAsync(() -> {
            sleep(500); // Simulate delay
            return "user123";
        });
    }

    // Simulates asynchronous fetching of user details using user ID
    static CompletableFuture<String> fetchUserDetails(String userId) {
```

```
        return CompletableFuture.supplyAsync(() -> {
            sleep(800); // Simulate delay
            if (ThreadLocalRandom.current().nextBoolean()) {
                return "Name: Alice, Email: alice@example.com";
            } else {
                throw new RuntimeException("Failed to fetch user details");
            }
        });
    }

    // Utility method to sleep without exception handling noise
    static void sleep(int millis) {
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
        } catch (InterruptedException ignored) {}
    }
}
```

12.4.3 Explanation

- **`fetchUserId()`** returns a `CompletableFuture<String>` simulating a remote call.
- **`thenCompose()`** is used to **chain another asynchronous call** based on the result of the first.
- **`thenApply()`** transforms the result without blocking.
- **`exceptionally()`** handles any error in the pipeline.
- **`thenAccept()`** consumes the final result without returning a value.

12.4.4 Benefits of Functional Asynchronous Composition

- **Non-blocking**: The main thread isn't blocked waiting for responses.
- **Composable**: Each step is expressed as a transformation or continuation.
- **Readable**: The linear flow avoids nested callbacks or complex error handling.
- **Robust**: Errors are handled declaratively with fallback logic.

This example demonstrates how to use Java's `CompletableFuture` in a clean, functional style for real-world asynchronous tasks.

Chapter 13.

Functional Programming in Collections and APIs

1. Using Streams for Collection Manipulation
2. Functional Interfaces in Java APIs (e.g., `Comparator`, `Runnable`)
3. Example: Sorting and Grouping Complex Data

13 Functional Programming in Collections and APIs

13.1 Using Streams for Collection Manipulation

Java's **Streams API**, introduced in Java 8, revolutionized how developers interact with collections by enabling **declarative**, **functional-style** data processing. Instead of writing verbose `for`-loops or mutating data imperatively, you can now express operations like filtering, transforming, sorting, and aggregating in a clean, composable way.

13.1.1 Why Streams?

Traditional loops operate **eagerly**, mix logic and iteration, and often mutate intermediate state. In contrast, the Streams API:

- Treats data as a **flow of elements** through transformations.
- Promotes **immutability**, reducing bugs caused by shared state.
- Encourages **composition**, where operations like `filter`, `map`, and `collect` are chained fluently.
- Employs **lazy evaluation**, meaning operations are only performed when necessary.

13.1.2 Core Stream Operations

Let's explore the most common operations with examples:

Filtering and Mapping

```
import java.util.List;
import java.util.stream.Collectors;

public class StreamBasics {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie", "David");

        List<String> filtered = names.stream()
            .filter(name -> name.length() > 3)
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println(filtered); // [ALICE, CHARLIE, DAVID]
    }
}
```

- `filter` removes elements based on a condition.
- `map` transforms each element (in this case, to uppercase).

- `collect` gathers the result into a new list (immutably).

Sorting and Distinct

```
List<Integer> numbers = List.of(5, 3, 4, 3, 1, 2);

List<Integer> sorted = numbers.stream()
    .distinct()
    .sorted()
    .collect(Collectors.toList());

System.out.println(sorted); // [1, 2, 3, 4, 5]
```

- `distinct` removes duplicates.
- `sorted` orders the elements naturally or via a comparator.

Grouping and Partitioning

```
import java.util.Map;
import java.util.stream.Collectors;

List<String> animals = List.of("cat", "cow", "dog", "dolphin");

Map<Character, List<String>> grouped = animals.stream()
    .collect(Collectors.groupingBy(name -> name.charAt(0)));

System.out.println(grouped); // {c=[cat, cow], d=[dog, dolphin]}
```

- `groupingBy` groups elements by a classifier function.

```
Map<Boolean, List<String>> partitioned = animals.stream()
    .collect(Collectors.partitioningBy(name -> name.length() > 3));

System.out.println(partitioned); // {false=[cat, cow, dog], true=[dolphin]}
```

- `partitioningBy` splits the stream into two groups based on a predicate.

13.1.3 Benefits of Functional Collection Processing

- **Readability**: Code is concise and intention-revealing.
- **Immutability**: Intermediate operations do not alter source data.
- **Lazy Evaluation**: Operations are only executed when the terminal operation is invoked (like `collect`, `forEach`, `count`).
- **Parallelism**: Streams can be easily parallelized using `.parallelStream()` when needed.

13.1.4 Summary

The Streams API enables expressive, functional-style manipulation of Java collections. By chaining operations like `map`, `filter`, and `collect`, developers can write cleaner, safer, and more maintainable code. Whether filtering lists, sorting elements, or grouping by criteria, streams provide a declarative approach that fits naturally with modern functional programming techniques in Java.

13.2 Functional Interfaces in Java APIs (e.g., `Comparator`, `Runnable`)

While Java 8 introduced the `java.util.function` package with core functional interfaces like `Function`, `Predicate`, and `Consumer`, several **long-standing interfaces** in Java's standard library are also **functional** by design. These interfaces typically define a **single abstract method**, making them ideal targets for **lambda expressions** and **method references**.

Let's explore some widely used functional interfaces already present in common Java APIs.

13.2.1 `Comparator<T>` Functional Sorting

The `Comparator<T>` interface is used to define custom ordering logic for objects. Its single abstract method is:

```
int compare(T o1, T o2);
```

Example: Sorting a list of strings by length

```
import java.util.*;

public class ComparatorExample {
    public static void main(String[] args) {
        List<String> names = List.of("Alice", "Bob", "Charlie");

        names.stream()
            .sorted(Comparator.comparingInt(String::length))
            .forEach(System.out::println); // Bob, Alice, Charlie
    }
}
```

The use of `Comparator.comparingInt` with a method reference makes the sort logic declarative and functional.

13.2.2 `Runnable` Deferred Execution

`Runnable` represents a task to run, typically in a separate thread. Its abstract method:

```
void run();
```

Example: Executing a task in a new thread

```
public class RunnableExample {
    public static void main(String[] args) {
        Runnable task = () -> System.out.println("Running task...");
        new Thread(task).start();
    }
}
```

Using a lambda for `Runnable` avoids boilerplate like anonymous classes, improving readability.

13.2.3 `Callable<V>` Asynchronous Computation with Return

`Callable<V>` is similar to `Runnable` but returns a result and may throw an exception:

```
V call() throws Exception;
```

Example: Executing a computation with `ExecutorService`

```
import java.util.concurrent.*;

public class CallableExample {
    public static void main(String[] args) throws Exception {
        Callable<String> task = () -> "Result from callable";

        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(task);

        System.out.println(future.get()); // Result from callable
        executor.shutdown();
    }
}
```

13.2.4 `Supplier<T>` Deferred Value Generation

`Supplier<T>` is used to generate or provide a value on demand:

```
T get();
```

Example: Lazy initialization

```java
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomSupplier = () -> Math.random();
        System.out.println(randomSupplier.get()); // e.g., 0.5834...
    }
}
```

13.2.5 Summary

Java's built-in functional interfaces like `Comparator`, `Runnable`, `Callable`, and `Supplier` support **functional programming idioms** across many APIs. Thanks to **lambda expressions**, these interfaces can be used more succinctly than ever, enabling **cleaner**, **more expressive**, and **less error-prone** code—whether you're sorting data, executing tasks, or providing values on demand.

13.3 Example: Sorting and Grouping Complex Data

Sorting and grouping complex objects like employees by attributes such as department or salary range is a common task in real-world applications. Using Java Streams and functional programming techniques, we can express this logic **declaratively**, resulting in cleaner and more maintainable code.

Below is a **runnable example** that demonstrates:

- Sorting employees by department and descending salary
- Grouping employees by department
- Partitioning employees by high/low salary threshold
- Using functional interfaces like `Comparator` and `Collectors.groupingBy`

13.3.1 Code Example

```java
import java.util.*;
import java.util.stream.*;

public class EmployeeProcessing {
    static class Employee {
        String name;
        String department;
        double salary;
```

```
        Employee(String name, String department, double salary) {
            this.name = name;
            this.department = department;
            this.salary = salary;
        }

        @Override
        public String toString() {
            return name + " (" + department + ", $" + salary + ")";
        }
    }

    public static void main(String[] args) {
        List<Employee> employees = List.of(
            new Employee("Alice", "HR", 48000),
            new Employee("Bob", "Engineering", 75000),
            new Employee("Charlie", "Engineering", 82000),
            new Employee("Diana", "HR", 52000),
            new Employee("Evan", "Marketing", 60000),
            new Employee("Fiona", "Engineering", 69000)
        );

        // 1. Sort by department, then by salary descending
        List<Employee> sorted = employees.stream()
            .sorted(Comparator.comparing((Employee e) -> e.department)
                    .thenComparing(Comparator.comparingDouble((Employee e) -> e.salary).reversed()))
            .collect(Collectors.toList());

        System.out.println("Sorted Employees:");
        sorted.forEach(System.out::println);

        // 2. Group by department
        Map<String, List<Employee>> groupedByDept = employees.stream()
            .collect(Collectors.groupingBy(e -> e.department));

        System.out.println("\nGrouped by Department:");
        groupedByDept.forEach((dept, list) -> {
            System.out.println(dept + ": " + list);
        });

        // 3. Partition by salary > 70000
        Map<Boolean, List<Employee>> highEarners = employees.stream()
            .collect(Collectors.partitioningBy(e -> e.salary > 70000));

        System.out.println("\nPartitioned by Salary > 70000:");
        highEarners.forEach((isHigh, list) -> {
            System.out.println((isHigh ? "High Earners" : "Others") + ": " + list);
        });
    }
}
```

13.3.2 Explanation

- **Sorting**: We chain comparators using `thenComparing`, sorting by department (ascending) and then salary (descending).
- **Grouping**: `Collectors.groupingBy()` organizes employees into lists keyed by department.
- **Partitioning**: `Collectors.partitioningBy()` separates employees into two groups based on a salary threshold.

13.3.3 Benefits of Functional Style

- **Declarative**: No manual loops or conditionals—intent is expressed clearly.
- **Composable**: Operations like `sorted`, `groupingBy`, and `partitioningBy` work seamlessly together.
- **Maintainable**: Adding new rules (e.g., sort by name) requires minimal code changes.

This approach demonstrates how Java's functional features simplify complex data manipulation tasks.

Chapter 14.

Functional Programming in Data Processing

1. Processing Files and I/O Functionally
2. Parsing and Transforming JSON Data
3. Example: Building a CSV Parser with Streams

14 Functional Programming in Data Processing

14.1 Processing Files and I/O Functionally

Java's **Streams API** and **functional interfaces** offer powerful tools to process files and I/O data in a clean, declarative way. Instead of writing complex loops and managing resources manually, you can leverage **stream pipelines** to read, transform, and aggregate file content efficiently and safely.

14.1.1 Reading Files with `Files.lines()`

The `Files.lines(Path)` method returns a **lazy stream of lines** from a file. This allows processing large files without loading the entire content into memory at once.

14.1.2 Example: Reading and Processing a Text File

Suppose we want to process a text file to:

- Read all lines,
- Filter lines containing a specific keyword,
- Convert lines to uppercase,
- Count the number of matching lines.

Here's how this can be done functionally:

```
import java.nio.file.*;
import java.io.IOException;
import java.util.stream.Stream;

public class FileProcessingExample {

    public static void main(String[] args) {
        Path filePath = Paths.get("example.txt");
        String keyword = "java";

        // Use try-with-resources for automatic resource management
        try (Stream<String> lines = Files.lines(filePath)) {
            long count = lines
                .filter(line -> line.toLowerCase().contains(keyword))
                .map(String::toUpperCase)
                .peek(System.out::println)  // Print each matching line
                .count();

            System.out.println("Total lines containing '" + keyword + "': " + count);
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
```

```
        }
    }
}
```

14.1.3 Key Points

- **Lazy evaluation**: The file is read line-by-line as the stream pipeline executes, making it memory-efficient.
- **Declarative transformation**: Operations like `filter` and `map` describe *what* to do rather than *how*.
- **Resource safety**: `try-with-resources` ensures the file stream is closed automatically.
- **Exception handling**: Checked exceptions from I/O are handled cleanly with a `try-catch` block outside the stream.

14.1.4 Handling Exceptions Functionally

Since Java streams do not natively support checked exceptions in lambdas, you can either:

- Wrap checked exceptions in runtime exceptions within lambdas, or
- Use helper methods to handle exceptions functionally (not covered here for brevity).

14.1.5 Aggregating and Collecting

Streams allow aggregating data easily:

```
// Example: Count occurrences of words starting with "f"
try (Stream<String> lines = Files.lines(filePath)) {
    long count = lines
        .flatMap(line -> Stream.of(line.split("\\s+")))
        .filter(word -> word.startsWith("f"))
        .count();

    System.out.println("Words starting with 'f': " + count);
}
```

14.1.6 Summary

By combining Java's Streams API with file I/O, you can write concise, readable, and efficient code for processing text files. The **functional approach** encourages immutability, lazy evaluation, and easy composition of operations, while **try-with-resources** handles resource management seamlessly. This leads to safer, more maintainable file-processing code.

14.2 Parsing and Transforming JSON Data

Parsing and transforming JSON data is a common task in modern Java applications, especially when dealing with APIs or configuration files. While popular libraries like **Jackson** and **Gson** handle JSON serialization and deserialization, you can integrate **functional programming techniques** to process and transform JSON data in a clean, declarative style.

14.2.1 Functional JSON Parsing with Jackson

Jackson is one of the most widely used JSON libraries in Java. It can convert JSON into Java objects (POJOs) or generic structures like `JsonNode`. Combined with Java Streams and lambdas, you can process JSON collections and transform data efficiently.

14.2.2 Example: Mapping JSON Array to Objects and Transforming

Suppose you have a JSON array of users, and you want to parse it, filter users by age, and transform their names to uppercase.

```
import com.fasterxml.jackson.databind.*;
import java.util.*;
import java.util.stream.*;

public class JsonFunctionalExample {
    public static void main(String[] args) throws Exception {
        String json = """
            [
                {"name": "Alice", "age": 30},
                {"name": "Bob", "age": 22},
                {"name": "Charlie", "age": 25}
            ]
            """;

        ObjectMapper mapper = new ObjectMapper();

        // Deserialize JSON array into List<User>
```

```
        List<User> users = Arrays.asList(mapper.readValue(json, User[].class));

        // Use stream to filter and transform user names
        List<String> names = users.stream()
            .filter(user -> user.age >= 25)
            .map(user -> user.name.toUpperCase())
            .collect(Collectors.toList());

        names.forEach(System.out::println); // ALICE, CHARLIE
    }

    static class User {
        public String name;
        public int age;

        // Default constructor needed by Jackson
        public User() {}
    }
}
```

14.2.3 Functional Transformations on JSON Trees

Jackson also supports a tree model (`JsonNode`), which allows functional traversal and transformations without full binding.

```
JsonNode root = mapper.readTree(json);

List<String> filteredNames = StreamSupport.stream(root.spliterator(), false)
    .filter(node -> node.get("age").asInt() >= 25)
    .map(node -> node.get("name").asText().toUpperCase())
    .collect(Collectors.toList());

filteredNames.forEach(System.out::println); // ALICE, CHARLIE
```

14.2.4 Using Gson with Functional Patterns

Gson can deserialize JSON similarly, allowing you to use streams for post-processing:

```
import com.google.gson.*;
import com.google.gson.reflect.TypeToken;

Gson gson = new Gson();
List<User> users = gson.fromJson(json, new TypeToken<List<User>>(){}.getType());

List<String> names = users.stream()
    .filter(u -> u.age >= 25)
    .map(u -> u.name.toUpperCase())
    .collect(Collectors.toList());
```

14.2.5 Summary

Functional programming enhances JSON parsing and transformation by:

- Mapping JSON arrays directly to Java collections.
- Using streams and lambdas for filtering, mapping, and aggregating.
- Leveraging method references for concise transformations.
- Processing JSON trees with lazy, declarative traversal.

Combining powerful JSON libraries with functional patterns results in clean, expressive, and maintainable data-processing code.

14.3 Example: Building a CSV Parser with Streams

Parsing CSV files is a common task when dealing with tabular data. Using Java Streams and functional interfaces, we can build a clean, concise CSV parser that reads lines from a file, splits fields, maps them to domain objects, filters invalid rows, and collects results—all in a declarative style.

14.3.1 Runnable CSV Parser Example

```
import java.nio.file.*;
import java.io.IOException;
import java.util.*;
import java.util.stream.*;

public class CsvParserExample {

    // Domain class representing a Person
    static class Person {
        String name;
        int age;
        String email;

        Person(String name, int age, String email) {
            this.name = name;
            this.age = age;
            this.email = email;
        }

        @Override
        public String toString() {
            return name + " (" + age + "), " + email;
        }
    }
```

```
    public static void main(String[] args) {
        Path csvFile = Paths.get("people.csv");

        try (Stream<String> lines = Files.lines(csvFile)) {

            List<Person> people = lines
                // Skip header line (assuming first line is headers)
                .skip(1)

                // Split each line by comma into String array
                .map(line -> line.split(","))

                // Filter out invalid lines (e.g., wrong number of fields)
                .filter(fields -> fields.length == 3)

                // Map fields to Person objects, parsing age as int
                .map(fields -> {
                    try {
                        String name = fields[0].trim();
                        int age = Integer.parseInt(fields[1].trim());
                        String email = fields[2].trim();
                        return new Person(name, age, email);
                    } catch (NumberFormatException e) {
                        // Skip lines with invalid age
                        return null;
                    }
                })

                // Filter out nulls from failed parses
                .filter(Objects::nonNull)

                // Collect into a list
                .collect(Collectors.toList());

            people.forEach(System.out::println);

        } catch (IOException e) {
            System.err.println("Failed to read CSV file: " + e.getMessage());
        }
    }
}
```

14.3.2 Sample `people.csv` File

```
name,age,email
Alice,30,alice@example.com
Bob,notanumber,bob@example.com
Charlie,25,charlie@example.com
Dana,40,dana@example.com
```

14.3.3 Explanation

- **Reading lines**: `Files.lines(csvFile)` lazily reads the file line by line.
- **Skipping headers**: `.skip(1)` ignores the CSV header row.
- **Splitting fields**: `.map(line -> line.split(","))` transforms each line into a String array.
- **Filtering invalid rows**: Ensures each line has exactly 3 fields.
- **Mapping to domain objects**: Tries to parse and create `Person` objects; returns `null` on parse failure.
- **Filtering nulls**: Removes entries where parsing failed.
- **Collecting results**: Gathers valid `Person` objects into a list.

14.3.4 Benefits of this Functional Approach

- **Readability**: Each operation focuses on a single transformation or filter.
- **Composability**: Easy to add new validation or transformation steps.
- **Efficiency**: Lazy reading and processing avoid loading entire files into memory.
- **Error handling**: Parsing errors are handled gracefully without exceptions propagating.

This simple CSV parser illustrates how Java Streams and lambdas can be combined for clean, functional data processing pipelines.

Chapter 15.

Building Domain-Specific Languages (DSLs)

1. What is a DSL?
2. Using Lambdas to Create Fluent APIs
3. Example: Building a Simple Query DSL

15 Building Domain-Specific Languages (DSLs)

15.1 What is a DSL?

A **Domain-Specific Language (DSL)** is a specialized programming language tailored to express solutions and logic within a particular problem domain. Unlike general-purpose programming languages (e.g., Java, Python), DSLs focus on providing **concise, readable, and expressive syntax** that matches the terminology and concepts familiar to domain experts.

15.1.1 Why Use DSLs?

DSLs improve productivity and maintainability by:

- **Simplifying complex logic**: They reduce verbose, low-level code into clear, domain-aligned expressions.
- **Enhancing readability**: DSLs make programs look closer to natural language or domain terms, easing understanding for developers and non-developers alike.
- **Reducing errors**: By restricting syntax and providing domain-specific operations, DSLs help catch mistakes early.

For example, a SQL query language is a classic DSL for managing databases, focusing specifically on querying data without worrying about implementation details.

15.1.2 Internal vs External DSLs

DSLs come in two main flavors:

- **External DSLs** are standalone languages with their own syntax, grammar, and parsers. Examples include SQL, regular expressions, or build tools like Gradle's Groovy DSL. These require separate tooling and often a compilation step.
- **Internal DSLs** (or embedded DSLs) are written within a host language like Java by leveraging the language's syntax and features. They provide domain-specific expressiveness **without needing a separate parser**. Java's lambdas and fluent APIs make building internal DSLs easier and more powerful.

15.1.3 Characteristics of Good DSLs

A well-designed DSL should have:

- **Fluency**: The code reads smoothly, almost like natural language, enabling easy chaining of operations.
- **Expressiveness**: It should capture domain concepts clearly, allowing complex logic to be expressed succinctly.
- **Minimal boilerplate**: Avoid excessive ceremony or verbose syntax that obscures intent.

15.1.4 Simple Illustrative Example

Consider a DSL for filtering orders by status and amount in an e-commerce system.

Traditional Java code might look like this:

```
List<Order> filtered = orders.stream()
    .filter(o -> o.getStatus().equals("SHIPPED"))
    .filter(o -> o.getAmount() > 100)
    .collect(Collectors.toList());
```

An internal DSL designed with fluent methods could look like:

```
List<Order> filtered = OrderQuery.from(orders)
    .statusIs("SHIPPED")
    .amountGreaterThan(100)
    .execute();
```

This reads more clearly, directly expressing the domain intent without exposing stream details.

15.1.5 Summary

DSLs empower developers to write domain logic that is easier to read, write, and maintain. Internal DSLs built with Java's functional features strike a great balance by embedding expressive, fluent APIs directly into the language, making complex domains approachable and code more expressive.

15.2 Using Lambdas to Create Fluent APIs

Java's introduction of **lambdas** and **functional interfaces** has transformed how we build expressive, fluent APIs that resemble internal domain-specific languages (DSLs). These APIs allow developers to write code that is readable, flexible, and concise—capturing domain intent while hiding boilerplate.

15.2.1 Fluent APIs and DSLs

A **fluent API** lets you chain method calls naturally, often resembling a sentence or domain-specific instruction. Combined with lambdas, fluent APIs can embed behavior directly, enabling custom logic passed inline without needing verbose anonymous classes.

15.2.2 How Lambdas Help

Lambdas reduce verbosity by replacing boilerplate code with concise functions. Functional interfaces such as `Predicate<T>`, `Function<T,R>`, and custom single-method interfaces enable passing behavior as parameters, configuring how a fluent API operates at runtime.

For example, consider a filtering API:

```
public interface Filter<T> {
    boolean test(T t);
}

public class FilterBuilder<T> {
    private Predicate<T> predicate = t -> true;

    public FilterBuilder<T> where(Predicate<T> condition) {
        predicate = predicate.and(condition);
        return this;
    }

    public List<T> apply(List<T> items) {
        return items.stream()
                    .filter(predicate)
                    .collect(Collectors.toList());
    }
}
```

Usage with lambdas:

```
List<String> data = List.of("apple", "banana", "avocado", "blueberry");

List<String> result = new FilterBuilder<String>()
    .where(s -> s.startsWith("a"))
    .where(s -> s.length() > 5)
    .apply(data);

System.out.println(result); // [avocado]
```

Here, lambdas allow callers to flexibly define conditions inline, while method chaining creates a fluent, readable API.

Full runnable code:

```java
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;

// Functional interface (optional since Predicate<T> is already functional)
interface Filter<T> {
    boolean test(T t);
}

// Builder class using Predicate<T> chaining
class FilterBuilder<T> {
    private Predicate<T> predicate = t -> true;

    public FilterBuilder<T> where(Predicate<T> condition) {
        predicate = predicate.and(condition);
        return this;
    }

    public List<T> apply(List<T> items) {
        return items.stream()
                    .filter(predicate)
                    .collect(Collectors.toList());
    }
}

public class FilterBuilderDemo {
    public static void main(String[] args) {
        List<String> data = List.of("apple", "banana", "avocado", "blueberry");

        List<String> result = new FilterBuilder<String>()
            .where(s -> s.startsWith("a"))
            .where(s -> s.length() > 5)
            .apply(data);

        System.out.println(result); // Output: [avocado]
    }
}
```

15.2.3 Builder Pattern and Method Chaining

The builder pattern pairs naturally with lambdas to create complex configurations step-by-step. Each method returns `this` or another builder, enabling chainable calls.

Example: configuring a report generator:

```java
public class ReportBuilder {
    private String title;
    private Consumer<String> formatter;

    public ReportBuilder title(String title) {
        this.title = title;
        return this;
    }
```

```
    public ReportBuilder format(Consumer<String> formatter) {
        this.formatter = formatter;
        return this;
    }

    public void generate() {
        String report = "Report: " + title;
        if (formatter != null) {
            formatter.accept(report);
        } else {
            System.out.println(report);
        }
    }
}
```

Usage:

```
new ReportBuilder()
    .title("Sales Q2")
    .format(r -> System.out.println("Formatted: " + r.toUpperCase()))
    .generate();
```

15.2.4 Function Composition

Functional interfaces support composition (`andThen`, `compose`) enabling DSLs that build pipelines of behavior dynamically. For instance:

```
Function<String, String> trim = String::trim;
Function<String, String> upper = String::toUpperCase;
Function<String, String> pipeline = trim.andThen(upper);

System.out.println(pipeline.apply("  hello world  ")); // "HELLO WORLD"
```

This composition style allows creating flexible, reusable DSL components.

15.2.5 Summary

By leveraging lambdas, method chaining, builders, and function composition, Java developers can craft **fluent APIs** that feel like internal DSLs. These APIs are more concise, flexible, and expressive than traditional patterns, making complex configuration or querying tasks more intuitive and maintainable.

15.3 Example: Building a Simple Query DSL

Creating a simple query DSL (Domain-Specific Language) using lambdas and fluent method chaining demonstrates how Java can embed expressive, readable domain logic directly in code. This example models a query over a collection of `Person` objects with customizable filters and transformations.

15.3.1 Defining the DSL

We start by defining a `Query<T>` interface representing a composable query. We use functional interfaces to capture predicates (filters) and mappers (transformations):

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class QueryDSL {

    // Core Query interface with fluent methods
    public static class Query<T> {
        private Stream<T> stream;

        private Query(Collection<T> source) {
            this.stream = source.stream();
        }

        // Static factory method to start query from a collection
        public static <T> Query<T> from(Collection<T> source) {
            return new Query<>(source);
        }

        // Filter with a predicate lambda
        public Query<T> where(Predicate<T> predicate) {
            stream = stream.filter(predicate);
            return this;
        }

        // Map to a different type with a mapper function
        public <R> Query<R> select(Function<T, R> mapper) {
            Stream<R> mappedStream = stream.map(mapper);
            Query<R> newQuery = new Query<>(List.of()); // dummy source for constructor
            newQuery.stream = mappedStream;
            return newQuery;
        }

        // Collect results into a list
        public List<T> execute() {
            return stream.collect(Collectors.toList());
        }
    }

    // Sample domain class
```

```
    static class Person {
        String name;
        int age;

        Person(String name, int age) { this.name = name; this.age = age; }

        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    // Sample usage
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 30),
            new Person("Bob", 20),
            new Person("Charlie", 25)
        );

        List<String> names = Query.from(people)
            .where(p -> p.age >= 21)                 // Filter adults
            .select(p -> p.name.toUpperCase())       // Map to uppercase names
            .execute();

        names.forEach(System.out::println);          // ALICE, CHARLIE
    }
}
```

15.3.2 Explanation

- **Fluent construction**: The static `from` method initializes the query over a collection.
- **Filtering with lambdas**: The `where` method accepts a `Predicate<T>`, enabling flexible, inline filter logic.
- **Mapping transformations**: The `select` method uses a `Function<T,R>` to transform items, returning a new `Query<R>`.
- **Execution**: The terminal `execute` method triggers the stream pipeline and collects results.

15.3.3 Benefits of this DSL Approach

- **Readability**: The chained method calls read like a declarative query aligned with domain logic.
- **Extensibility**: New methods like sorting or grouping can be added easily without changing client code.
- **Reusability**: Lambdas enable custom, reusable predicates and mappers.

- **Lazy evaluation**: Stream operations are only executed on `execute()`, improving efficiency.

This simple query DSL showcases how Java's functional features empower developers to create internal DSLs that cleanly express complex operations in an intuitive way.

Chapter 16.

Testing Functional Code

1. Unit Testing Lambdas and Functional Interfaces
2. Using Mocks and Stubs in Functional Context
3. Example: Testing Stream Pipelines

16 Testing Functional Code

16.1 Unit Testing Lambdas and Functional Interfaces

Testing lambda expressions and functional interfaces in Java poses some unique challenges due to their concise syntax and the fact that they often represent small, stateless functions embedded inside larger workflows. However, by applying best practices and leveraging modern testing frameworks like JUnit, you can write clear, maintainable tests for your functional code.

16.1.1 Challenges in Testing Lambdas

- **Inline definition**: Lambdas are often anonymous and defined inline, making them harder to isolate.
- **Side effects**: Functional interfaces like `Consumer<T>` may produce side effects that need verification.
- **Behavior verification**: Testing often requires checking function outputs for various inputs or capturing effects indirectly.
- **Composition**: Functional code frequently composes many small functions, requiring focused unit tests for each part.

16.1.2 Best Practices for Testing Functional Code

- **Extract lambdas to named variables or methods**: This makes your functions reusable and easier to test independently.
- **Test pure functions separately**: Functions without side effects are easiest to test with input/output assertions.
- **Use mocks or spies for side effects**: When lambdas interact with external systems or mutate state, use mocks (e.g., Mockito) to verify interactions.
- **Write parameterized tests**: Cover multiple input cases efficiently.

16.1.3 Testing Simple Functional Interfaces

Here are examples using JUnit 5 to test common functional interfaces:

Testing a Predicate

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.util.function.Predicate;

public class PredicateTest {

    Predicate<String> isLongerThan5 = s -> s.length() > 5;

    @Test
    void testIsLongerThan5() {
        assertTrue(isLongerThan5.test("functional"));
        assertFalse(isLongerThan5.test("java"));
    }
}
```

Testing a Function

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.util.function.Function;

public class FunctionTest {

    Function<String, Integer> stringLength = String::length;

    @Test
    void testStringLength() {
        assertEquals(4, stringLength.apply("test"));
        assertEquals(0, stringLength.apply(""));
    }
}
```

Testing a Consumer with Side Effects

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import java.util.function.Consumer;
import java.util.ArrayList;
import java.util.List;

public class ConsumerTest {

    @Test
    void testConsumerAddsToList() {
        List<String> list = new ArrayList<>();
        Consumer<String> addToList = list::add;

        addToList.accept("hello");
        addToList.accept("world");

        assertEquals(2, list.size());
        assertEquals("hello", list.get(0));
```

```
    }
}
```

16.1.4 Verifying Behavior with Mocks

For lambdas that invoke external dependencies, use mocking frameworks to verify interactions:

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.function.Consumer;

public class MockConsumerTest {

    @Test
    void testConsumerInvokesDependency() {
        Consumer<String> mockConsumer = mock(Consumer.class);

        mockConsumer.accept("data");

        verify(mockConsumer).accept("data");
    }
}
```

16.1.5 Summary

Testing lambdas and functional interfaces requires isolating the logic, handling side effects carefully, and using clear assertions. By extracting lambdas into named functions and applying standard JUnit techniques, you can ensure your functional code is robust, testable, and maintainable.

16.2 Using Mocks and Stubs in Functional Context

When testing functional code, especially where **side effects** or **external dependencies** occur, mocks and stubs become essential tools. Functional programming encourages pure functions, but real-world applications often interact with external systems (databases, APIs) or have stateful operations. Mocks and stubs help isolate the functional logic by simulating these interactions, making tests predictable, repeatable, and focused.

16.2.1 Why Use Mocks and Stubs?

- **Control side effects**: Prevent real external calls that are slow, unreliable, or hard to reproduce.
- **Verify behavior**: Confirm that functional interfaces like `Consumer` or callbacks are invoked correctly.
- **Isolate units**: Test functional code independently from dependencies.
- **Test async/stateful behavior**: Simulate asynchronous callbacks or state changes.

16.2.2 Using Mockito to Mock Functional Interfaces

Mockito is a popular mocking framework in Java that seamlessly integrates with lambdas and functional interfaces.

Mocking a Supplier

A `Supplier<T>` returns a value without input. Mocking allows specifying what the supplier returns:

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.function.Supplier;

public class SupplierMockTest {

    @Test
    void testMockSupplier() {
        Supplier<String> supplier = mock(Supplier.class);
        when(supplier.get()).thenReturn("mocked value");

        String result = supplier.get();

        assertEquals("mocked value", result);
        verify(supplier).get();
    }
}
```

Mocking a Consumer

A `Consumer<T>` performs an action with side effects. Mocks verify that expected inputs are consumed:

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.function.Consumer;

public class ConsumerMockTest {

    @Test
```

```
    void testMockConsumer() {
        Consumer<String> consumer = mock(Consumer.class);

        consumer.accept("test input");

        verify(consumer).accept("test input");
    }
}
```

Mocking Callbacks and Handling Asynchrony

Functional code often uses callbacks invoked asynchronously. You can simulate asynchronous behavior by invoking the callback mock manually or with executors.

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.function.Consumer;

public class AsyncCallbackTest {

    interface AsyncService {
        void fetchData(Consumer<String> callback);
    }

    @Test
    void testAsyncCallback() {
        AsyncService service = mock(AsyncService.class);
        Consumer<String> callback = mock(Consumer.class);

        doAnswer(invocation -> {
            Consumer<String> cb = invocation.getArgument(0);
            cb.accept("async result");  // Simulate async callback
            return null;
        }).when(service).fetchData(any());

        service.fetchData(callback);

        verify(callback).accept("async result");
    }
}
```

16.2.3 Strategies for Pure and Predictable Tests

- **Keep mocks focused**: Only mock what's necessary to isolate the tested logic.
- **Avoid complex behavior in mocks**: Use simple returns or verifications rather than simulating full external systems.
- **Use stubs for static responses**: When interaction verification isn't needed, stubs can provide canned responses.
- **Combine with pure functions**: Minimize side effects inside lambdas and isolate side-effectful code for easier mocking.

16.2.4 Summary

Mocks and stubs are vital in functional programming tests where side effects or dependencies exist. Mockito's ability to mock lambdas and functional interfaces lets you simulate suppliers, consumers, and callbacks effortlessly. By isolating effects and verifying interactions, your tests stay clean, pure, and predictable, ensuring functional logic is robust and reliable.

16.3 Example: Testing Stream Pipelines

Testing stream pipelines involves verifying that a sequence of operations—such as filtering, mapping, and collecting—produces the expected output for given inputs. The goal is to isolate the stream logic and assert outcomes clearly, including edge cases like empty or null inputs.

16.3.1 Example: Stream Pipeline to Process Employee Names

Suppose we have a method that processes a list of employees, filters those older than 25, converts their names to uppercase, and collects them into a list.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import java.util.*;
import java.util.stream.*;

public class StreamPipelineTest {

    static class Employee {
        String name;
        int age;

        Employee(String name, int age) {
            this.name = name;
            this.age = age;
        }
    }

    /**
     * Processes employees by filtering age > 25,
     * mapping names to uppercase, and collecting as list.
     */
    public List<String> processEmployees(List<Employee> employees) {
        if (employees == null) {
            return Collections.emptyList(); // Defensive null handling
        }

        return employees.stream()
```

```
                .filter(e -> e.age > 25)                    // Filter employees older than 25
                .map(e -> e.name.toUpperCase())              // Convert names to uppercase
                .collect(Collectors.toList());               // Collect to list
    }

    @Test
    void testProcessEmployees_withValidData() {
        List<Employee> input = Arrays.asList(
                new Employee("Alice", 30),
                new Employee("Bob", 20),
                new Employee("Charlie", 28)
        );

        List<String> expected = Arrays.asList("ALICE", "CHARLIE");
        List<String> result = processEmployees(input);

        assertEquals(expected, result);
    }

    @Test
    void testProcessEmployees_withEmptyList() {
        List<Employee> input = Collections.emptyList();
        List<String> result = processEmployees(input);

        assertTrue(result.isEmpty());
    }

    @Test
    void testProcessEmployees_withNullInput() {
        List<String> result = processEmployees(null);
        assertTrue(result.isEmpty());
    }

    @Test
    void testProcessEmployees_allFilteredOut() {
        List<Employee> input = Arrays.asList(
                new Employee("Dave", 22),
                new Employee("Eve", 18)
        );

        List<String> result = processEmployees(input);
        assertTrue(result.isEmpty());
    }
}
```

16.3.2 Explanation

- The method `processEmployees` is isolated and handles null inputs gracefully.
- Stream operations are tested end-to-end: filtering employees older than 25, mapping names, collecting results.
- Multiple test cases cover:

  - Typical input producing filtered, mapped results.
  - Empty input list.
  - Null input handling.
  - Input where all employees are filtered out.
- Assertions check for exact match or empty results.
- Tests focus on input/output behavior without internal implementation details.

16.3.3 Summary

Unit testing stream pipelines is straightforward when you isolate the pipeline into a method and provide diverse input scenarios. Handling edge cases like empty or null inputs prevents runtime errors and ensures robustness. Clear, assertive tests verify that filtering, mapping, and collecting logic behaves as expected, maintaining code quality in functional Java applications.

Chapter 17.

Appendix

1. Common Functional Interfaces and Usage Examples
2. Writing Efficient Functional Java Code
3. Common Pitfalls and How to Avoid Them

17 Appendix

17.1 Common Functional Interfaces and Usage Examples

Java's `java.util.function` package provides a rich set of **functional interfaces**—interfaces with a single abstract method—that serve as building blocks for functional programming. Understanding these interfaces helps write concise, reusable, and expressive lambda expressions and method references.

Below are the most frequently used functional interfaces, their method signatures, and typical use cases:

`Function<T, R>`

- **Signature:** `R apply(T t)`
- **Purpose:** Represents a function that takes an argument of type `T` and returns a result of type `R`.
- **Use Case:** Transforming data, mapping values.

```
Function<String, Integer> stringLength = s -> s.length();
int len = stringLength.apply("hello");  // returns 5
```

Tip: Use `Function` when you need to convert or transform one type into another.

`PredicateT`

- **Signature:** `boolean test(T t)`
- **Purpose:** Represents a boolean-valued function of one argument.
- **Use Case:** Filtering or matching elements based on a condition.

```
Predicate<String> isEmpty = String::isEmpty;
boolean result = isEmpty.test("");  // returns true
```

Tip: Use `Predicate` for conditions, filters, and validations.

`Consumer<T>`

- **Signature:** `void accept(T t)`
- **Purpose:** Represents an operation that takes a single input and returns no result.
- **Use Case:** Performing side effects like printing or modifying external state.

```
Consumer<String> print = System.out::println;
print.accept("Hello World");  // Prints: Hello World
```

Tip: Use `Consumer` when you want to perform an action with an input but don't need to return anything.

`Supplier<T>`

- **Signature:** `T get()`
- **Purpose:** Represents a supplier of results with no input.
- **Use Case:** Lazy generation or provision of values, factories.

```
Supplier<Double> randomSupplier = Math::random;
double randomValue = randomSupplier.get();
```

Tip: Use `Supplier` to defer execution or provide default/lazy values.

`UnaryOperator<T>`

- **Signature:** `T apply(T t)`
- **Purpose:** A special case of `Function` where the input and output are the same type.
- **Use Case:** In-place transformations or updates.

```
UnaryOperator<String> toUpperCase = String::toUpperCase;
String result = toUpperCase.apply("java");  // returns "JAVA"
```

Tip: Use `UnaryOperator` when transforming an object to another instance of the same type.

`BiFunction<T, U, R>`

- **Signature:** `R apply(T t, U u)`
- **Purpose:** Represents a function that takes two arguments and produces a result.
- **Use Case:** Operations involving two inputs, like combining or comparing values.

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
int sum = add.apply(5, 7);  // returns 12
```

Tip: Use `BiFunction` when you need a function with two inputs and one output.

17.1.1 Choosing the Right Interface

- **Transformation:** Use `Function` or `UnaryOperator`.
- **Condition checking:** Use `Predicate`.
- **Side effects:** Use `Consumer`.
- **No input, only output:** Use `Supplier`.
- **Two inputs:** Use `BiFunction`.

17.1.2 Summary

Java's built-in functional interfaces form the foundation for clean, expressive functional code. By selecting the right interface based on your operation's input and output requirements, you simplify code and enable easy composition of behavior with lambdas and method references.

Mastering these interfaces unlocks the full power of Java's functional programming capabilities.

17.2 Writing Efficient Functional Java Code

Functional programming in Java offers expressive and concise ways to write code, but writing **efficient** functional code requires attention to performance details. Here are key considerations and best practices to help you write clean **and** performant functional Java code.

Minimize Unnecessary Object Creation

Lambdas and streams can generate many temporary objects (e.g., boxed primitives, intermediate collections). Excessive object creation increases GC pressure and slows down your application.

- **Use primitive-specialized functional interfaces** such as `IntPredicate`, `IntFunction`, and `IntStream` when working with primitives to avoid boxing overhead.
- Avoid unnecessary mapping steps that create new objects if you don't need them.

```
// Prefer IntStream over Stream<Integer> to avoid boxing
int sum = IntStream.range(1, 1000)
                   .filter(i -> i % 2 == 0)
                   .sum();
```

Avoid Expensive Intermediate Operations

Intermediate stream operations are lazy but can be costly if applied incorrectly.

- Use **short-circuiting operations** like `limit()`, `findFirst()`, or `anyMatch()` early to reduce processing.
- Avoid repeated traversals of the same stream.
- Combine multiple operations when possible to reduce overhead.

```
// Stop processing after finding the first even number
OptionalInt firstEven = IntStream.range(1, 1_000_000)
                                 .filter(i -> i % 2 == 0)
                                 .findFirst();
```

Choose Between Sequential and Parallel Streams Wisely

Parallel streams can speed up CPU-bound operations but add overhead due to thread management.

- Use parallel streams for large datasets and CPU-intensive tasks.
- For small or simple pipelines, sequential streams often perform better.
- Always **benchmark** your specific use case before adopting parallelism.

```
List<String> data = ...;
// Use parallel stream only when justified
List<String> results = data.parallelStream()
                           .filter(s -> s.length() > 5)
                           .collect(Collectors.toList());
```

Leverage Method References and Avoid Capturing Variables

Method references (`Class::method`) are often more efficient than lambdas because they can avoid capturing variables and thus reduce object allocation.

```
// Efficient and clean method reference
list.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

Keep Pipelines Simple and Readable

Overly complex pipelines with nested lambdas hurt readability and maintainability, which indirectly affect long-term performance due to bugs or poor optimizations.

- Break down complex pipelines into smaller reusable functions.
- Use descriptive variable names and avoid side effects inside stream operations.

17.2.1 Summary

Efficient functional Java code balances readability with performance. Use primitive streams to avoid boxing, leverage short-circuiting to limit processing, carefully decide when to parallelize, and prefer method references for concise, low-overhead lambdas. By combining these best practices, you write clean, maintainable, and performant functional programs. Always measure performance impacts in the context of your real application to make informed decisions.

17.3 Common Pitfalls and How to Avoid Them

Adopting functional programming in Java brings many benefits but also introduces pitfalls, especially for developers new to the paradigm. Being aware of these common mistakes helps you write more robust, maintainable functional code.

Mutating State Inside Streams

Problem: Streams are designed for declarative, side-effect-free operations. Mutating external state (e.g., modifying a collection or a variable) within stream operations breaks this model, causing unpredictable behavior, especially with parallel streams.

Why it's problematic:

- Causes race conditions and data corruption with parallel streams.
- Makes code harder to understand and debug.

How to avoid: Use pure functions inside streams. Accumulate results using collectors or return new immutable objects instead of modifying shared state.

```
// Bad: mutating external list inside stream
List<String> names = new ArrayList<>();
stream.forEach(s -> names.add(s.toUpperCase())); // Unsafe!

// Good: collect results immutably
List<String> names = stream
                     .map(String::toUpperCase)
                     .collect(Collectors.toList());
```

Improper Exception Handling in Streams

Problem: Checked exceptions cannot be thrown directly from lambdas, which leads to boilerplate or swallowing exceptions inside streams.

Why it's problematic:

- Exception handling code becomes cluttered or ignored.
- Can cause silent failures or runtime crashes.

How to avoid: Wrap checked exceptions into unchecked ones or create utility methods to handle exceptions functionally.

```
// Example utility wrapper for checked exceptions
static <T, R> Function<T, R> wrap(CheckedFunction<T, R> func) {
    return t -> {
        try {
            return func.apply(t);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    };
}
```

```
// Usage:
stream.map(wrap(s -> someIOOperation(s)))
      .collect(Collectors.toList());
```

Misusing Parallel Streams

Problem: Parallel streams don't always improve performance and can degrade it due to overhead or thread contention.

Why it's problematic:

- Small or IO-bound tasks often run slower in parallel.
- Shared mutable state leads to concurrency bugs.

How to avoid:

- Benchmark before parallelizing.
- Ensure operations are stateless and side-effect free.
- Avoid shared mutable data structures.

Overcomplicating Simple Problems

Problem: Functional style can tempt developers to over-engineer solutions using streams or lambdas where simple loops or conditionals suffice.

Why it's problematic:

- Decreases code readability and maintainability.
- Introduces unnecessary complexity and performance overhead.

How to avoid:

- Choose the simplest clear approach, even if imperative.
- Use streams for data transformations and pipelines, but not for trivial logic.

17.3.1 Summary

Avoid mutating state in streams, handle exceptions thoughtfully, use parallel streams judiciously, and resist overcomplicating code. These guidelines will help you write clear, correct, and efficient functional Java programs. Functional programming isn't about forcing every piece of code into lambdas but about leveraging their power where they provide genuine benefits.