# PAO-LABORATOR 12

*2019-May-20*

# CONTENT

- Lambda expression

- Streams

# LAMBDA EXPRESSION

A *lambda expression* can be understood as a concise representation of an anonymous function that can be passed around: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown. That's one big definition; let's break it down:

- ***Anonymous—*** We say *anonymous* because it doesn't have an explicit name like a method would normally have: less to write and think about!

- ***Function—*** We say *function* because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.

- ***Passed around—*** A lambda expression can be passed as argument to a method or stored in a variable.

- ***Concise—*** You don't need to write a lot of boilerplate like you do for anonymous classes.

If you're wondering where the term *lambda* comes from, it originates from a system developed in academia called *lambda calculus*, which is used to describe computations.

# LAMBDA EXPRESSION

```
// class comparator
Comparator<Apple> byWeight = new Comparator<Apple>() {
    @Override
    public int compare(Apple o1, Apple o2) {
        return o1.getWeight().compareTo(o2.getWeight());
    }
};
```

```
// Lambda comparator
Comparator<Apple> byWeightLambda =
        (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

List<Apple> apples = new ArrayList<>();
Collections.sort(apples, byWeightLambda);
```

# LAMBDA EXPRESSION

## Example of lambda

| Use case | Examples of lambdas |
|---|---|
| A boolean expression | (List<String> list) -> list.isEmpty() |
| Creating objects | () -> new Apple(10) |
| Consuming from an object | (Apple a) -> {<br>System.out.println(a.getWeight());<br>} |
| Select/extract from an object | (String s) -> s.length() |
| Combine two values | (int a, int b) -> a * b |
| Compare two objects | (Apple a1, Apple a2) -><br>a1.getWeight().compareTo(a2.getWeight()) |

# LAMBDA EXPRESSION

- @FunctionalInterface

If you explore the new Java API, you'll notice that functional interfaces are annotated with `@FunctionalInterface`

This annotation is used to indicate that the interface is intended to be a functional interface. The compiler will return a meaningful error if you define an interface using the `@FunctionalInterface` annotation and it isn't a functional interface.

# LAMBDA EXPRESSION

Comparator interface in java 8

```
 * @param <T> the type of objects that may be compared by this comparator
 *
 * @author  Josh Bloch
 * @author  Neal Gafter
 * @see Comparable
 * @see java.io.Serializable
 * @since 1.2
 */
@FunctionalInterface
public interface Comparator<T> {
    /**
     * Compares its two arguments for order.  Returns a negative integer,
     * zero, or a positive integer as the first argument is less than, equal
     * to, or greater than the second.<p>
     *
```

# LAMBDA EXPRESSION

| Functional interface | Function descriptor | Primitive specializations |
|---|---|---|
| Predicate<T> | T -> boolean | IntPredicate,<br>LongPredicate,<br>DoublePredicate |
| Consumer<T> | T -> void | IntConsumer,<br>LongConsumer,<br>DoubleConsumer |
| Function<T, R> | T -> R | IntFunction<R>,<br>IntToDoubleFunction,<br>IntToLongFunction,<br>LongFunction<R>,<br>LongToDoubleFunction,<br>LongToIntFunction,<br>DoubleFunction<R>,<br>ToIntFunction<T>,<br>ToDoubleFunction<T>,<br>ToLongFunction<T> |

# LAMBDA EXPRESSION

| Functional interface | Function descriptor | Primitive specializations |
|---|---|---|
| Supplier<T> | () -> T | BooleanSupplier,<br>IntSupplier,<br>LongSupplier,<br>DoubleSupplier |
| UnaryOperator<T> | T -> T | IntUnaryOperator,<br>LongUnaryOperator,<br>DoubleUnaryOperator |
| BinaryOperator<T> | (T, T) -> T | IntBinaryOperator,<br>LongBinaryOperator,<br>DoubleBinaryOperator |
| BiPredicate<L, R> | (L, R) -> boolean | |
| BiConsumer<T, U> | (T, U) -> void | ObjIntConsumer<T>,<br>ObjLongConsumer<T>,<br>ObjDoubleConsumer<T> |
| BiFunction<T, U, R> | (T, U) -> R | ToIntBiFunction<T, U>,<br>ToLongBiFunction<T, U>,<br>ToDoubleBiFunction<T, U> |

# LAMBDA EXPRESSION

- Type inference

There are no restriction, you can use it to make it more clear what you are implementing

```
// Lambda comparator
Comparator<Apple> byWeightLambda =
        (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());


// Lambda comparator
Comparator<Apple> byWeightLambdaNoInference =
        (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

# LAMBDA EXPRESSION

Lambda expressions can access:

- Instance variables.
- Effectively final method parameters.
- Effectively final local variables.

“Effectively final” means that if you could add the **final** modifier to a local variable, that variable is considered effectively final.

# LAMBDA EXPRESSION

Method references help to point to methods using their names.

A method reference is described using **::** symbol.

```
x -> System.out.println(x);
```

Is equivalent with:

```
System.out::println
```

# LAMBDA EXPRESSION

## Method references

| Type | Example | Syntax |
|---|---|---|
| Reference to a static method | ContainingClass::staticMethodName | Class::staticMethodName |
| Reference to a constructor | ClassName::new | ClassName::new |
| Reference to an instance method of an arbitrary object of a particular type | ContainingType::methodName | Class::instanceMethodName |
| Reference to an instance method of a particular object | containingObject::instanceMethodName | object::instanceMethodName |

# LAMBDA EXPRESSION

## Function interface

```
@FunctionalInterface
public interface Function<T, R>

R apply(T t);
```

```
Function<String, Integer> func = Integer::parseInt;
List<String> strings = new ArrayList<>();

strings.stream().forEach(x -> func.apply(x));

strings.stream().forEach(Integer::parseInt);
```

# LAMBDA EXPRESSION

**Optional** type - A container object which may or may not contain a non-null value.

- An Optional is created using a factory.
- You can either request an empty Optional or pass a value for the Optional to wrap.

# LAMBDA EXPRESSION

```
10: public static Optional<Double> average(int... scores) {
11:     if (scores.length == 0)
12:         return Optional.empty();
13:     int sum = 0;
14:     for (int score : scores)
15:         sum += score;
16:     return Optional.of((double) sum / scores.length);
}
```

Line 12 returns an empty Optional when we can’t calculate an average.

Lines 13, 14 and 15 add up the scores.

Line 16 creates an Optional to wrap the average.

System.out.println(average(90, 100));

System.out.println(average());

# LAMBDA EXPRESSION

## Optional methods

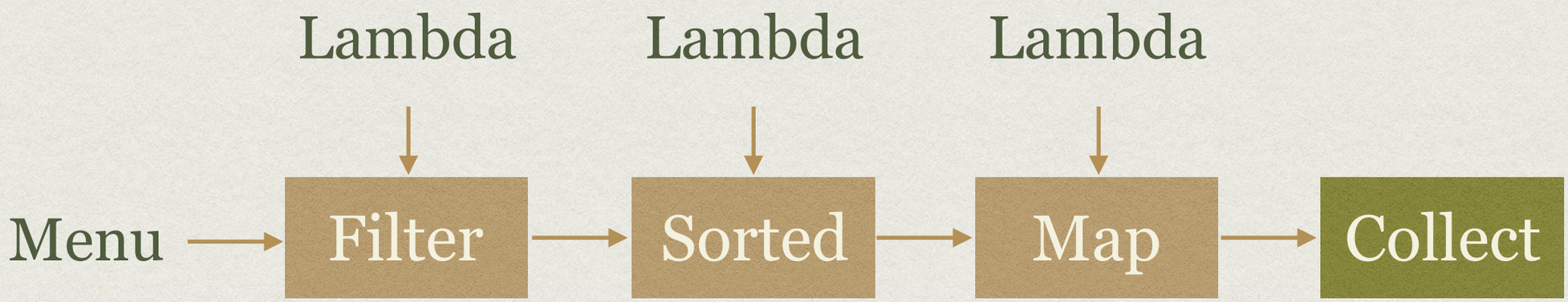
| Method | When Optional Is Empty | When Optional Contains a Value |
|---|---|---|
| get() | Throws an exception | Returns value |
| ifPresent(Consumer c) | Does nothing | Calls Consumer c with value |
| isPresent() | Returns false | Returns true |
| orElse(T other) | Returns other parameter | Returns value |
| orElseGet(Supplier s) | Returns result of calling supplier | Returns value |
| orElseThrow(Supplier s) | Throws an exception created by calling Supplier | Returns value |

# STREAMS

- Streams allow us to write code:
- Declarative - concise and readable
- Composable - greater flexibility
- Parallelizable - better performance

# STREAMS

- Chaining stream operations to form a stream pipeline



# STREAMS

To summarise, working with streams in general involves three items:

- A *data source* (such as a collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

# STREAMS

## Intermediate operations:

| Operation | Type | Return type | Argument of the operation | Function descriptor |
|---|---|---|---|---|
| filter | Intermediate | Stream<T> | Predicate<T> | T -> boolean |
| map | Intermediate | Stream<R> | Function<T, R> | T -> R |
| limit | Intermediate | Stream<T> | | |
| sorted | Intermediate | Stream<T> | Comparator<T> | (T, T) -> int |
| distinct | Intermediate | Stream<T> | | |

# STREAMS

## Terminal operations:

| Operation | Type | Purpose |
|---|---|---|
| forEach | Terminal | Consumes each element from a stream and applies a lambda to each of them. The operation returns void. |
| count | Terminal | Returns the number of elements in a stream. The operation returns a long. |
| collect | Terminal | Reduces the stream to create a collection such as a List, a Map, or even an Integer. |