Lambda Expressions in Java

Writing Lambda Expressions with Functional Interfaces

José Paumard

PhD, Java Champion, JavaOne Rockstar

@JosePaumard | https://github.com/JosePaumard

How can you use lambda expressions efficiently?

How can you use them to improve the readability of your code?

Course Overview

How can you write lambda expressions?

How to compose them and to chain them

What are method references?

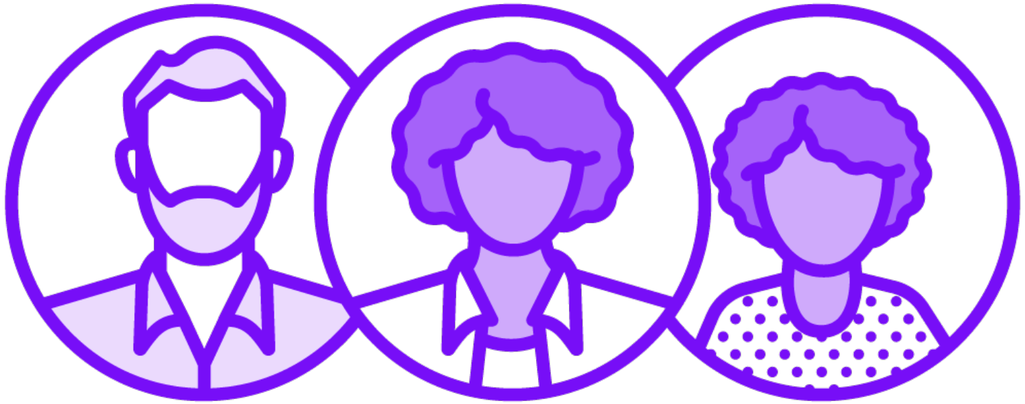
Use case: the Comparator API

Use case: analyzing a CSV file

This is a Java course

Basic knowledge of the language

And its main API: Collections, Streams



Based on Java 21

The IDE used is IntelliJ IDEA

Ultimate or Community version

Any other Java compatible IDE will do

Version Check

This course was created by using:

- Java 21
- IntelliJ IDE Ultimate Edition

Not Applicable

This course is NOT applicable to:

- Any Java version prior to 8

Relevant Notes

A note on frameworks and libraries:

- Any version of Java after 21 is fine
- Any other IDE (Eclipse, NetBeans, VS Code), will work fine

Lambda expressions are part of the Java language

You need to understand them as a Java developer

This course shows you advanced techniques to use them efficiently

Module Overview

What is the type of a lambda expression?

What is a functional interface?

Knowing its type, how can you write one?

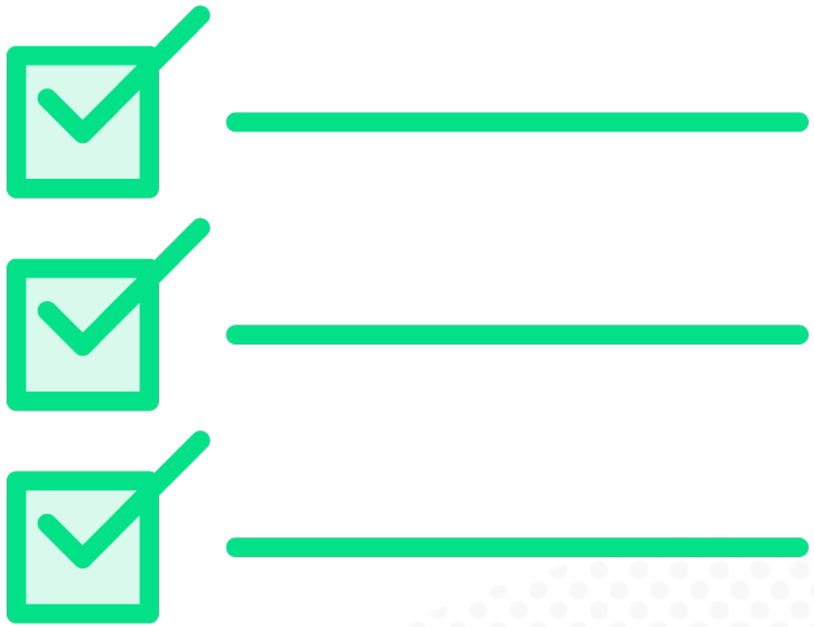
Defining the Type of Lambdas

Java is a statically typed language

Every type is known at compile time

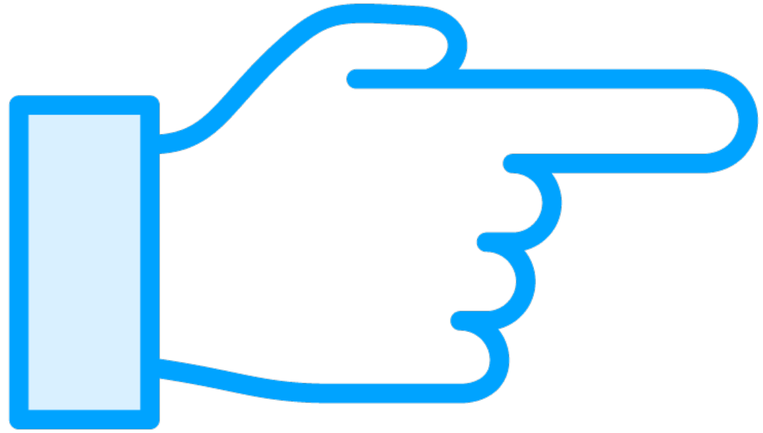
Every type has a name

And that is the case for lambda expressions



A lambda expression is typed by a functional interface

A lambda implements a functional interface



Default methods are not abstract

Static methods cannot be abstract

Abstract methods from Object do not count

```java
@FunctionalInterface
public interface Comparator<T> {

    // This is the abstract method
    int compare(T o1, T o2);

    // Several default and
    // static methods

    /**
      * This method can return true only if the specified object
      * is also a comparator and imposes the same ordering as this
      * comparator.
      */
    boolean equals(Object obj);
}
```

The Comparator Interface

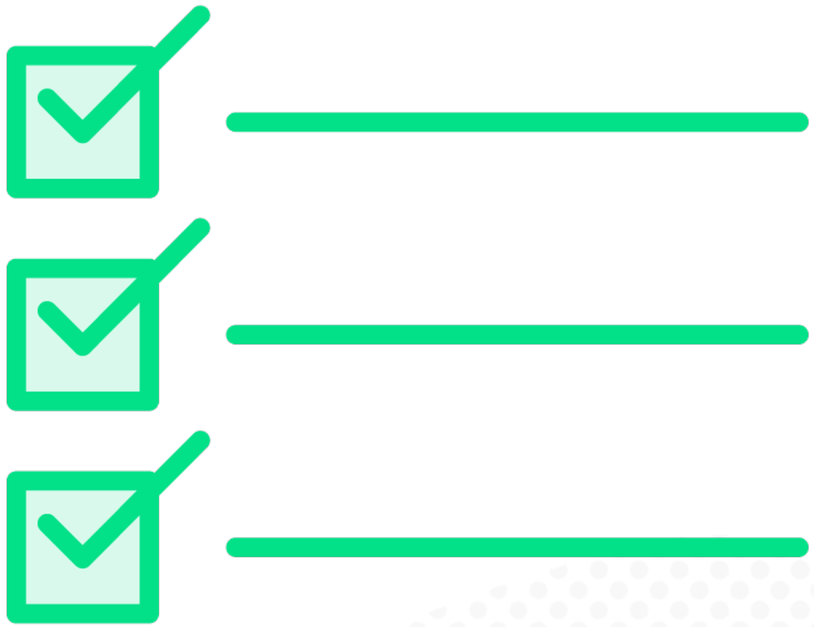
This interface has one abstract method: compare()
Several default and static methods, not shown here
And defines what it means for a comparator to be equal to another one (the JavaDoc has been simplified)

Using @FunctionalInterface

Using
@FunctionalInterface
is not mandatory

It is there to help you

For legacy reasons



Writing Lambdas in Any Situation

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>`	`s -> System.out.println(s)`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>`	`s -> s.length() > 0`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`
`Function<T, R>`	`s -> s.length()`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`
`Function<T, R>`	`s -> s.length()`
`BiFunction<T, U, R>`	`(word, sentence) -> sentence.indexOf(word)`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`
`Function<T, R>` `UnaryOperator<T>`	`s -> s.length()` `s -> s.toUpperCase()`
`BiFunction<T, U, R>`	`(word, sentence) -> sentence.indexOf(word)`

The Four Fundamental Functional Interfaces

`Supplier<T>`	`() -> "This is a message"`
`Consumer<T>` `BiConsumer<T, U>`	`s -> System.out.println(s)` `(element, sink) -> sink.accept(element)`
`Predicate<T>` `BiPredicate<T, U>`	`s -> s.length() > 0` `(i1, i2) -> i1 > i2`
`Function<T, R>` `UnaryOperator<T>`	`s -> s.length()` `s -> s.toUpperCase()`
`BiFunction<T, U, R>` `BinaryOperator<T>`	`(word, sentence) -> sentence.indexOf(word)` `(i1, i2) -> i1 + i2`

Writing Your First Lambdas

Let us write some lambdas

See the available functional interfaces

And create some

Are Lambda Expresions Objects?

You write them as objects

You can call the methods from Object on lambdas

You can serialize them

But you should refrain from doing it

Module Summary

How you can write lambdas in any situation

Supplier, Consumer, Predicate, Function

How to use @FunctionalInterface

Are lambda expressions objects?

Up Next:

Composing and Chaining Lambda Expressions