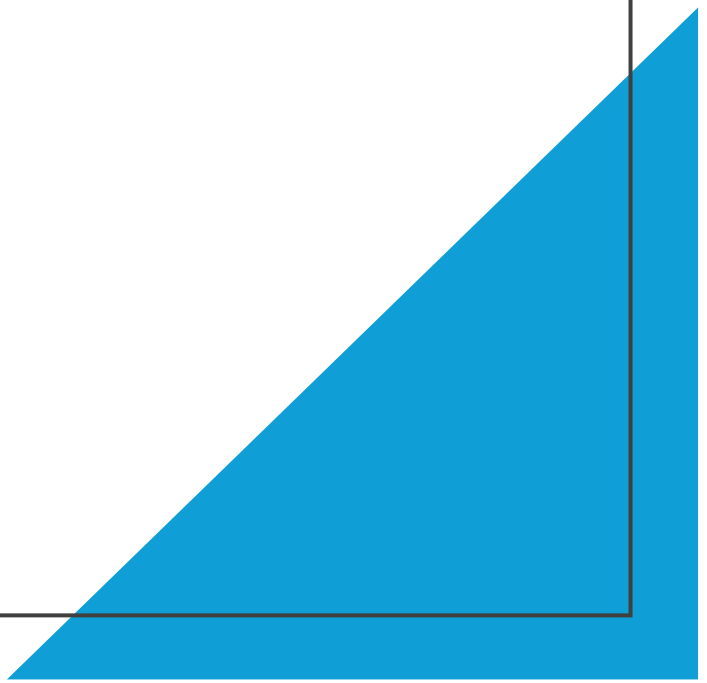
Functional Interfaces

...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

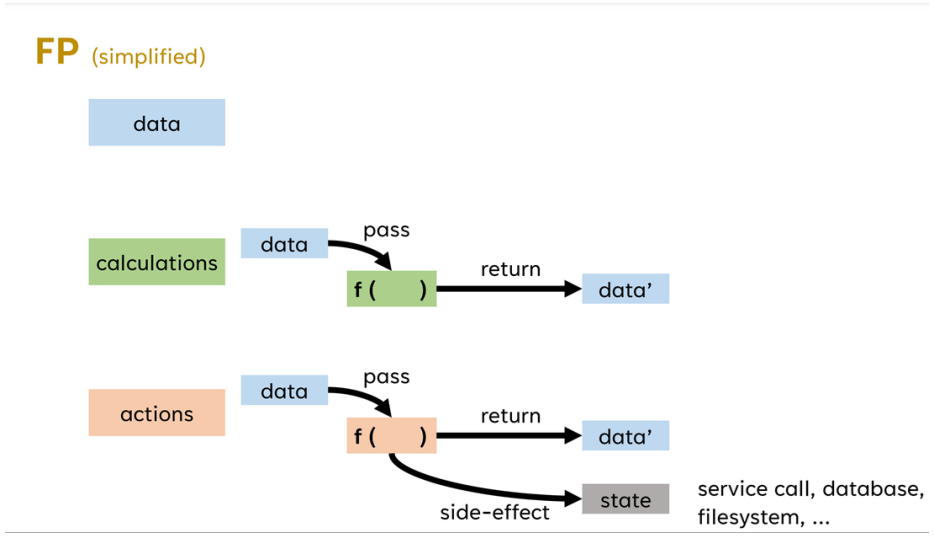
FP: Data and Operations are *separate*

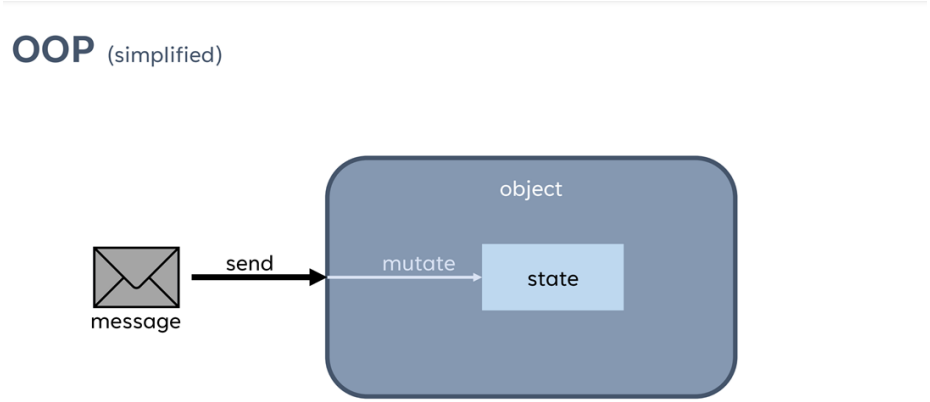
FP: Data is *immutable* (read-only)

OOP: Operations *encapsulate* Data in Object

OOP: Data is *mutable* (changeable)

Therefore: Record / Lombok





Data Classes in Java

Record

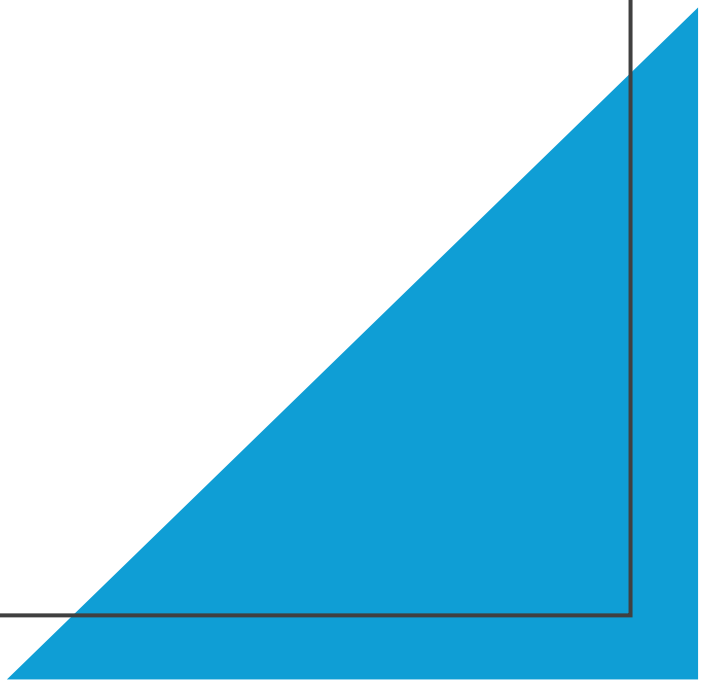
```java
import java.time.LocalDate;
import java.util.Comparator;
import java.util.List;

public class PersonRecord {

    public record Person(String name, LocalDate birthday) {}  8 usages

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

@Value
public class PersonLombok {

    String name;
    LocalDate birthday;

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

        Comparator<PersonLombok> byNameThenBirthday = Comparator
                .comparing(PersonLombok::getName)
                .thenComparing(Comparator.comparing(PersonLombok::getBirthday).reversed());

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

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

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

Musterlösung

```java
public class ProduktDemo {

    public record Produkt(String name, double preis, double gewicht) {}  13 usages

    public static void main(String[] args) {
        List<Produkt> produkte = List.of(
                new Produkt("Apfel", 2.50, 0.2),
                new Produkt("Birne", 3.00, 0.25),
                new Produkt("Zahnbürste", 12.90, 0.1),
                new Produkt("Banane", 2.20, 0.22),
                new Produkt("Schokolade", 4.80, 0.15)
        );

        // 1. Comparator: nach Name, Preis, Gewicht
        Comparator<Produkt> produktComparator = Comparator
                .comparing(Produkt::name)
                .thenComparing(Produkt::preis)
                .thenComparing(Produkt::gewicht);

        // 2. Predicate: Preis < 10 CHF
        Predicate<Produkt> billig = Produkt p -> p.preis() < 10.0;

        // 3. Function: Namen in GROSSBUCHSTABEN
        Function<Produkt, String> nameGross = Produkt p -> p.name().toUpperCase();

        // 4. Consumer: Formatierte Ausgabe
        Consumer<Produkt> ausgabe = Produkt p ->
                System.out.printf(
                        "Produkt: %-15s Preis: %5.2f CHF Gewicht: %4.2f kg%n",
                        p.name(),
                        p.preis(),
                        p.gewicht());

        // 5. Bonus: Sortieren, Filtern, Ausgabe
        System.out.println("=== Bonus: Sortieren, Filtern, Ausgabe ===");
        produkte.stream()
                .sorted(produktComparator)
                .filter(billig)
                .forEach(ausgabe);

        System.out.println("\n=== Namen in GROSSBUCHSTABEN ===");
        produkte.stream() Stream<Produkt>
                .map(nameGross) Stream<String>
                .forEach(System.out::println);
    }
}
```

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);
```