Programmeertechnieken/Programming Techniques

Part 3
Functional programming

Koen Pelsmaekers
Campus Groep T, 2022-2023

Content of this chapter

- Functional Programming: introduction
- Functional Programming in Java
    - Lambda expressions
        - Functional interfaces
    - Streams
    - Hands on exercises

Introduction to functional programming

KU LEUVEN

Functional vs. Imperative Programming

- Functional programming: “what” (declarative)
  - “pure functions” (based on λ-calculus)
    - Result of function call only relies on input parameters
    - No variables => no changing state
    - No side effects
    - Result of function can be used as input of other function
  - Easier to parallelize
  - Treat functions (= code) as “data”
    - Assign it to a variable
    - Higher-order functions: pass a function as an argument and/or use a function as a return value
- Imperative programming: “how”
  - Von Neumann architecture (aka “pigeon hole model”)
    - Variables to store and retrieve values => state changes
    - Function can have “side effects”

Functional programming: advantages

- More readable code
- Less verbose code
- Higher level of abstraction: focus on “what” instead of on “how”
- No side effects

=> More maintainable code, more “pleasant” to write

- Easier to introduce parallelism
- Improve the use of the (Collection) libraries
- Not: more performant programs

Functional programming languages

- Lisp (1960)
- ML (1975)
- Erlang (1987) (Ericsson)
- Haskell (1992)
- Scala (2003)
- Clojure (2007) (Lisp dialect on JVM)
- …

Most of these languages are “impure” (= with more or less imperative language constructs)

Functional additions to non-functional languages

- C
  - Pointer to function (C, 1972)
- C#
  - Lambda expressions, delegate types, … (C# version 3, 2007)
- Kotlin
  - Function type (2011)
- C++
  - Lambdas, std::function<>, … (C++11, 2011)
- Java
  - Lambda expressions, Streams, Functional interface, …(Java 8, 2014)
- JavaScript (ECMAScript)
  - First-class functions (1995), arrow functions (ECMAScript 6, 2015)
- PHP
  - Arrow Functions (PHP 7.4, 2019)
- …

Example: internal vs. external iteration

Imperative (= external iteration)

```
for each element in the collection {
 get the element;
 apply some code to the element;
}
```

Functional (= internal iteration)

```
collection.doForEachElement(some code);
```

λ: code as argument

Functional programming in Java

Java 8: lambda expressions

- Brings functional programming to Java
  - Pre-Java 8: Java = imperative + object-oriented
  - Since Java 8 (2014): Java += functional
  - Introduced with @FunctionalInterface type for lambdas
    - Interface with exactly one abstract method
    - Lambda can implement (the one abstract method of) a Functional interface
      - Can replace anonymous inner classes
    - Package: `java.util.function`
  - Streams
    - Package: `java.util.stream`

Lambda expression

Method without a name, used to pass around behaviour as if it were data.

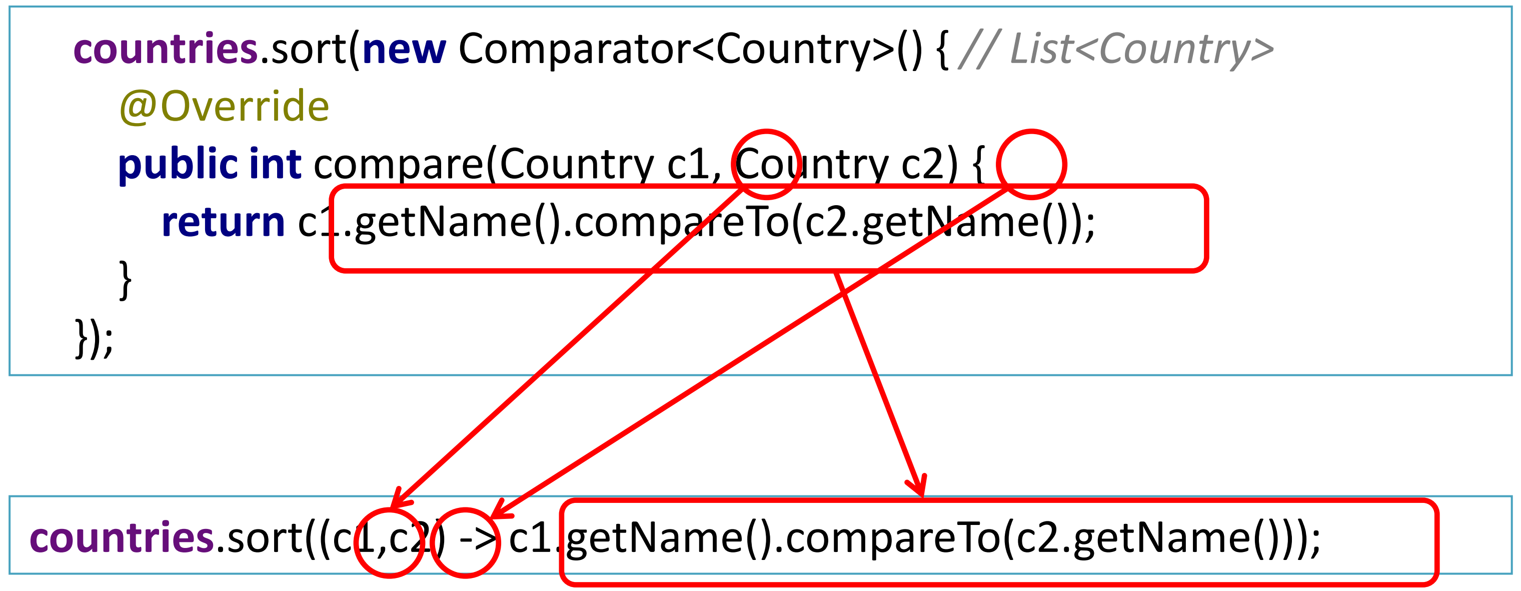
Lambda expression: examples

```
IntFunction<Integer> integerIntFunction = (int x) -> 2 * x;

IntBinaryOperator intBinaryOperator1 = (int x, int y) -> x + y;
IntBinaryOperator intBinaryOperator2 = (int x, int y) -> {
    int max = x > y ? x : y;
    return max;
};

Supplier<String> stringSupplier = () -> "Hello World!";

Consumer<String> stringConsumer1 = (String msg) -> System.out.println(msg);
Consumer<String> stringConsumer2 = msg -> System.out.println(msg);

Function<String, Integer> stringIntegerFunction = (String str) -> str.length();
```

Lambda expression: syntax

Argument list	Arrow token	Body
(int x, int y)	->	return x + y
()	->	42
(String s)	->	{System.out.println(s);}
s	->	s.toUpper();

Lambda expression example: Comparator interface



```
countries.sort(new Comparator<Country>() { // List<Country>
    @Override
    public int compare(Country c1, Country c2) {
        return c1.getName().compareTo(c2.getName());
    }
});
```

```
countries.sort((c1,c2) -> c1.getName().compareTo(c2.getName()));
```

Java compiler knows the signature of the single abstract function: "compiler inference"
Other examples of **@FunctionalInterface**: Runnable, Consumer, Predicate, …

Lambda expression: background

- Functional interface
  - Interface with only one abstract method
    - @FunctionalInterface
    - Methods of class Object and static methods in the interface do not count
    - “default” implementations do not count
  - java.util.function package (since Java 8, see API documentation)
    - Contains many pre-defined functional interfaces (see next slides)
  - Examples
    - ActionListener, Runnable, Comparator, Consumer, Predicate, …
- Before Java 8: anonymous inner class
  - Create class in place (see Comparator example)
  - Verbose!

Hands on exercises part 1…

Get code snippets from git: https://gitlab.groept.be/koen.pelsmaekers/t2vpt_2021_2022coursedemos.git

Exercise 1

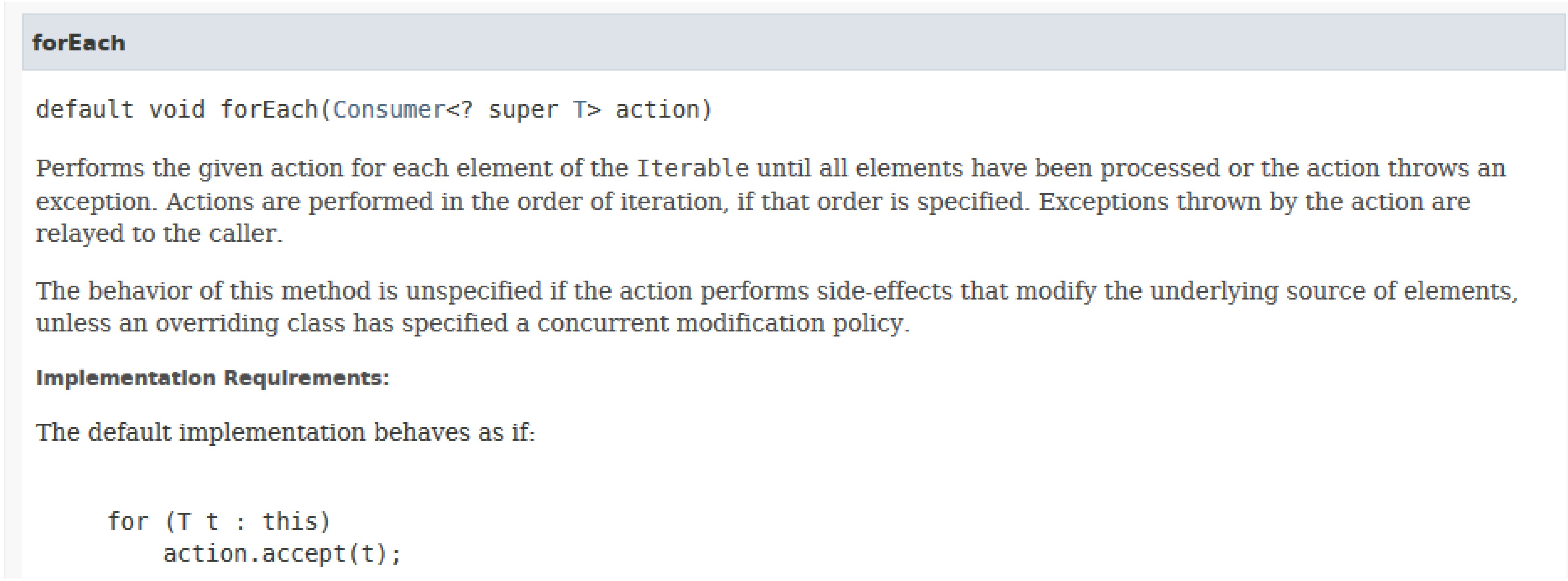
a. Print the names of the capitals of the Belgian provinces in upper case (Brussels != province capital):
   - Use external iteration
   - Use internal iteration & lambda expression
   - (do not use the overridden toString() method of the List's implementations, but use an iteration)

b. Remove the names of the capitals with more than 5 characters
   - Use external iteration (iterator() & remove())
   - Use a "remove" method in the Collection interface & lambda expression

```java
private List<String> capitals = new ArrayList<>(Arrays.asList(
        "Brugge", "Gent", "Antwerpen", "Hasselt", "Leuven", "Mons",
        "Namur", "Wavre", "Liege", "Arlon"));
```

forEach: under the hood

- Iterable interface



forEach

```
default void forEach(Consumer<? super T> action)
```

Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

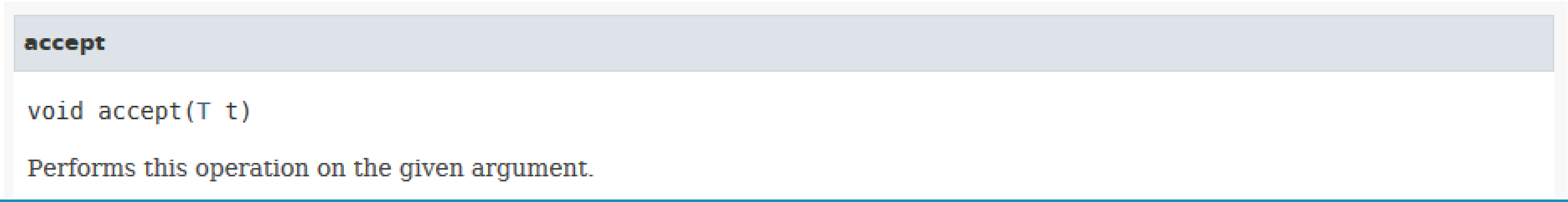
The behavior of this method is unspecified if the action performs side-effects that modify the underlying source of elements, unless an overriding class has specified a concurrent modification policy.

Implementation Requirements:

The default implementation behaves as if:

```
     for (T t : this)
         action.accept(t);
```

- Consumer interface (@FunctionalInterface)



accept

```
void accept(T t)
```

Performs this operation on the given argument.

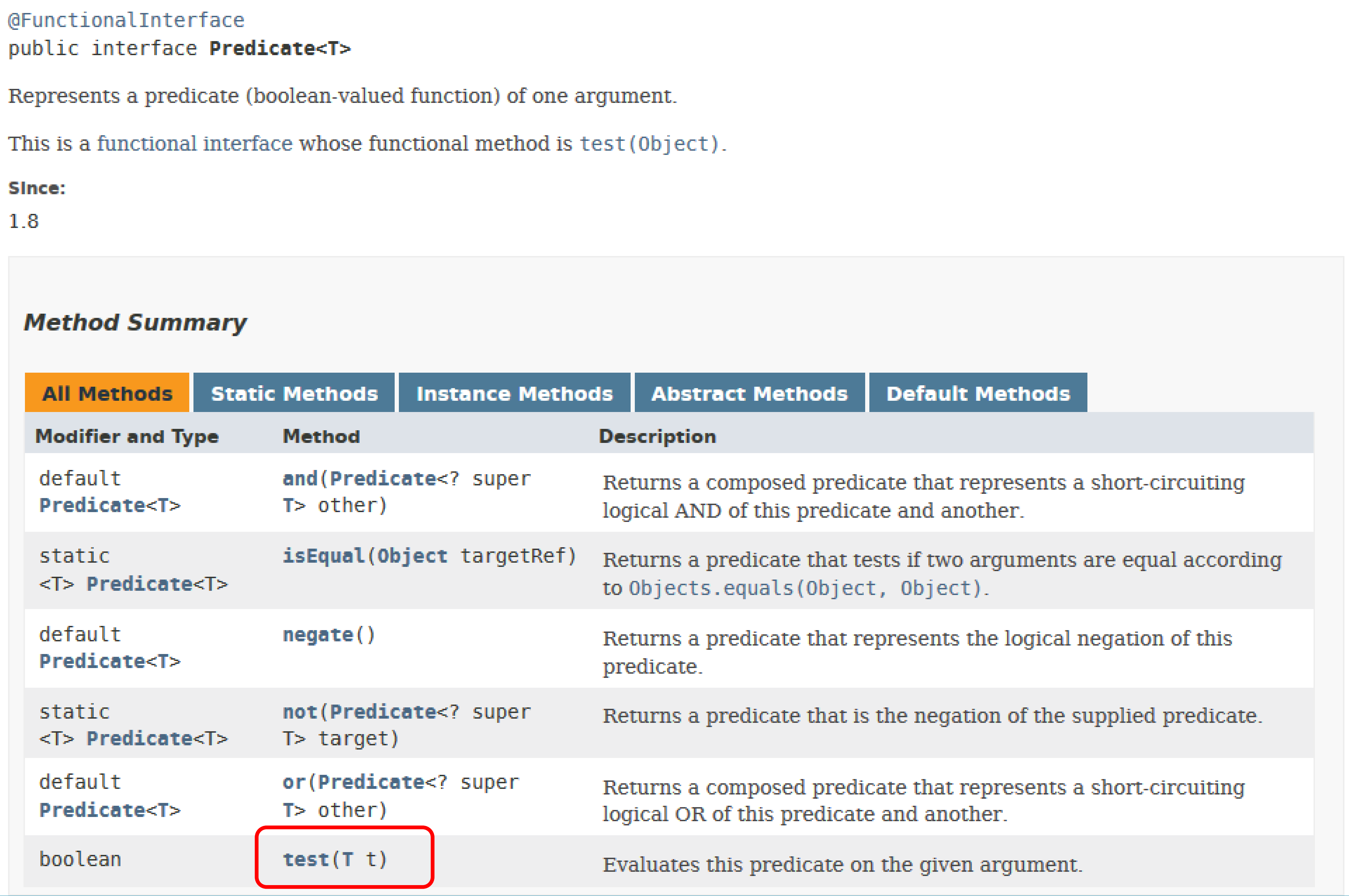
forEach: example

```
capitals.forEach(new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
});
```

```
capitals.forEach(s -> System.out.println(s));
```

removeIf: implementation

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```



```
@FunctionalInterface
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:
1.8

Method Summary

Modifier and Type	Method	Description
default Predicate<T>	and(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	isEqual(Object targetRef)	Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.
default Predicate<T>	negate()	Returns a predicate that represents the logical negation of this predicate.
static <T> Predicate<T>	not(Predicate<? super T> target)	Returns a predicate that is the negation of the supplied predicate.
default Predicate<T>	or(Predicate<? super T> other)	Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	test(T t)	Evaluates this predicate on the given argument.

Java.util.function

- Pre-defined functional interfaces: some examples

Consumer<T>	void accept(T t)
BiConsumer<T,U>	void accept(T t, U u)
Function<T,R>	R apply(T t)
BiFunction<T,U,R>	R apply(T t, U u)
Predicate<T>	boolean test(T t)
BiPredicate<T,U>	boolean test(T t, U u)
Supplier<T>	T get()
…	…

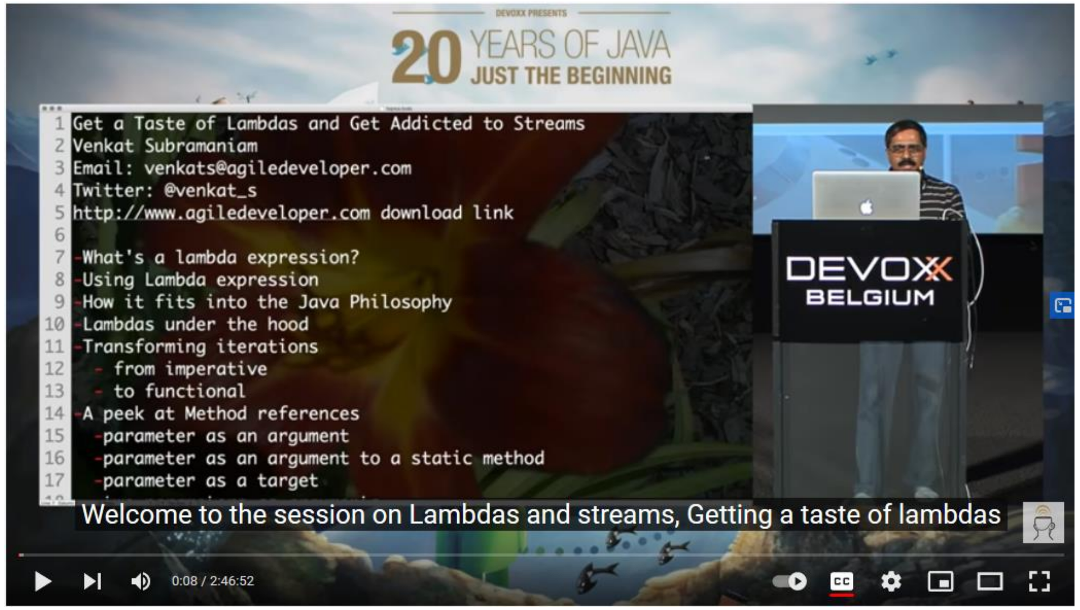
Java.util.function

- Specialized versions for int, double, long (example with “double”)

DoubleConsumer	void accept(double value)
DoubleFunction<R>	R apply(double value)
DoublePredicate	boolean test(double value)
DoubleSupplier	double getAsDouble()
DoubleToIntFunction	int applyAsInt(double value)
…	…

“Get a Taste of Lambdas and Get Addicted to Streams”

- Venkat Subrimaniam, popular speaker
  (see next slide)
- http://www.agiledeveloper.com/
- Devoxx 2015, deep dive session:
  https://www.youtube.com/watch?v=1OpAgZvYXLQ



```
Get a Taste of Lambdas and Get Addicted to Streams
Venkat Subramaniam
Email: venkats@agiledeveloper.com
Twitter: @venkat_s
http://www.agiledeveloper.com download link

-What's a lambda expression?
-Using Lambda expression
-How it fits into the Java Philosophy
-Lambdas under the hood
-Transforming iterations
  - from imperative
  - to functional
-A peek at Method references
  -parameter as an argument
  -parameter as an argument to a static method
  -parameter as a target
```

Welcome to the session on Lambdas and streams, Getting a taste of lambdas

- If you are not able to view the full video, view at least these parts:
  - 01:19:50 – 01:27:22 (parallelization)
  - 02:04:47 – 02:19:30 (lazy streams)

Appetizer: Venkat’s remarkable/funny quotes…

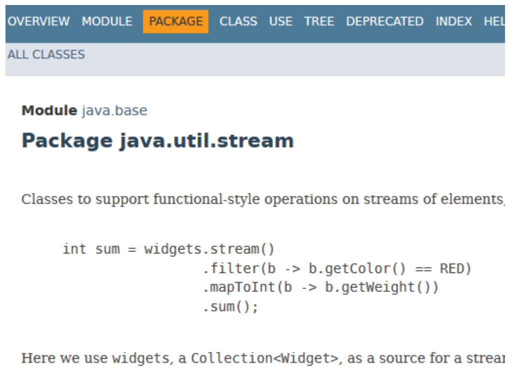
- "When you have 9 million programmers using your language and out of which 1 million programmers know where you live, you have to decide things differently."
- "If a language does not support backward compatibility, it is DOOMED; we also know if a language supports backward compatibility it's also DOOMED!... and so it's a question really choosing which way you like to be DOOMED!"
- "If you iterate through a collection of integers, in your wildest imagination, can you guess what you will pull out of the collection, if you take out an element?"
- "They think that WE Programmers are antisocial, but WE are not. We are absolutely social with the right kind of people.“
- “Call daddy”

Streams

Venkat:
“lambdas are cool, but streams are awesome”

- Supports “functional-style” operations on streams of elements
    - filter-map-reduce
- A stream:
    - Is an object on which you can define operations (± pipeline)
    - Does not hold any data
    - Will not modify the data it processes (no shared mutability)
    - Will process data in “one pass”
    - Is built on highly optimized algorithms, and that can work in parallel

Streams



Module java.base

Package java.util.stream

Classes to support functional-style operations on streams of elements,

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Here we use widgets, a Collection<Widget>, as a source for a strea

- Interface in java.util.stream
  - “A sequence of elements supporting sequential and parallel aggregate operations.”
  - Read the package information in the Java API
- (Specialized) streams + specialized operations
  - Stream<T> + IntStream, LongStream, DoubleStream
- Stream pipeline
  - Source
  - **Intermediate** operation(s) (lazy, Stream as return value)
  - **Terminal** or “aggregate” operation (eager, “they trigger the computation”)
- Parallel stream

Optional<T>

- A container (“box”) with or without a non-null value
  - isPresent() – isEmpty() – orElse()
- Introduced for use as a method return type with “no result”
  - Introduced with Streams
    - For instance: result of findFirst() on an empty stream or average() on an empty IntStream
  - Is null a value?
- Package: java.util
- Specialized optionals:
  - OptionalInt, OptionalLong, OptionalDouble

```
OD od ..
if (od.isPresent()) { } ← getDouble
else { }
```

Optional<T>: example

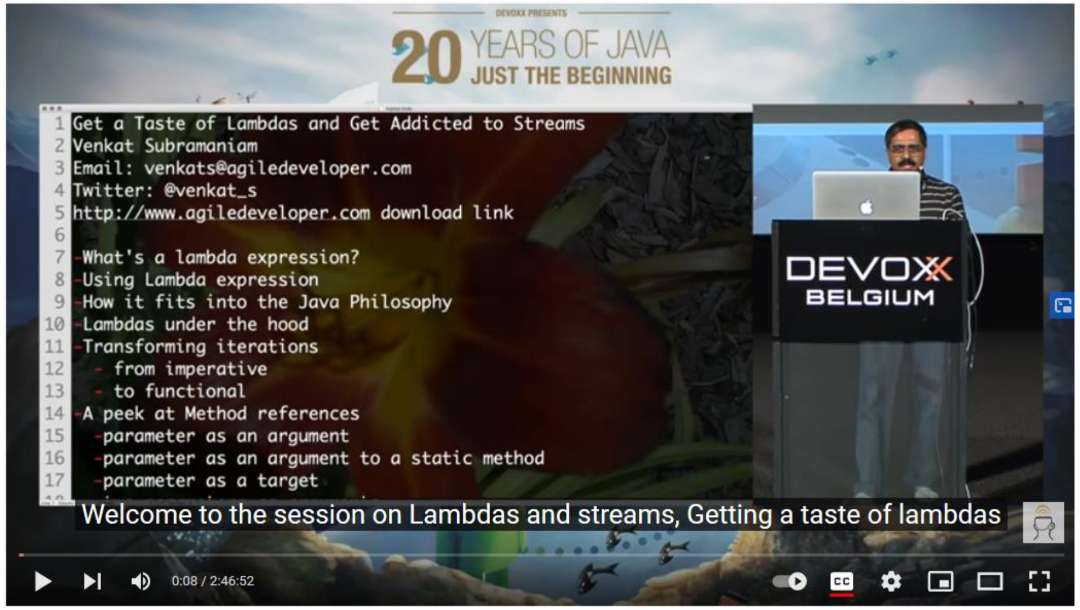
```
Optional<String> first = people.stream()
        .filter(p -> p.getWeight() < 60.5)
        .map(Person::getName)
        .findFirst();

System.out.println(first.orElse("no one"));
```

“Get a Taste of Lambdas and Get Addicted to Streams”

- Venkat Subrimaniam, popular speaker

- http://www.agiledeveloper.com/
- Devoxx 2015, deep dive session:
  https://www.youtube.com/watch?v=1OpAgZvYXLQ

02:04:47 – 02:19:30 (lazy streams)



```
Get a Taste of Lambdas and Get Addicted to Streams
Venkat Subramaniam
Email: venkats@agiledeveloper.com
Twitter: @venkat_s
http://www.agiledeveloper.com download link

-What's a lambda expression?
-Using Lambda expression
-How it fits into the Java Philosophy
-Lambdas under the hood
-Transforming iterations
  - from imperative
  - to functional
-A peek at Method references
  -parameter as an argument
  -parameter as an argument to a static method
  -parameter as a target
```

Welcome to the session on Lambdas and streams, Getting a taste of lambdas

Some stream methods

Method	Intermediate/ terminal	Comment
forEach(Consumer c)	Terminal	Performs action on every element; lambda expression as instance of Consumer interface
map(Function f)	Intermediate	Apply a mapping function to a stream of values and create another stream
filter(Predicate p)	Intermediate	Filters out elements
reduce(T id, BinaryOperator b)	Terminal	Reduction on the elements of a stream (see next slide)
max/min(Comparator c)	Terminal	Returns the maximum/minimum element of a stream according the provided Comparator as an Optional
flatMap(Function, Stream)	Intermediate	“Flattens” a stream of streams into one stream; a one-to-many transformation (see next slide)

Hands on exercises part 2…

Get code snippets from git: https://gitlab.groept.be/koen.pelsmaekers/t2vpt_2021_2022coursedemos.git

Exercise 2

- Ropepulling team exercises
  (use Person, Gender and RopePullingDemo)
  a) Find the total weight of a rope pulling team
  b) Find the weight of the heaviest person in a rope pulling team
  c) Find the heaviest person in a rope pulling team
  d) Find the lightest female person in a rope pulling team
  e) …

“reduce” operation

- Reduction operation
  - Collapses a stream to one result
- T reduce(T identity, BinaryOperator<T> accumulator)

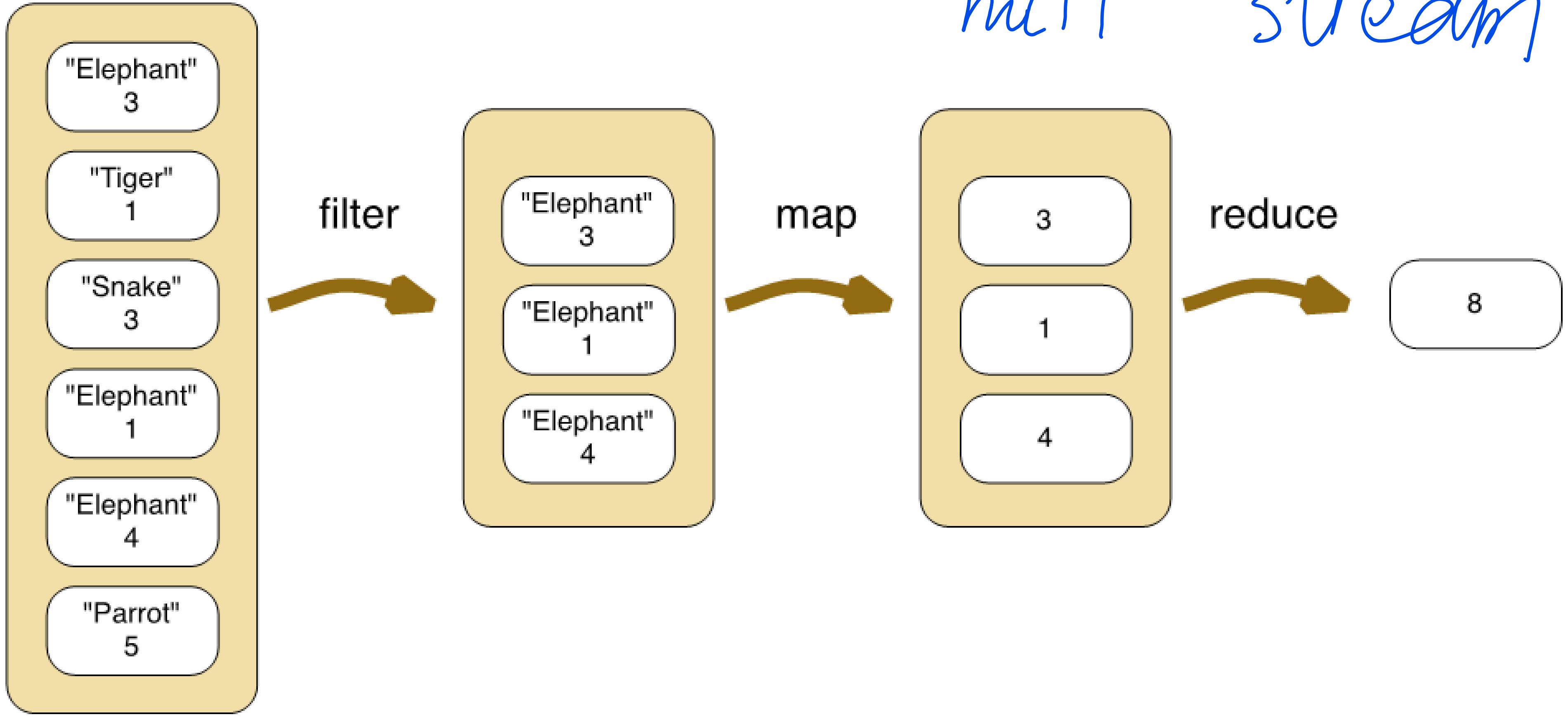
```
T result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

```
Integer sum = integers.reduce(0, (a, b) -> a+b);
```

- sum, min, max, average, string concatenation: special cases of reduction

Filter-map-reduce example

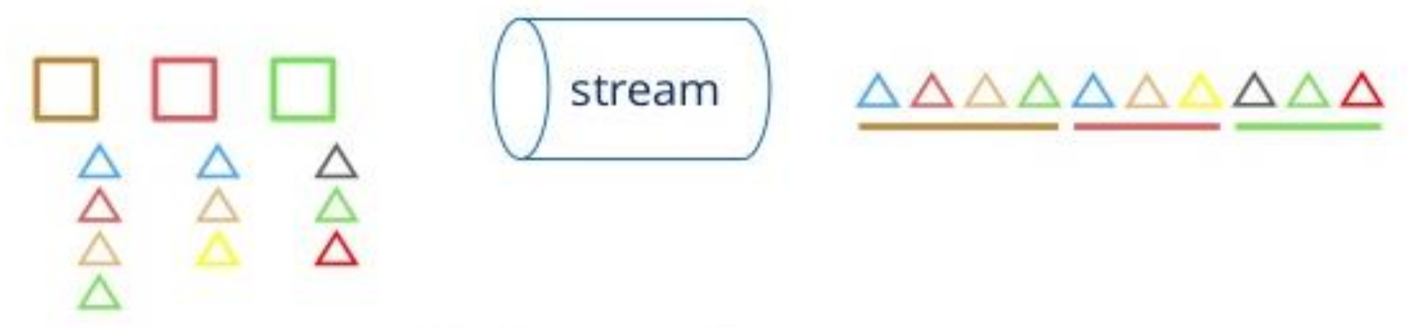
OptionalDouble for null stream



filter(*name is elephant*).map(*count*).reduce(*add up*)

Taken from: Objects First with Java, 6th edition

flatMap operation



Source: José Paumard, https://www.slideshare.net/jpaumard/collectors-in-the-wild

- Example use case: “Give a list (set?) of all programming languages known by a team of software developers, given this definition of Developer”:

```
private List<Developer> team;
```

```
public class Developer {
   private String name;
   private Set<String> programmingLanguages;
   ...
}
```

Overview of stream methods

return a stream

- Intermediate operations
  - Stream<T> distinct()
  - Stream<T> filter(Predicate)
  - Stream<R> flatMap(Function)
  - …Stream flatMapTo…(Function)*[*]*
  - Stream<T> limit(long)
  - Stream<R> map(Function)
  - …Stream mapTo…(Function)*[*]*
  - Stream<T> peek(Consumer)
  - Stream<T> skip(long)
  - Stream<T> sorted()
  - Stream<T> parallel()
  - …

[*] … = Int, Long, Double

- Terminal operations
  - boolean allMatch(Predicate)
  - boolean anyMatch(Predicate)
  - R collect(Collector)
  - long count()
  - Optional<T> findAny()
  - Optional<T> findFirst()
  - void forEach(Consumer)
  - Optional<T> max(Comparator)
  - Optional<T> min(Comparator)
  - boolean noneMatch(Predicate)
  - …

Overview of specialized stream methods
(IntStream [*]) (+ specialized version of methods of previous slide)

- Intermediate operations
  - DoubleStream asDoubleStream()
  - LongStream asLongStream()
  - Stream<Integer> boxed()
  - DoubleStream mapToDouble(*IntToDoubleFunction*)
  - LongStream mapToLong(*IntToLongFunction*)
  - <U> Stream<U> mapToObj(*IntFunction*)
  - IntStream range()
  - IntStream rangeClosed()
  - …

- Terminal operations
  - OptionalDouble average()
  - OptionalInt max()
  - OptionalInt min()
  - int sum()
  - IntSummaryStatistics summaryStatistics()
  - toArray()
  - …

[*] … = same for DoubleStream and LongStream

Some operation characteristics

- Intermediate operations
  - Operations without a buffer: stateless operations
    - map(), filter(), flatMap()
  - Operations with an internal buffer: stateful operations
    - sorted(), distinct()
  - Operations that need to keep track of the index
    - limit(), skip()
- Terminal operations
  - Operations that need to consume all data
    - foreach(), count(), max(), min(), reduce(), toArray()
  - Short-circuit operations
    - allMatch(), anyMatch(), noneMatch(), findFirst(), findAny()

they continue

they wait

Hands on exercises part 3…

Get code snippets from git: https://gitlab.groept.be/koen.pelsmaekers/t2vpt_2021_2022coursedemos.git

Exercise 3: Lotto numbers check

- Can you solve this small exercise?
  - A lotto number generator has to generate 6 different numbers between 1 and 45 (both included)
  - These numbers are stored in this data structure: int[6] numbers
    - See lab session 8 of Object-oriented Software Development
  - Write code to check if
    1. For every number: 1 <= number <= 45
    2. All numbers are different
  - Use streams and lambda expressions (you do not have to write the code to generate the numbers as it was an exercise in a previous programming course, but you can try it…)

"Get a Taste of Lambdas and Get Addicted to Streams"

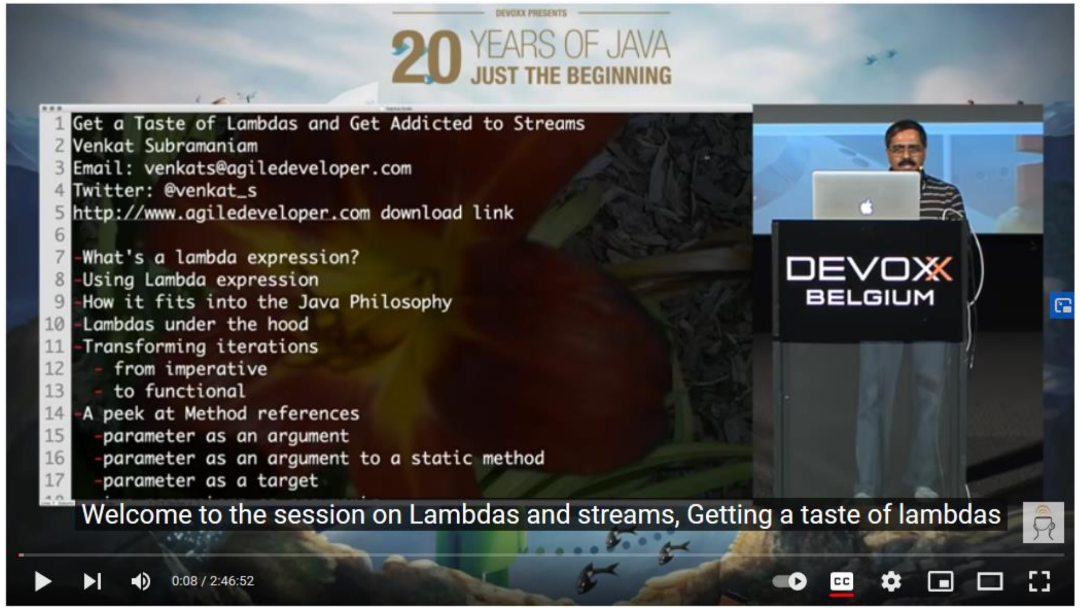
- Venkat Subrimaniam, popular speaker

- http://www.agiledeveloper.com/
- Devoxx 2015, deep dive session:
  https://www.youtube.com/watch?v=1OpAgZvYXLQ

01:19:50 – 01:27:22 (parallelization)

parallelStream()

or .parallel()



```
Get a Taste of Lambdas and Get Addicted to Streams
Venkat Subramaniam
Email: venkats@agiledeveloper.com
Twitter: @venkat_s
http://www.agiledeveloper.com download link

-What's a lambda expression?
-Using Lambda expression
-How it fits into the Java Philosophy
-Lambdas under the hood
-Transforming iterations
  - from imperative
  - to functional
-A peek at Method references
  -parameter as an argument
  -parameter as an argument to a static method
  -parameter as a target
```

Welcome to the session on Lambdas and streams, Getting a taste of lambdas

Parallel stream

- Easiest way to start multiple threads
  => multiple cores
- Overhead
- Conditions
  - Long operation and/or big collection
  - Collection is easy to divide in parts

```
Consumer<String> consumer = System.out::print;

Arrays.asList("a", "b", "c", "d", "e", "f", "g")
        .parallelStream()
        .forEach(consumer);

// output: edfgbca
```

```
Arrays.asList("a", "b", "c", "d", "e", "f", "g")
        .stream()
        .forEach(consumer);

// output: abcdefg
```

Method reference: “::”

- Shorthand notation for a lambda expression with only one method
  - Reference to a method, not executing the method
- Makes code more readable, for instance: “System.out::println”, “Person::getAge”
- Types of method references:
  1. Reference to a static method (lambda parameter as argument)
  2. Reference to an instance method (lambda parameter is object)
  3. Reference to a constructor (Classname::new)
- Examples: next slide
- See also: Venkat Subramanian, “Get a Taste of Lamdas and Get Addicted to Streams”, 37:00 - …

Method reference: examples

```
IntStream.range(100, 120)
      .forEach(n -> System.out.println(n));
IntStream.range(100, 120)
      .forEach(System.out::println);
```

```
OptionalDouble average1 = team.stream()
      .mapToInt(p -> p.getAge())
      .average();

OptionalDouble average2 = team.stream()
      .mapToInt(Person::getAge)
      .average();
```

Collector interface

- Mutable reduction operation
  - Creates a new result container: supplier()
  - Accumulates elements into a mutable result container: accumulator()
  - Combines multiple result containers into one (parallel): combiner()
  - Optionally transforms it into a final representation: finisher()
- For example: accumulating elements in a collection, concatenating strings, computing summary information, apply grouping, …
- Collectors can take a collector to produce a new collector
- Many implementations in the Collectors class (next slide)

Some Collector implementations in Collectors

- To accumulate elements into a collection
  - toList(), toSet(), toMap(), toCollection()
- To concatenate strings
  - joining()
- To summarize elements according some criterium
  - summingInt(), averagingDouble(), counting(), maxBy(), minBy(), summarizingDouble(), …
- To group elements, to partition elements
  - groupingBy(), partitioningBy()
- …

more examples: see later

Hands on exercises part 4…

Get code snippets from git: https://gitlab.groept.be/koen.pelsmaekers/t2vpt_2021_2022coursedemos.git

Exercise 4

- Prepare for some exercises
  1) Create 3 “model” classes: Team, Player and Racket
  2) Create a “main” class: BadmintonExercise
  3) Copy their implementation from git
  4) Implement the method getAge() in Player (hint: use Period)
  5) Start with the exercises (next slide); you can implement them in the go() method of the BadmintonExercise class

Exercise 5

- Write code, using streams and lambda expressions, to:
  a) Count the number of teams from “Belgium”
  b) Print the names of all the players of a team (f.i. “Poona”)
  c) Print the names of all the players of a team whose name starts with the character “J”
  d) Collect the different rankings of all players for a given team in a Set (or in a List); try with and without using “distinct()”
  e) Do the same for the players older than 25
  f) Do the same for the players older than 25 of all the teams
  g) Give a Set of brand names of rackets of all the players older than 25

Exercise 5 (cont.)

- Write code, using streams and lambda expressions, to:
  - h) Give the weight of the heaviest racket
  - i) Give the racket with the heaviest weight
  - j) Calculate some statistics (average, minimum, maximum, …) of the weight of all rackets
  - k) Give statistics of the age of all the players
  - l) Find the first player older than 25 with three rackets (look in the Stream API for appropriate methods)
  - m) Give the statistics (sum, average, count, …) of the weight of the badminton rackets of the players with the character ‘e’ in their name of the teams from Belgium

Collecting data in maps

- Three versions of Collectors.toMap()
  - toMap(keyMapper, valueMapper)
  - toMap(keyMapper, valueMapper, mergeFunction)
  - toMap(keyMapper, valueMapper, mergeFunction, mapSupplier)
- Exercises

  “Give for each gender the oldest person of that gender”, “Count the number of people per gender”, …

Grouping data

- Comparable to “group by” clause in SQL
- Three versions of groupingBy

  Collector

  - groupingBy(classifier)
  - groupingBy(classifier, downstream)
  - groupingBy(classifier, mapFactory, downstream)
- Can be nested
- Exercises

  “Group people by gender in a list”, “Group names of people by gender in a comma seperated String”, “Count the number of people by gender”, “Group people by gender and by month of birth”, …

Partitioning data

- Partitioned in two groups based on a Predicate, in a map with Boolean keys
- Special case of groupingBy
- Two versions of partitioningBy
  - partitioningBy(predicate)
  - partitioningBy(predicate, downstream)

Collector

- Exercises

  "Partition people by a certain age", "Count people by a certain age", "List of names of people by a certain age", …