Applying Functional Programming Techniques in Java

THINKING FUNCTIONALLY WITH JAVA

Esteban Herrera
JAVA ARCHITECT

@eh3rrera www.eherrera.net

Java 8 Functional Features

Lambda Expressions

Method References

Streams

Is This Functional Programming?

```
list.stream()
    .filter(Validator::isValidOrThrowRuntimeException)
    .map(State::addToState)
    .forEach(System.out::println);
```

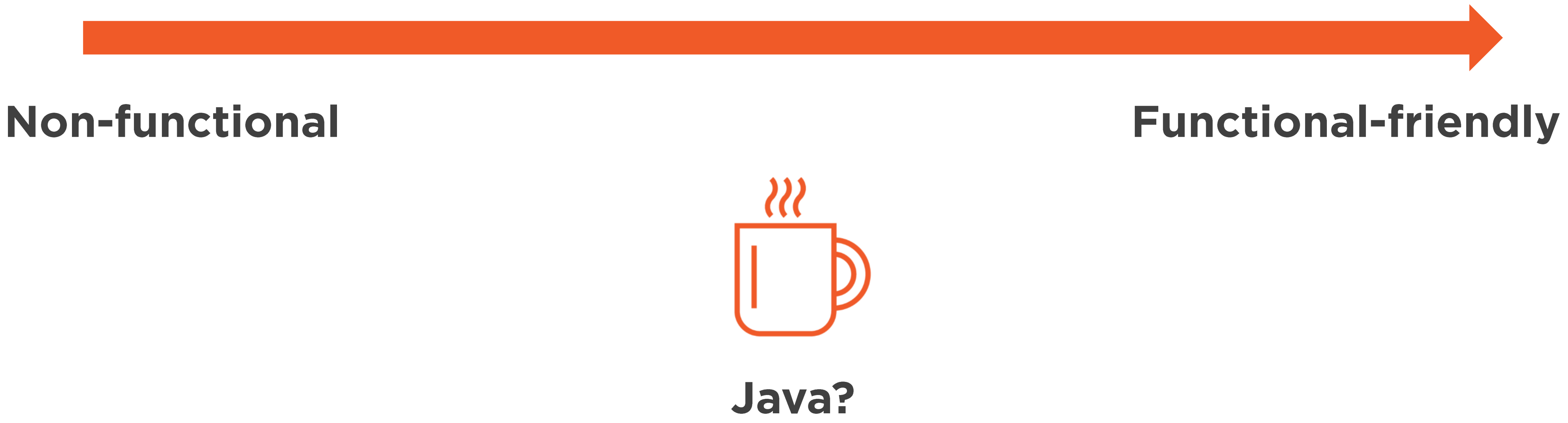
Functional Programming

$$\text{Pure Functions} \longrightarrow \text{Mathematical Functions}$$

Functions

- Just do one thing
- Don’t depend on anything else but their arguments
- And always give us the same result

Functional Programming Languages?



It’s not the language that makes programming functional.

It’s the way you write the code.

In Java

Imperative

```
List<Order> shipped = new ArrayList<>();
for (Order order : orders)
   if (order.isShipped())
      shipped.add(order);
```

Functional

```
List<Order> shipped =
      orders.stream()
            .filter(Order::isShipped)
            .collect(Collectors.toList());
```

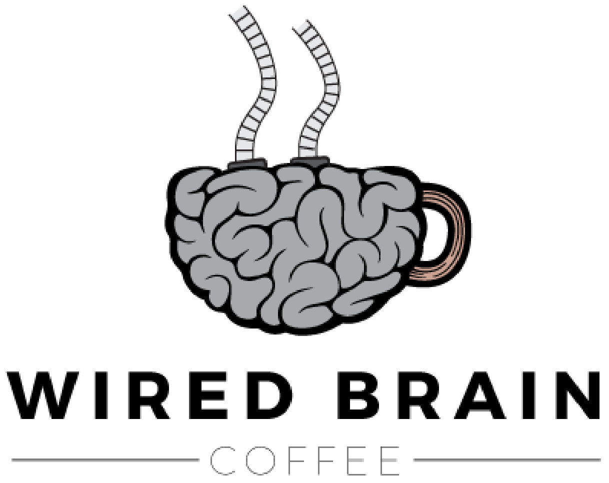
It’s Like Learning a Foreign Language

Practicing is the key

This Course

λ	*f*
Java 8 features	**Functional concepts**
✗	✓

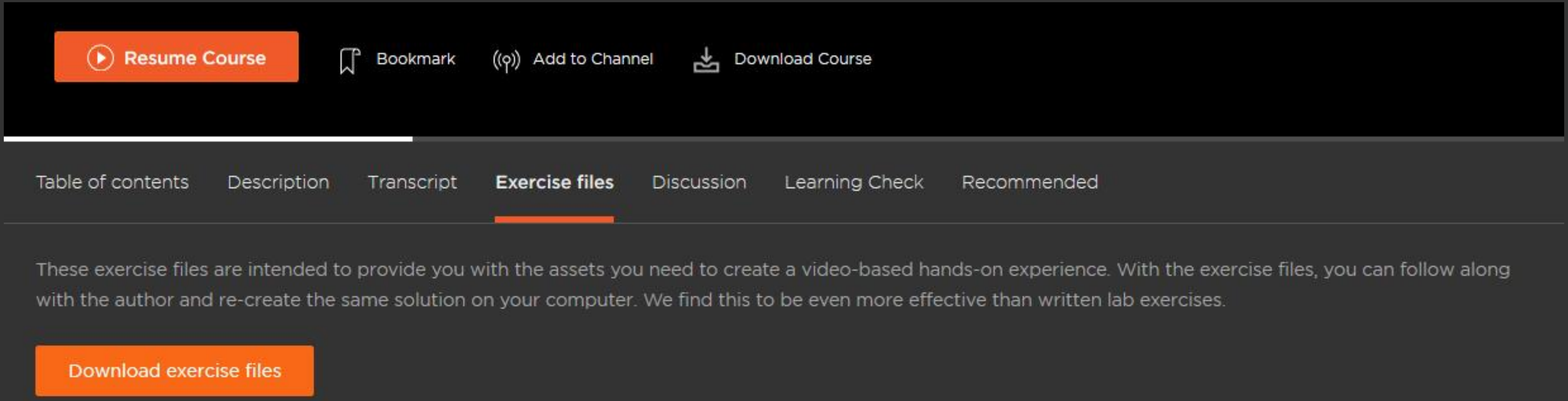
Sample Scenario



Loyalty Program

- Reward Points
- Discounts or gift products

Download the Demo Code



These exercise files are intended to provide you with the assets you need to create a video-based hands-on experience. With the exercise files, you can follow along with the author and re-create the same solution on your computer. We find this to be even more effective than written lab exercises.

Code compatible with Java 8+

The Single Responsibility Principle

```java
class Order {
    private Customer customer;
    private OrderStatus orderStatus;

    public void issueRewards() {
        this.orderStatus = OrderStatus.REWARDS_ISSUED;
        if (this.getOrderRewards() != null) {
            this.customer.addToRewardBalance(this.getOrderRewards());
        }
    }
}
```

```java
class Order {
    private Customer customer;
    private OrderStatus orderStatus;

    public void issueRewards() {
        this.orderStatus = OrderStatus.REWARDS_ISSUED;
    }

    public void updateBalanceReward() {
        if (this.getOrderRewards() != null) {
            this.customer.addToRewardBalance(this.getOrderRewards());
        }
    }
}
```

Caller Code

```
order.issueRewards();
```

↓

```
order.issueRewards();
order.updateBalanceReward();
```

```java
class Order {
    private Customer customer;
    private OrderStatus orderStatus;

    public void issueRewards() {
        this.orderStatus = OrderStatus.REWARDS_ISSUED;
    }

    public void updateBalanceReward() {
        if (this.getOrderRewards() != null) {
            this.customer.addToRewardBalance(this.getOrderRewards());
        }
    }
}
```

```java
class Order {

    private OrderStatus orderStatus;

    public void issueRewards() {
        this.orderStatus = OrderStatus.REWARDS_ISSUED;
    }

    public void updateBalanceReward(Customer customer) {
        if (this.getOrderRewards() != null) {
            customer.addToRewardBalance(this.getOrderRewards());
        }
    }
}
```

Caller Code

```
order.issueRewards();
order.updateBalanceReward();
```

$$\downarrow$$

```
Customer customer = //...
order.issueRewards();
order.updateBalanceReward(customer);
```

Functional Programming

Object state

Void Functions

Functions with One Responsibility

Single Responsibility Principle

S	Single responsibility principle
O	Open/closed principle
L	Liskov substitution principle
I	Interface segregation principle
D	Dependency inversion principle

Trust

Doing one thing

```java
class Order {

    private OrderStatus orderStatus;

    public void issueRewards() {
        this.orderStatus = OrderStatus.REWARDS_ISSUED;
    }

    public void updateBalanceReward(Customer customer) {
        if (this.getOrderRewards() != null) {
            customer.addToRewardBalance(this.getOrderRewards());
        }
    }
}
```

No Side Effects

A Mathematical Function

```
f(total) = total * 0.1
```

```
customer.getRewardBalance(); // 10

order.issueRewards();

customer.getRewardBalance(); // 15
```

Side effect

Examples of Side Effects

✓ Mutation of variables

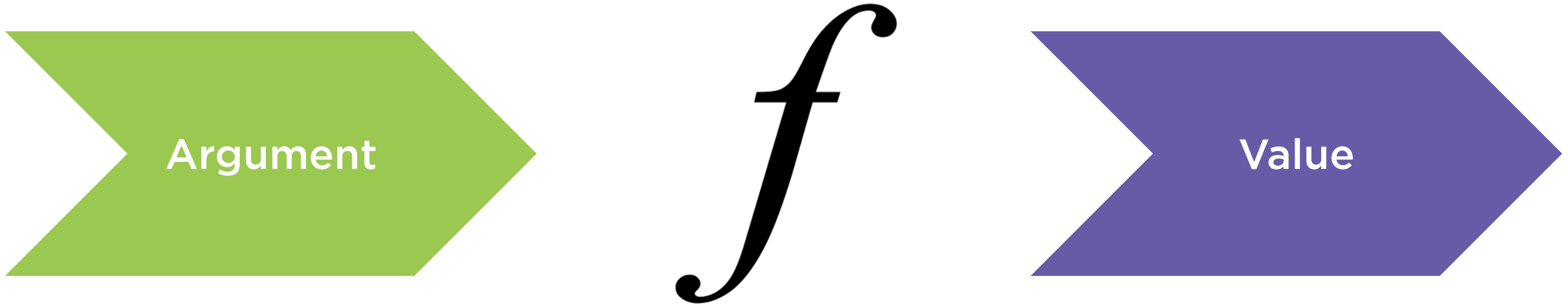
✓ Printing to the console

✓ Writing to files, databases, or anything in the outside world

The goal is NOT to eliminate side effects.

The goal is to eliminate OBSERVABLE side effects.

Functions Are Black Boxes



Referential Transparency

```
f(total) = total * 0.1
```

Replace a Function Call with Its Value

$$f$$

Replace a Function Call with Its Value

Referential Transparency

Object-oriented Programming

Encapsulation

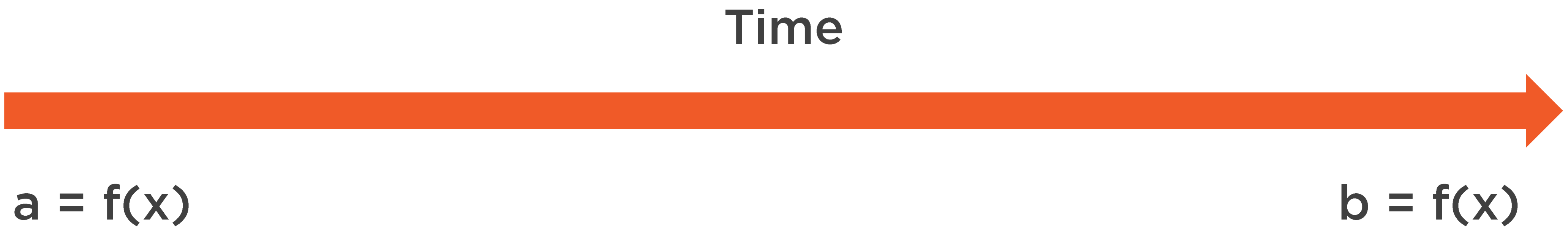
Functional Programming

Referential transparency

```
customer.getRewardBalance(); // 10

Customer newCustomer = updateBalanceReward(order, customer);

newCustomer.getRewardBalance(); // 15

customer.getRewardBalance(); // 10
```

To achieve referential transparency, we need immutable data.

The Problem with Assignment Statements



a == b

May be false

Immutability in Java

```
class Order {
      OrderStatus orderStatus;
      Integer orderRewards;

      BigDecimal total;
      Integer orderNumber;
      List<Products> products;
      Customer customer;
      Invoice invoice;
      Payment payment;
}
```

Immutability in Java

Affects performance

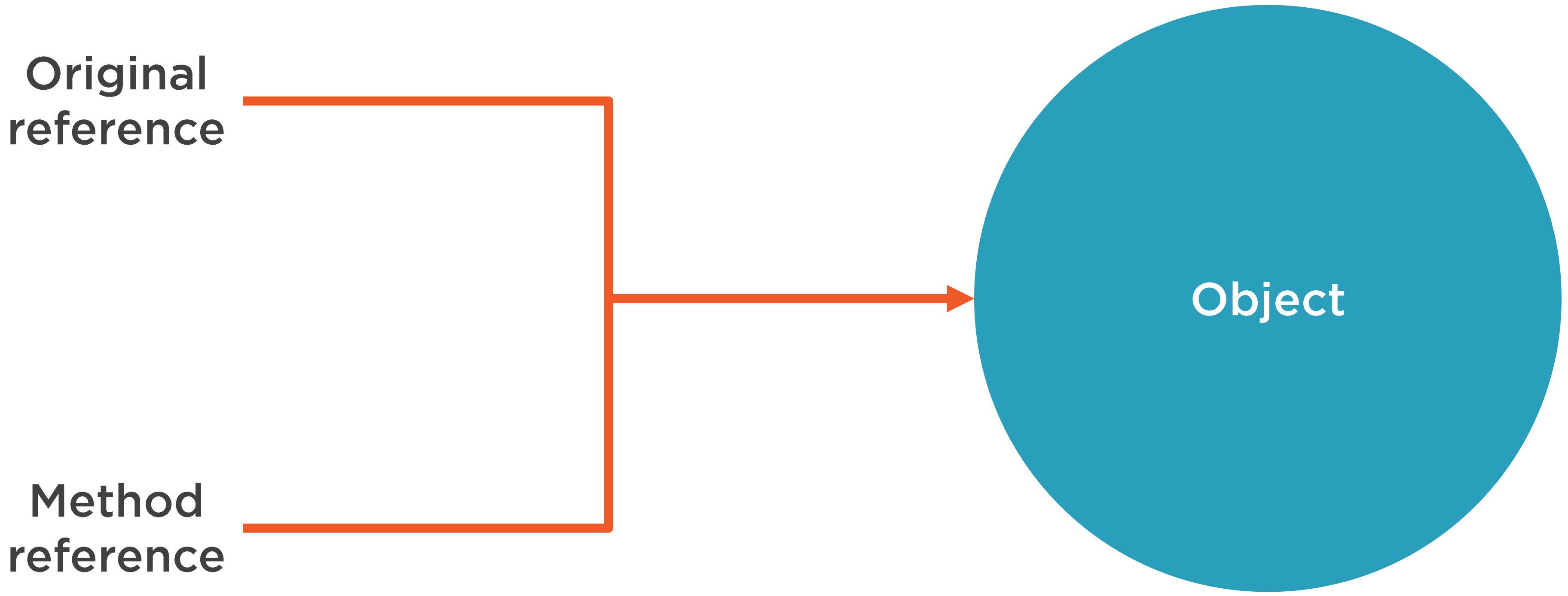
Increases memory

Functional Programming Languages

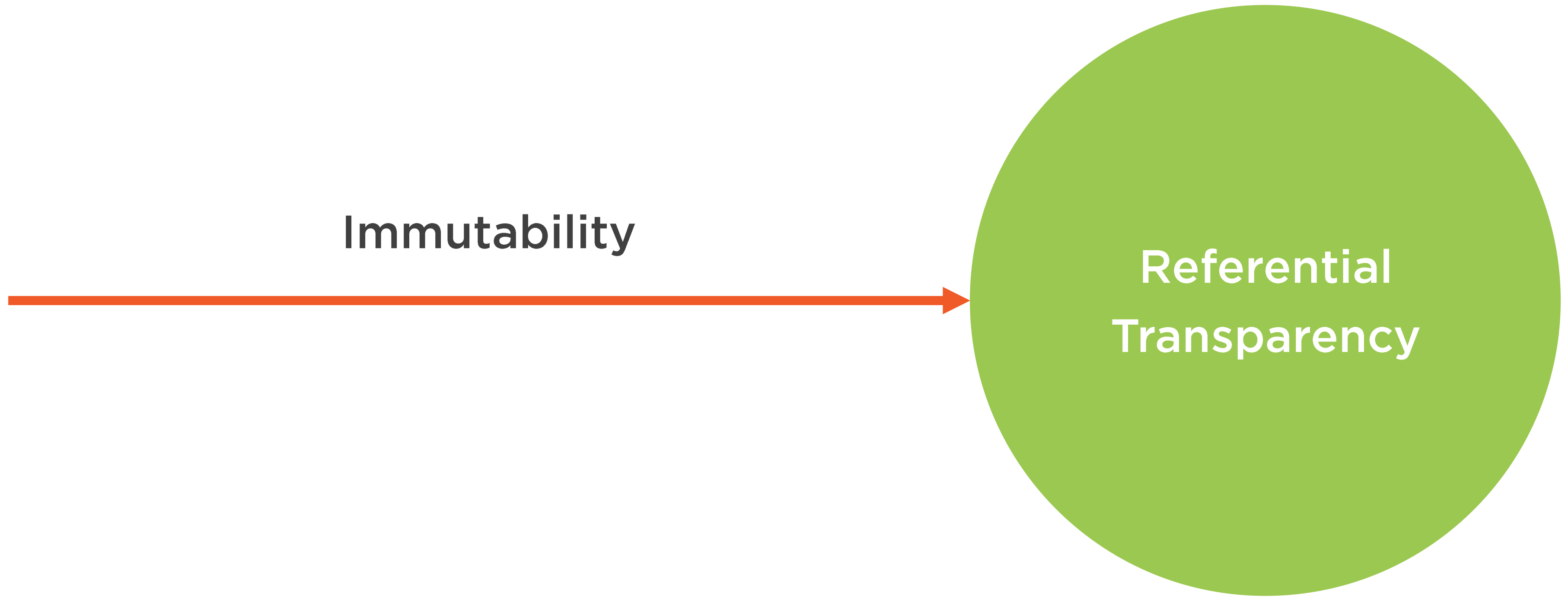
Immutable data
(optimized data structures)

Data modification
as a compiler error

Passing by Reference in Java



Immutability Is Not the Goal



Pushing Abstraction

Pure Functions Benefits

No unexpected results

No side effects

Thread safety

Modular programs

f

```
List<Order> shipped = new ArrayList<>();
for (Order order : orders)
   if (order.isShipped())
      shipped.add(order);
```

$$f$$

```java
List<Order> shipped =
      orders.stream()
          .filter(Order::isShipped)
          .collect(Collectors.toList());
```

Summary and What’s Ahead

Functional Programming

Pure Functions $\longrightarrow$ **Mathematical Functions**

Pure Functions

Single Responsibility

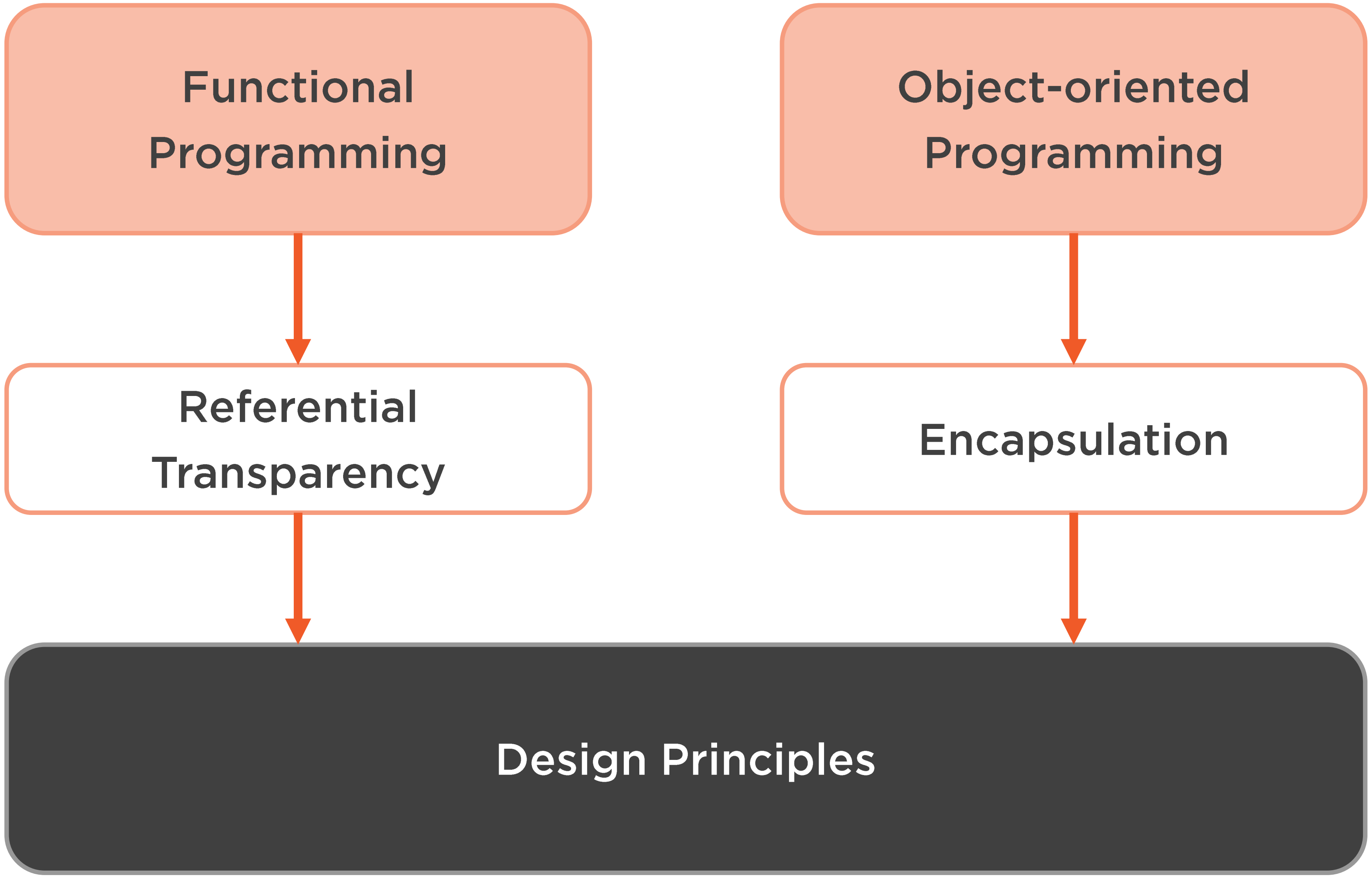
No Side effects

Referentially Transparent

From Imperative to Functional

1. Move methods to a new class, honoring the single responsibility principle
2. Pass the original classes as inputs of the new methods
3. Modify these methods to honor immutability

Functional and Object-oriented Programming



Functional Programming Techniques

Immutability

High-order Functions

Currying

Recursion

Lazy Evaluation

The Plan for This Course

Building complex functionality by composing functions

Creating reusable functions with partial application and currying

Abstracting control structures to control the application flow

The Plan for This Course

Avoiding nulls with the Optional type

Handling errors in a functional way

Building containers for side effects

The Plan for This Course

Java features

➖

Build own types

➕