Software Development

Lambdas and streams

Koen Pelsmaekers

Unit Informatie (GT 03.14.05)

email: koen.pelsmaekers@kuleuven.be

Java 8: lambda expressions

Brings functional programming to Java

- pre-Java 8: `Java = imperative & object-oriented`
- since Java 8: `Java += functional`
- use functions (= code) as "data"
  - assign to a variable
  - pass as argument
- Introduced with @FunctionalInterface
  - interface with exactly one abstract method
  - packages: `java.util.function/java.util.stream`
- Importance
  - more readable code, less verbose, higher level of abstraction; "what" vs."how"
  - improve the (use of the) Collection libraries
  - introduce parallellism
  - not more performant programs

Imperative vs. Functional (pseudo-code)

Imperative (= external iteration)

```
for each element in the collection {
   get the element;
   do something with the element;
}
```

Functional (= internal iteration)

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

λ: code as argument

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 knows the function name, the arguments, …: compiler inference
Other examples: Runnable, Consumer, Predicate, ... = Functional Interface
@FunctionalInterface

Lambda syntax

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

Background

- Anonymous inner class
  - create class in place (cf. Comparator example)
  - verbose
- Functional interfaces
  - 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
  - examples
    - ActionListener, Runnable, Comparator, Consumer
  - java.util.function
    - contains many pre-defined functional interfaces (see later)
- Lambda = method without a name, used to pass around behavior as if it were data

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)`
  - `BinaryOperator<T>------ T apply(T t1, T t2)`*[*]*
  - `Predicate<T>----------- boolean test(T t)`
  - `BiPredicate<T,U>------- boolean test(T t, U u)`
  - `Supplier<T>------------ T get()`
  - `BooleanSupplier-------- boolean getAsBoolean()`
  - …

+ specialized versions for int, long, double

[*] BinaryOperation is a specialisation of BiFunction

Exercises

- Print the names of the capitals of the Belgian provinces in upper case (Brussels != province capital): (do not use the overridden toString() method of the List's!)
  - use external iteration
  - use internal iteration & lambda expression
- Remove the names with more than 5 characters
  - find the appropriate method in the Java API

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

Get code snippets from http://studev.groept.be/scratchbook/

external/internal iteration: solution (1)

```java
// print all the capitals in uppercase: external iteration

for (String capital: capitals) {
    System.out.println(capital.toUpperCase());
}
// other external iterations: while, for(;;), do..while, iterator


// print all the capitals in uppercase: internal iteration

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

external/internal iteration: solution (2)

```
// remove all the capitals with more than 5 characters
// external iteration

Iterator<String> it = capitals.iterator();
while (it.hasNext()) {
    String capital = it.next();
    if (capital.length() > 5) {
        it.remove();
    }
}

// remove all the capitals with more than 5 characters
// removeIf (internal iteration)

capitals.removeIf(c → c.length() > 5);
```

Under the hood: forEach

Iterable interface

Modifier and Type	Method and Description
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.

```
default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
}
```

Consumer interface (= Functional interface)

Modifier and Type	Method and Description
void	accept(T t) Performs this operation on the given argument.

```
capitals.forEach(c -> System.out.print(c));
```

Consumer in detail

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


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

Under the hood: 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;
}
```

Predicate<T>: @FunctionalInterface

```
@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

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and 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.
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)`
  - `BinaryOperator<T>------ T apply(T t1, T t2)`*[*]*
  - `Predicate<T>----------- boolean test(T t)`
  - `BiPredicate<T,U>------- boolean test(T t, U u)`
  - `Supplier<T>------------ T get()`
  - `BooleanSupplier-------- boolean getAsBoolean()`
  - …

+ specialized versions for int, long, double

[*] BinaryOperation is a specialisation of BiFunction

Switch to

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

Venkat Subramaniam, Devoxx 2015

Some of Venkat's remarkable and 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."

Devoxx talk “Get a Taste of Lambdas and Get Addicted to Streams”
https://www.youtube.com/watch?v=1OpAgZvYXLQ

Website: http://www.agiledeveloper.com/

To be discussed...

- Optional
- Stream operations
  - FlatMap operation
  - min()/max()
- Specialized streams
  - int, double, long
- Exercises
  - Scratchbook: https://studev.groept.be/scratchbook/

All other examples will be published, together with the slides on Toledo: lectures, part 6.

Streams

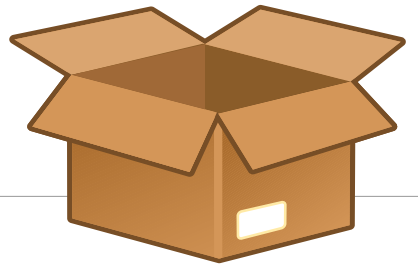
- A stream is:
  - an object on which I can define operations (± pipeline)
  - that does not hold any data
  - that will not modify the data it processes
  - that will process data in « one pass »
  - that is built on higly optimized algorithms, and that can work in parallel
- `stream()`: part of collections interface
  - See also: introduction of default methods
- Example
  - print the capitals with <= 5 characters in uppercase using a stream
  - find the name of the “fattest” person in a list
    - use Gender.java, Person.java and RopePullingDemo.java
  - make sum of weights of persons in a list

→ External iteration vs. internal iteration: for-loop vs. forEach

Streams (cont.)

- interface in java.util.stream:
  - "A sequence of elements supporting sequential and parallel aggregate operations"
- Specialized streams
  - IntStream, LongStream, DoubleStream, Stream<T>
- Stream pipeline
  - source
  - **Intermediate** operation(s) (lazy)
  - **Terminal** or "aggregate" operation (eager)
- Parallel stream
  - see: ParallelStreamTest example

class Optional<T>



- Introduced to deal with "no data" return value
  - is null a value?
- "box" with or whithout a value
- specialized Optional: OptionalInt, ...
- Example: "find the heaviest person in a (empty) list"

```
Optional<Person> fattest = people
            .stream()
            .max(Comparator.comparingDouble(Person::getWeight));

if (fattest.isPresent())
    System.out.println("present");
else
    System.out.println("not present");

System.out.println(fattest.map(Person::getName).orElse("no one"));
```

Some stream methods

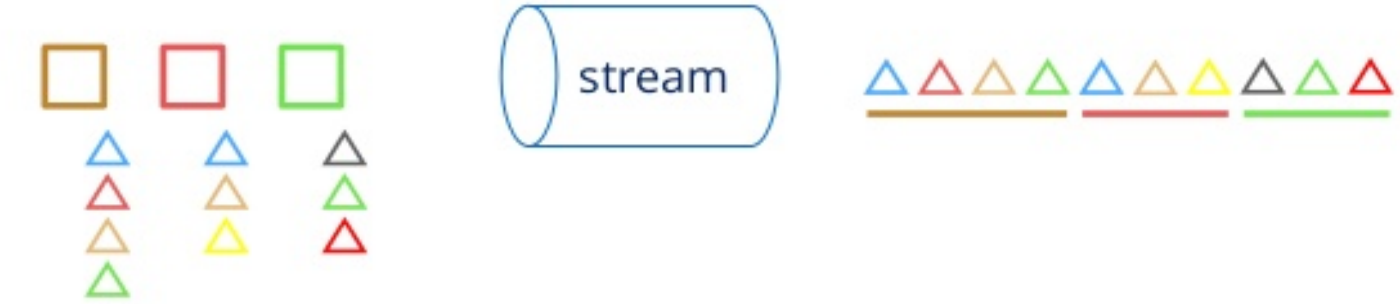
- forEach(Consumer)/forEachOrdered(Consumer)
  - terminal operation, perfoms action on every element (non-deterministic order/deterministic order)
  - lambda expression: instance of Consumer
- collect(toList)
  - terminal operation, generates list from values in stream
- map(Function)
  - intermediate operation, apply a mapping function to a stream of values and create another stream
  - lambda expression: instance of Function
- filter(Predicate)
  - intermediate operation, filters out elements
  - lamda expression: instance of Predicate (boolean result)

Some stream methods (cont.)

- flatMap(Function, Stream)
  - intermediate operation; "flattening" a stream of streams into one stream; one-to-many transformation (next slide)
- max/min(Comparator)
  - terminal operation; returns max/min of stream elements according to a Comparator
  - returns "Optional" (to deal with an empty stream)
- reduce(T, BinaryOperator)/reduce(BinaryOperator)
  - terminal operation

```
Object accumulator = initialValue;
for (Object element: collection) {
   accumulator = combine(accumulator, element);
}
```

FlatMap operation



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

Exercise
"Give a list of all the programming languages known by a team of software developers (unique values only), given this definition of Developer" (use Developer.java):

```
public class Developer {
   private String name;
   private Set<String> languages;
   …
}
```

Overview of stream methods

- Terminal operations
  - boolean allMatch(*Predicate*)
  - boolean anyMatch(*Predicate*)
  - long count()
  - Optional<T> findAny()
  - Optional<T> findFirst()
  - void forEach(*Consumer*)
  - Optional<T> max(*Comparator*)
  - Optional<T> min(*Comparator*)
  - boolean noneMatch(*Predicate)*
  - ...
- Intermediate operations
  - Stream<T> distinct()
  - Stream<T> filter(*Predicate*)
  - flatMap(*Function*)
  - ...Stream flatMapTo...(*Function*)[*]
  - Stream<T> limit(*long*)
  - <R> Stream<R>map(*Function*)
  - ...Stream mapTo...(*Function*)[*]
  - Stream<T> skip(*long*)
  - Stream<T> sorted()
  - ...

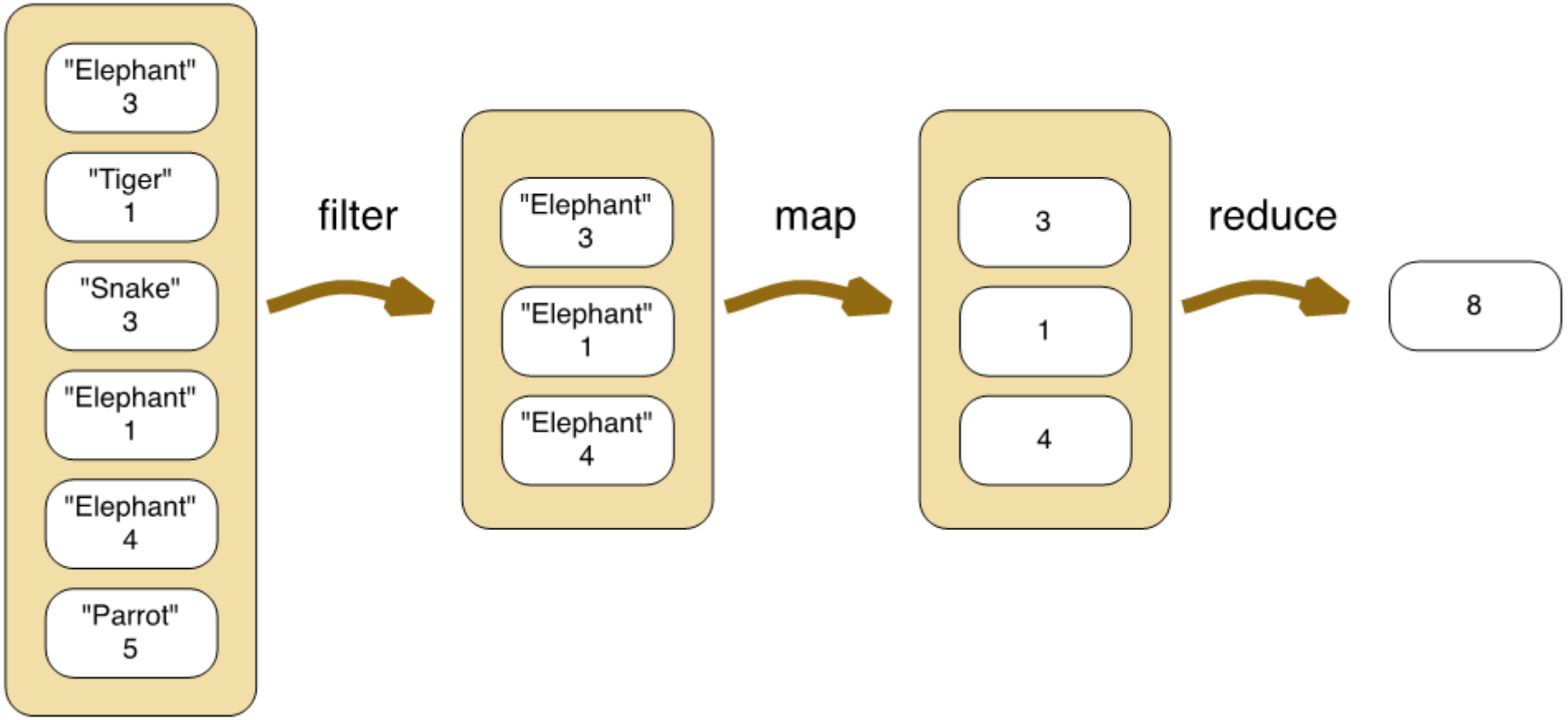
[*]... = Int, Long, Double

Overview of methods on specialized stream (f.i. IntStream)

- Terminal operations
  - OptionalDouble average()
  - OptionalInt max()
  - OptionalInt min()
  - int sum()
  - IntSummaryStatistics summaryStatistics()
  - toArray()
  - ...
- Intermediate operations
  - DoubleStream asDoubleStream()
  - LongStream asLongStream()
  - Stream<Integer> boxed()
  - DoubleStream mapToDouble(*IntToDoubleFunction*)
  - LongStream mapToLong(*IntToLongFunction*)
  - <U> Stream<U> mapToObj(*IntFunction*)
  - IntStream range()
  - IntStream rangeClosed()
  - ...

Same for DoubleStream and LongStream

Pipeline of stream functions: typical scenario




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

Taken from: Objects First with Java, 6th edition

Reduce

- Reduction operation (terminal operation)
  - collapses a stream to one result
  - accumulator should be associative (parallel!)

```
(((a * b) * c) * d) = ((a * b) * (c * d))
```

- API definition

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

- Implementation

lambda expression

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

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

Method reference

- shorthand notation for Lambda expression with only one method
- code becomes more readable
- four types of method references:
  - reference to a static method
  - reference to an instance method of a particular object
  - reference to an instance method of an arbritary object of a particular type
  - reference to a constructor
- Venkat Subramanian, "Lambdas are cool, but streams are awesome", 00:37:00 - ...

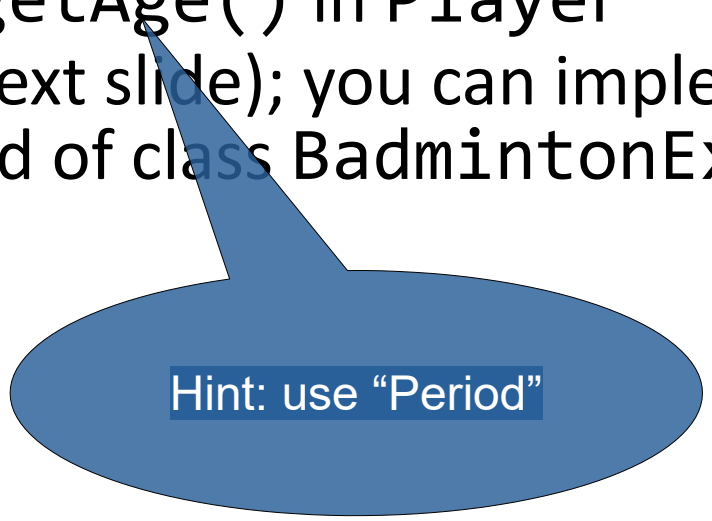
Some method reference examples

```
flags.stream()
        .map(b -> String.valueOf(b))
        .forEach(s → System.out.println(s));

flags.stream()
        .map(String::valueOf)
        .forEach(System.out::println);
```

```
artists.stream()
        .map(a -> a.getName()))
        .forEach(s → System.out.println(s));

flags.stream()
        .map(Artist::getName)
        .forEach(System.out::println);
```

Hands-on

- Prepare for some exercises
  1) create 3 classes: Team, Player and Racket
  2) create a Main class: BadmintonExercise
  3) copy their implementation from: http://studev.groept.be/scratchbook
  4) implement the method `getAge()` in `Player`
  5) start with the exercise (next slide); you can implement them in the `go()` method of class `BadmintonExercise`



Exercises

1) Count the number of teams from "Belgium"
2) Print the names of all the players of a team (f.i. Poona)
3) Print the names of all the players of a team whose name starts with the character "J"
4) Give the different rankings of all players for a given team in a Set (or in a List); try with and without using "distinct()"
5) Do the same for the players older than 25
6) Do the same for the players older than 25 of all the teams

Excercises (cont.)

7) Give a Set of brand names of rackets of all the players older than 25
8) Give the weight of the heaviest racket
9) Give the racket with the heaviest weight
10) Calculate some statistics (average, minimum, maximum, ..) of the weight of all rackets
11) Give statistics of the age of all the players
12) Find the first player older than 25 with three rackets (look in the Stream API for appropriate functions)