Functional composition in Java

give me the deep and detailed explaination of Functional Composition
in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Alright Piyush 🙂
Since you already understand **lambdas, functional interfaces, and higher-order functions**, we'll go **very deep and very systematic** so that *functional composition in Java* becomes crystal clear and you don't carry any hidden confusion.

I'll explain **what it is, why it exists, how Java implements it, how it actually works internally, edge cases, mental models, and real-world usage**.

---

1. What is Functional Composition? (Core Idea)

Functional composition means:

> **Combining multiple functions so that the output of one function becomes the input of another function**

Mathematically:

```
(f ∘ g)(x) = f(g(x))
```

Read this as:

> "Apply `g` first, then apply `f` on the result"

Simple real-life analogy 🍵

Imagine:

1. **Wash vegetables**
2. **Cut vegetables**
3. **Cook vegetables**

Each step:

- Takes something

- Returns something

You don't do all steps at once.
You **compose** them in a pipeline.

Functional composition in Java does **exactly this**, but with **functions instead of steps**.

---

2. Why Do We Need Functional Composition?

Without composition, code looks like this:

```
java

int result = addOne(square(doubleIt(x)));
```

Problems:
❌ Hard to read
❌ Deep nesting
❌ Difficult to reuse logic
❌ Difficult to reorder steps

With **functional composition**:

```
java

Function<Integer, Integer> pipeline =
        doubleIt.andThen(square).andThen(addOne);

int result = pipeline.apply(x);
```

✔️ Clean
✔️ Readable
✔️ Reusable
✔️ Flexible

---

3. Functional Interfaces That Support Composition

In Java, **composition is built into functional interfaces**.

Main ones:

Interface	Purpose
`Function<T, R>`	Transform input to output
`UnaryOperator<T>`	Same input & output type
`BiFunction<T, U, R>`	Two inputs
`Predicate<T>`	Returns boolean
`Consumer<T>`	Performs action
`Supplier<T>`	Produces value

4. Functional Composition with `Function<T, R>`

This is **THE MOST IMPORTANT** interface for composition.

```
java

Function<T, R>
```

Means:

- Takes `T`
- Returns `R`

4.1 Two Composition Methods

1️⃣ `andThen()`

```
java

f.andThen(g)
```

➡️ First f, then g

```
css

x → f(x) → g(f(x))
```

2️⃣ `compose()`

```
java

f.compose(g)
```

➡️ First g, then f

```
css

x → g(x) → f(g(x))
```

4.2 Visual Difference

```
makefile

andThen:
x → f → g → result
```

```
compose:
x → g → f → result
```

---

5. Step-by-Step Example (No Confusion Version)

Define functions:

```
java

Function<Integer, Integer> doubleIt = x -> x * 2;
Function<Integer, Integer> square = x -> x * x;
```

Using `andThen`

```
java

Function<Integer, Integer> result =
        doubleIt.andThen(square);

System.out.println(result.apply(3));
```

Execution flow:

```
scss

3 → doubleIt → 6 → square → 36
```

Output:

```

36
```

Using `compose`

```
java

Function<Integer, Integer> result =
        doubleIt.compose(square);

System.out.println(result.apply(3));
```

Execution flow:

```
scss

3 → square → 9 → doubleIt → 18
```

Output:

```
18
```

🔥 Golden Rule (Memorize This)

`andThen()` → **left to right execution**
`compose()` → **right to left execution**

---

6. Internal Working (How Java Actually Does It)

When you write:

```
java

f.andThen(g)
```

Java internally creates **a new Function**:

```
java

x -> g.apply(f.apply(x))
```

Similarly:

```
java

f.compose(g)
```

Becomes:

```
java

x -> f.apply(g.apply(x))
```

💡 **No magic**
💡 Just function wrapping

---

7. Multiple Function Composition (Chaining)

You can chain **any number of functions**.

```
java

Function<Integer, Integer> pipeline =
        doubleIt
            .andThen(square)
```

```
                .andThen(x -> x + 10)
                .andThen(x -> x / 2);
```

Execution:

```
csharp

x → double → square → add 10 → divide by 2
```

This is how **data pipelines** are built in Java.

---

8. Composition with `UnaryOperator<T>`

`UnaryOperator<T>` is just:

```
java

Function<T, T>
```

Same input and output type.

```
java

UnaryOperator<Integer> inc = x -> x + 1;
UnaryOperator<Integer> triple = x -> x * 3;

UnaryOperator<Integer> result = inc.andThen(triple);

System.out.println(result.apply(5)); // (5+1)*3 = 18
```

✔️ Cleaner
✔️ Type-safe

---

9. Predicate Composition (Very Important)

Predicates return **boolean**, so composition uses **logical operators**.

Methods:

- **`and()`**
- **`or()`**
- **`negate()`**

Example:

```
java

Predicate<Integer> isPositive = x -> x > 0;
Predicate<Integer> isEven = x -> x % 2 == 0;

Predicate<Integer> positiveAndEven =
```

```
        isPositive.and(isEven);

System.out.println(positiveAndEven.test(10)); // true
```

Execution:

```
scss

isPositive(10) && isEven(10)
```

Negation:

```
java

Predicate<Integer> notPositive =
        isPositive.negate();
```

---

10. Consumer Composition

Consumers don't return values → they perform **actions**.

Method:

```
java

andThen()
```

```
java

Consumer<String> print = s -> System.out.println(s);
Consumer<String> log = s -> System.out.println("LOG: " + s);

Consumer<String> combined = print.andThen(log);

combined.accept("Hello");
```

Output:

```
makefile

Hello
LOG: Hello
```

---

11. Supplier Composition (Important Limitation)

❌ Suppliers cannot be composed directly

Why?

- Supplier has **no input**

- Composition requires chaining input → output

So instead:

```
java

Supplier<Integer> supplier = () -> 10;
Function<Integer, Integer> square = x -> x * x;

int result = square.apply(supplier.get());
```

---

12. Real-World Example (Backend Perspective 🚀)

Since you're learning **backend & Spring Boot**, this is VERY relevant.

Example: Request Processing Pipeline

```
java

Function<String, String> trim = String::trim;
Function<String, String> toLower = String::toLowerCase;
Function<String, String> validate = s -> {
    if (s.isEmpty()) throw new RuntimeException("Invalid");
    return s;
};

Function<String, String> pipeline =
        trim.andThen(toLower).andThen(validate);
```

✔️ Clean
✔️ Testable
✔️ Reusable

---

13. Common Mistakes & Confusions

❌ Mistake 1: Confusing order

```
java

f.compose(g)  // g first, then f
```

❌ Mistake 2: Type mismatch

```
java

Function<Integer, String> f;
Function<Double, Integer> g;

f.andThen(g); // ❌ types don't match
```

❌ Mistake 3: Expecting composition for Supplier

Not possible directly.

---

14. When Should You Use Functional Composition?

✔️ Data transformation pipelines
✔️ Stream operations
✔️ Validation chains
✔️ Pre-processing & post-processing
✔️ Clean backend service layers

❌ Simple one-line logic
❌ Overusing where readability drops

---

15. Mental Model (Very Important)

Always think:

> "This function produces something → the next function consumes it"

If that chain makes sense → composition works.

---

16. One-Line Summary (For Exams & Interviews)

> **Functional composition in Java is the process of combining multiple functions into a single function such that the output of one function becomes the input of the next, achieved using methods like `andThen()` and `compose()` in functional interfaces.**

If you want next, I can:

- Dry-run **compose vs andThen** with diagrams
- Show **Stream API + composition**
- Explain **performance & immutability aspects**
- Give **interview-level tricky questions**

Just tell me 👍