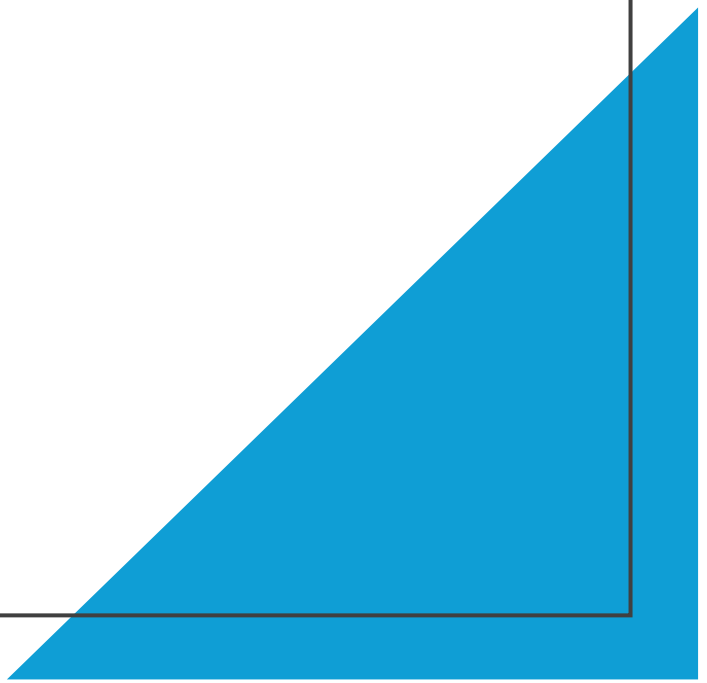
Functional Interfaces

...and Lambdas



1

Part I

- Sorting
- FP vs. OOP
- Record / Lombok

2

Part II

- Functional Interfaces
- Lambdas

3

Part III

- Exercise
- Conclusion

Part I

Retrospection

Review of Sorting

How did we change the sorting behavior in each case?

```
Comparator<String> byLength = new Comparator<>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
};
```

FP vs. OOP

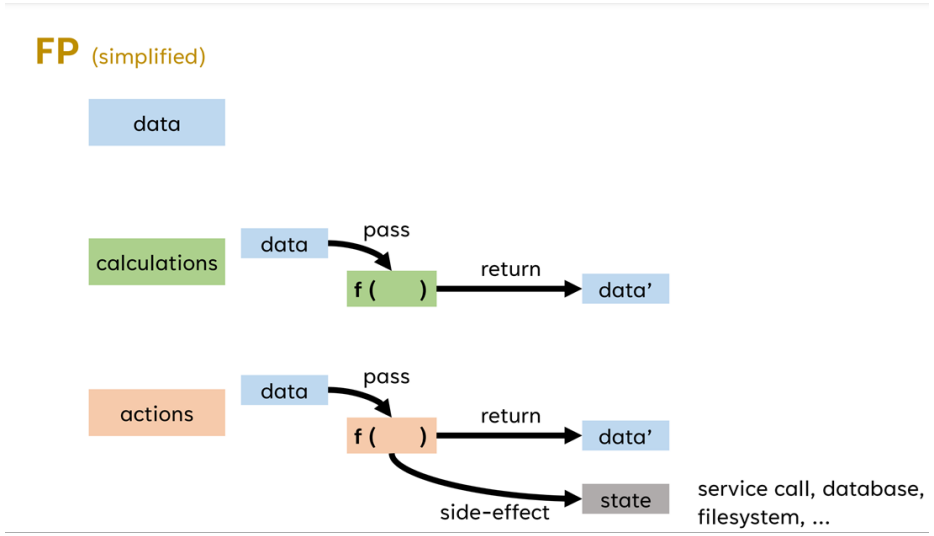
FP: Data and Operations are *separate*

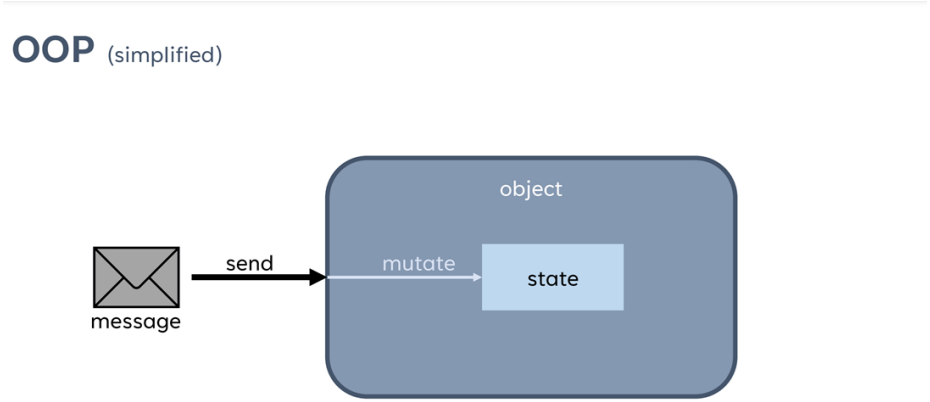
FP: Data is *immutable* (read-only)

OOP: Operations *encapsulate* Data in Object

OOP: Data is *mutable* (changeable)

Therefore: Record / Lombok





Data Classes in Java

Record

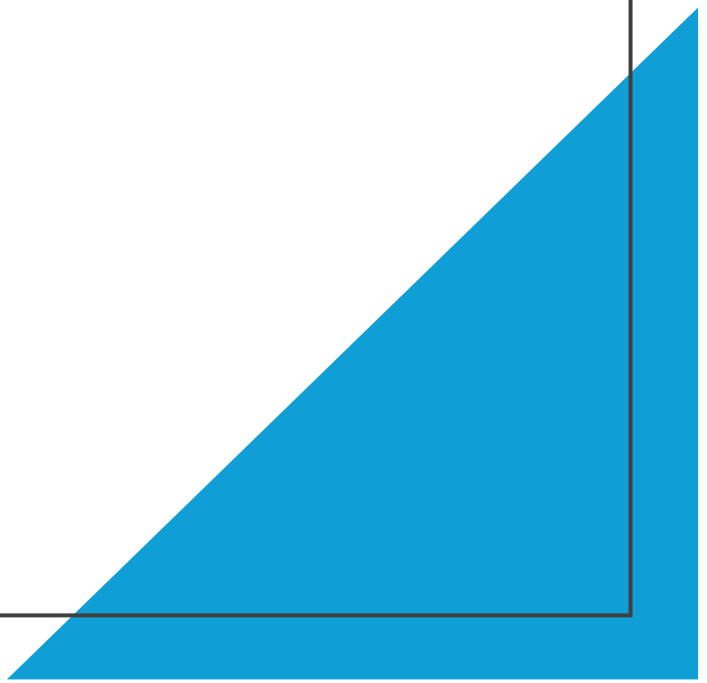
```java
import java.time.LocalDate;
import java.util.Comparator;
import java.util.List;

public class PersonSortExample {

    public record Person(String name, LocalDate birthday) {}

    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Anna", LocalDate.of(1990, 5, 12)),
            new Person("Ben", LocalDate.of(1985, 8, 23)),
            new Person("Anna", LocalDate.of(1988, 2, 3)),
            new Person("Chris", LocalDate.of(1992, 11, 7))
        );

        Comparator<Person> byNameThenBirthday = Comparator
            .comparing(Person::name)
            .thenComparing(Comparator.comparing(Person::birthday).reversed());

        people.stream()
            .sorted(byNameThenBirthday)
            .forEach(System.out::println);
    }
}
```

Lombok

```java
package ch.bbw.pr;

import lombok.Value;

import java.util.Comparator;
import java.util.Date;

/**
 * @author Peter Rutschmann
 * @version 05.11.2023
 */
@Value
public class Customer implements Comparable<Customer>{
    private String lastname;
    private String firstname;
    private Date birthdate;
    private double size;
    private String phone;

    public static Comparator<Customer> comparatorACLastNameBirthdate = new Comparator<Customer>() {
        @Override
        public int compare(Customer o1, Customer o2) {
            int value = o1.getLastname().compareTo(o2.getLastname());
            if (value == 0){
                //reversed order
                value = o2.getBirthdate().compareTo(o1.getBirthdate());
            }
            return value;
        }
    };
```

Part II

Intro: Functional Interfaces and Lambdas



Motivation

- Java is object-oriented (primarily)
- Java does not know first-class functions (in contrast to functional languages such as Haskell or even JavaScript)
- Java needs a “trick” to represent functions as objects.
- This is done via so-called *functional interfaces*.

What a Functional Interface means

```
@FunctionalInterface
interface MyFunction {
    int apply(int x);
}
```

- An interface with exactly one abstract method
- Enables „behavior as parameters “
- Tag: @FunctionalInterface (optional, but helpful)

Functional Interfaces

java.util.function

Functional Interface	Method	Purpose
`Predicate<T>`	`boolean test(T t)`	Checking Conditions
`Function<T,R>`	`R apply(T t)`	Transformation
`Consumer<T>`	`void accept(T t)`	Action on Object
`Supplier<T>`	`T get()`	Creation of a Value
`Comparator<T>`	`int compare(T a, T b)`	Comparison of two Objects

Common Signatures

Functional Interface	Signature	Example (Lambda)	Description
`Runnable`	`() -> void`	`() -> System.out.println("Hi")`	Executes code without arguments and return value
`Consumer<T>`	`(T) -> void`	`s -> System.out.println(s)`	Consumes an argument, has no return value
`Function<T, R>`	`(T) -> R`	`x -> x * x`	Converts input to output
`BiFunction<T, U, R>`	`(T, U) -> R`	`(a, b) -> a + b`	Two inputs, one output
`BiConsumer<T, U>`	`(T, U) -> void`	`(a, b) -> System.out.println(a + b)`	Two inputs, no return value
`Predicate<T>`	`(T) -> boolean`	`x -> x > 10`	Returns true or false
`BiPredicate<T, U>`	`(T, U) -> boolean`	`(a, b) -> a.equals(b)`	Two inputs, returns boolean value
`Supplier<T>`	`() -> T`	`() -> Math.random()`	Returns a value, has no input
`IntFunction<R>`	`(int) -> R`	`i -> i * i`	Primitive input, generic return type
`ToIntFunction<T>`	`(T) -> int`	`s -> s.length()`	Generic input, primitive return type
`IntPredicate`	`(int) -> boolean`	`i -> i % 2 == 0`	Primitive input, returns boolean value
`IntConsumer`	`(int) -> void`	`i -> System.out.println(i)`	Primitive input, no return value

Lambda Expressions: How?

- 💡 Kurzform für Implementierungen funktionaler Interfaces
- 📃 Allgemeine Syntax:

```
(Parameter) -> { Ausdruck oder Block }
```

- 🔍 Beispiel (statt anonymer Klasse):

```
Comparator<String> byLength =
    (s1, s2) -> Integer.compare(s1.length(), s2.length());
```

Part III

Vertiefungsübung und Abschluss

Sorting with Lambda

What interface is behind sorted(...)?

```
List<String> words = List.of("Apfel", "Banane", "Mango", "Zitrone");


words.stream()
     .sorted((a, b) -> b.compareTo(a))  // Lambda!
     .forEach(System.out::println);
```

Übung: Sortieren, Filtern, Verarbeiten

🛠️ Alleine oder zu zweit an folgenden Aufgaben arbeiten:

1. `Comparator`: nach Name, Preis, Gewicht sortieren
2. Predicate: Produkte mit Preis < 10 CHF zeigen
3. Function: nur die Namen in Grossbuchstaben zurückgeben
4. Consumer: Ausgabe formatieren
5. Bonus: Sortieren + Filtern + Ausgabe kombinieren

📄 Startercode & Datensatz sind vorbereitet.

Kahoot

play.kahoot.it

Ausblick: Streams

🔭 Alles kombinieren mit Streams

- Listen transformieren (map)
- Elemente filtern (filter)
- Sortieren, zählen, aggregieren, u.v.m.

➡️ Beispiel funktionaler Datenfluss

```
products.stream()
        .filter(p -> p.price < 10)
        .map(p -> p.name.toUpperCase())
        .forEach(System.out::println);
```