Why I ❤ streams and you should too!

by Patryk Drabiński, 24.05.2020

Who am I?

- Java Developer (3 years of exp)

- Learned streams API on a job

- Fell in ❤ with streams

Agenda

1. Presentation (what and why)

2. Practical Training
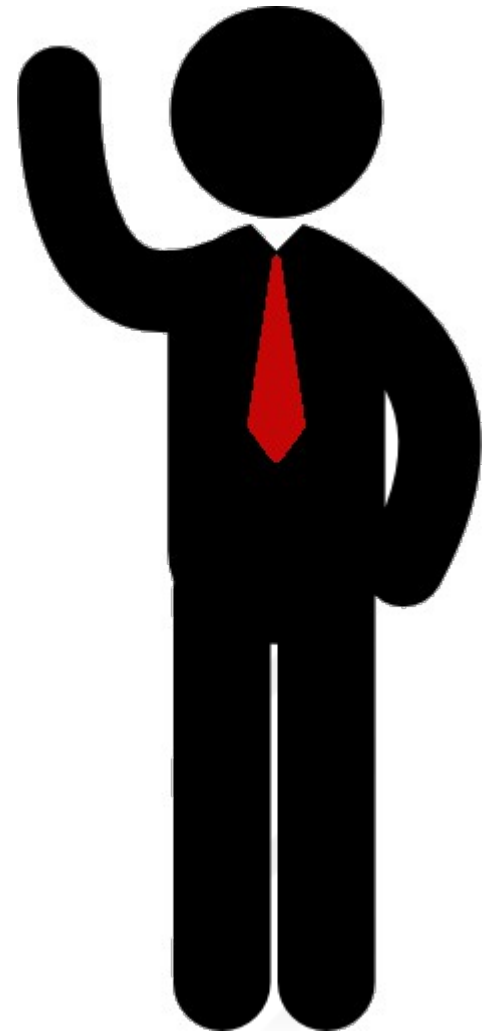
3. Wrap-up

Presentation

Streams look scary

```
people.stream()
    .map(person -> person.getName())
    .filter(name -> name.startsWith(''A''))
    .collect(Collectors.toList())
```

But they are not!

Story time
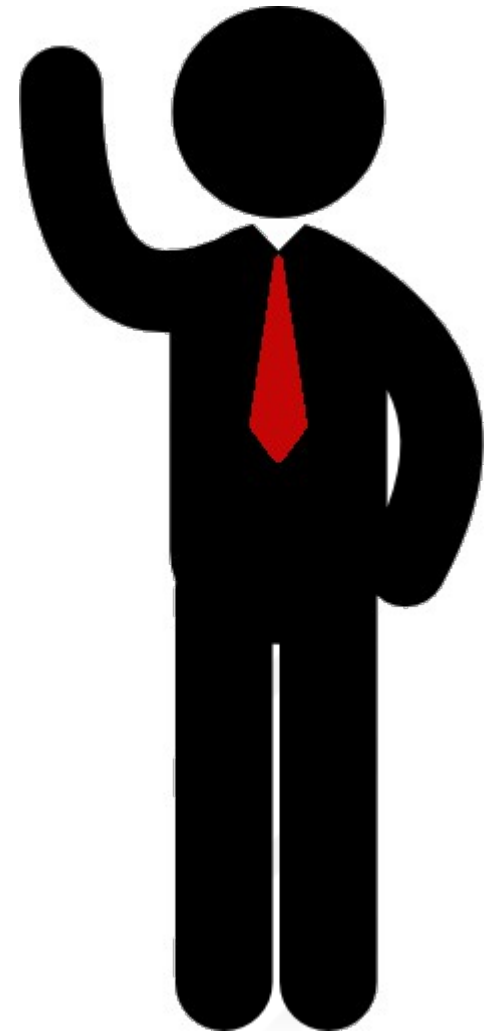
- Find all female employees

Story time

Function:

```
result = new ArrayList<>();
for (Person person: people) {
    if (person.isFemale()) {
        result.add(person)
    }
}

return result;
```

Story time

- Find all female employees
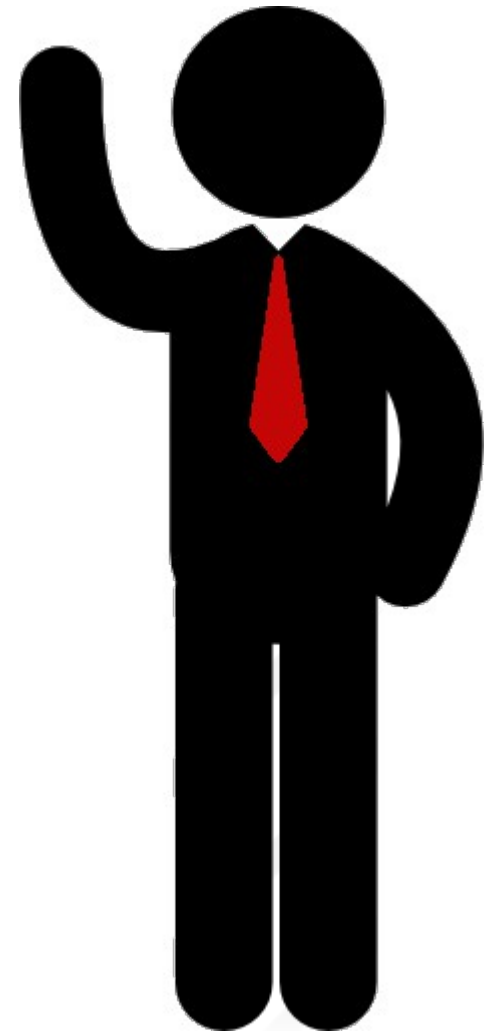
- Find all employees over 50

Story time

Another function:

```
result = new ArrayList<>();
for (Person person: people) {
    if (person.getAge() > 50) {
        result.add(person)
    }
}

return result;
```

Story time

- Find all female employees

- Find all employees over 50

- Find all employees with salary over 70 000

Story time

Another function:

```
result = new ArrayList<>();
for (Person person: people) {
    if (person.getSalary() > 70000) {
        result.add(person)
    }
}

return result;
```
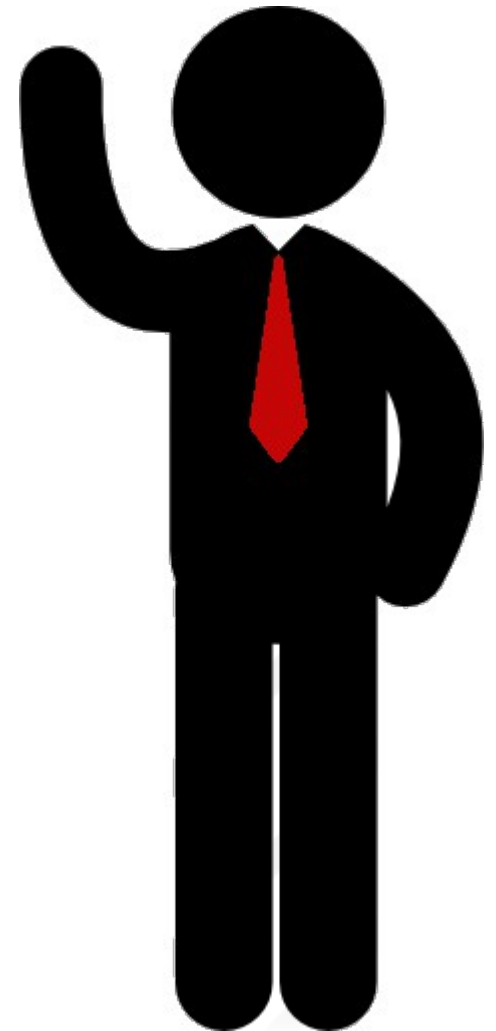
Story time – the final straw

- Find all female employees

  AND

- over 50

  AND

- with salary over 70 000

Existential crisis

- I would like to pass function as argument but I can't because Java

- All day working with collections, why is it so complex?

- Why couldn't it be like SQL?

Why Streams?

- Introducing function paradigm to Java

- Makes passing function as argument possible

- Working with collections almost like SQL

- Makes coding easier

Java 8 Streams API

- Java 8: March 2014

- Streams API

- Sequential operations on elements

- Declarative approach

- Functional paradigm (function as arg)

Stream structure

- data source (collection)

people.stream()
   .map(person -> person.getName())
   .filter(name -> name.startsWith("A"))
   .collect(Collectors.toList())

Stream structure

- stream() - transform to stream object

people.**stream**()
   .map(person -> person.getName())
   .filter(name -> name.startsWith(''A''))
   .collect(Collectors.toList())

Stream structure

- intermediate operations – zero or more

```
people.stream()
    .map(person -> person.getName())
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList())
```

Stream structure

- terminal operation – one at the end

```
people.stream()
    .map(person -> person.getName())
    .filter(name -> name.startsWith("A"))
    .collect(Collectors.toList())
```

Path to lambda

- What is lambda? (args -> logic)

```
people.stream()
   .map(person -> person.getName())
   .collect(Collectors.toList())
```

Path to lambda

- Functional interface

"A functional interface is an interface that contains only one abstract method."

Path to lambda

- Code: PathToLambdaTest

Path to lambda

- Implementing interface

- Problem: works but requires extra class

- Solution: anonymous class

Path to lambda

- Anonymous class

- Problem: works but makes code dirty

- Solution: predicate requires only one method to be implemented -> it is functional interface -> it can be written using lambda expression

Path to lambda

- Lambda can be inlined

- Usually logic using lambda won't be reused (write once and forget)

Lambda vs anonymous class

- Arguments

```
Predicate<Person>() {
    @Override
    public boolean test(Person person)
        { return person.getAge() > 70; }
};
```

```
(Person person) ->
    { return person.getAge() > 70; }
```

Lambda vs anonymous class

- Body

```
Predicate<Person>() {
    @Override
    public boolean test(Person person)
        { return person.getAge() > 70; }
};


(Person person) ->
    { return person.getAge() > 70; }
```

Lambda vs anonymous class

- Note: When dealing with lambdas we no longer speak about classes but instead we call it "a function that takes ... parameters and returns ..."

Simplifying lambda

- Skip argument type

- If one argument we can skip brackets ()

- If one instruction in body we can skip brackets {}, return and semicolon ;

- Short version makes it easier to inline lambda

Simplifying lambda

- Before:

(Person person) ->
    { return person.getAge() > 70; }

- After:

    person -> person.getAge() > 70

Method reference

- Method reference:
  people.stream()
    .map(**Person::getAge**)
    .collect(Collectors.toList())

- Simplifies lambda even further

- Only in specific cases

Method reference

Cases:

- Static methods

- Instance methods of particular objects

- Instance methods of an arbitrary object of a particular type

- Constructor

Method reference

- You don't have to remember that

- Intelij helps (Alt + Enter), both ways

```
List<Integer> ages = people.stream()
    .map(person -> person.getAge())
    .collect(Collector
```

💡	Replace lambda with method reference	▶
ⓐ↑	Go to super method	Ctrl+U
📝	Expand lambda body to {...}	▶
📝	Replace Stream API chain with loop	▶
📝	Split into declaration and assignment	▶

Properties of streams

- Lazy

- Closes after execution

- Elements are processed vertically
people.stream()
    .map(person -> person.getName())
    .filter(name -> name.startsWith(''A''))
    .collect(Collectors.toList())

Properties of streams

- No recursion allowed

- No stream splitting allowed

- Objects from outside must be final (or effective final)

```java
String prefix = "Mr. ";
prefix = "Mrs. ";
List<String> peopleWithPrefixes = people.stream()
    .map(person -> prefix + person.getName())
```

Variable used in lambda expression should be final or effectively final

Function types

- Predicate – returns true/false
  (Person person) -> person.isFemale()

- Supplier -> zero arguments,
  returns something
  () -> new ArrayList<Person>()

Function types

- Consumer - some arguments, returns nothing
  (Person person) ->
      System.out.println(person)

- BiFunction -> take two arguments, returns something
  (Person p1, Person p2) ->
      p1.getAge() > p2.getAge()

- And many more.....

Practical Training

Practical training

- Basics (solves 80% of problems)

- Advanced I

- Advanced II

- Bonus: Streams with Optional
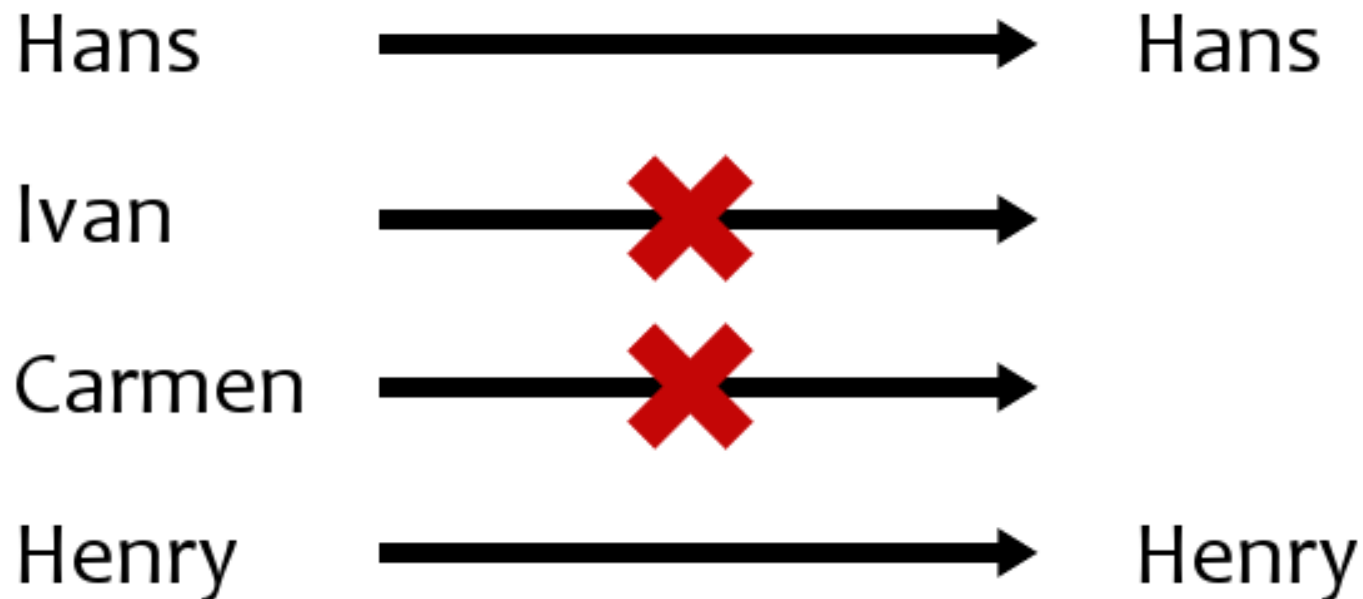
- Coffee breaks on demand

Basics

- forEach()
- collectors(toList(), toSet(), toMap())
- streams on map data structure
- filter()
- map()
- flatMap()
- debug & code style
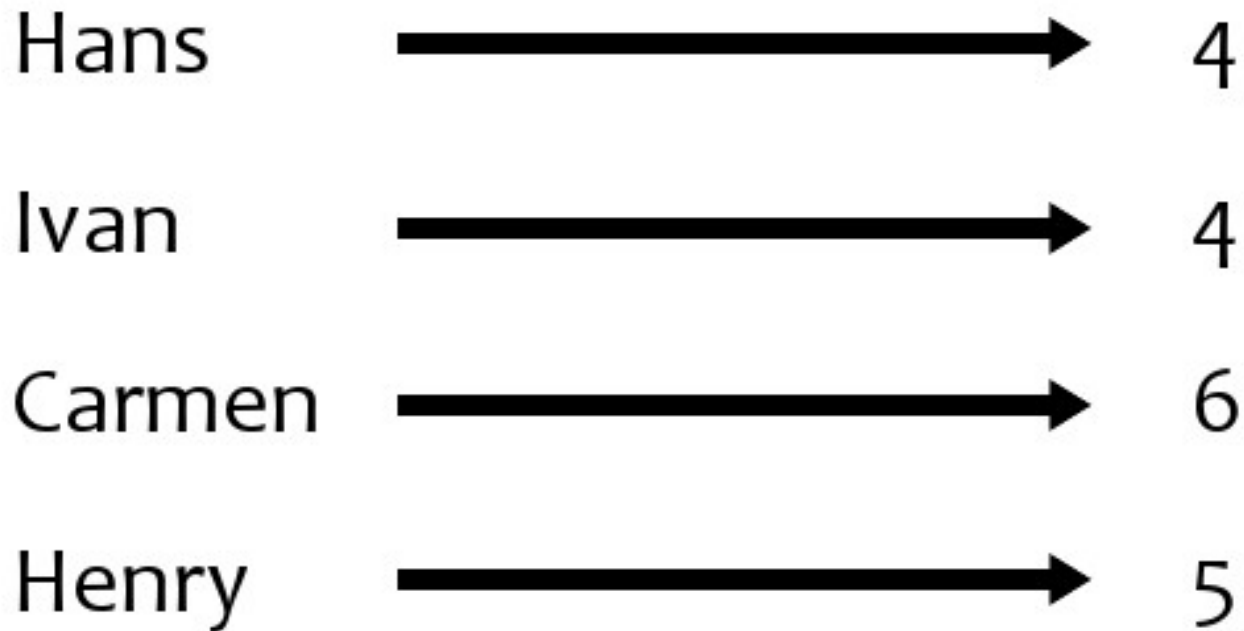- vertical processing
- lazy

filter()

- filter(name -> name.startsWith("H"))

Hans	———————>	Hans
Ivan	——✖——>	
Carmen	——✖——>	
Henry	———————>	Henry

- May change number of elements but keeps the type

map()

- map(name -> name.length())

Hans	⟶	4
Ivan	⟶	4
Carmen	⟶	6
Henry	⟶	5

- Keeps the number of elements but may change the type

flatMap()

- flatMap(nation ->
  nation.getPeople().stream())

```
[
    Germans: [ "Hans", "Heinrich" ],
    Russians: [ "Ivan", "Vladmir" ]
]
```

⬇

["Hans", "Heinrich", "Ivan", "Vladmir"]

flatMap()

- Flattens list of lists to list

- Instead of processing current element open new stream on element

Streams advantages

- Type faster, type nicer

- More expressive, higher readability

- **Required during job interviews**

- Common style for developers

Streams advantages

- Synergy with optionals

- Works similarly in other languages (arrow functions in JavaScript)

- Easy parallel processing

Streams disadvantages

- Slower than standard loops

- No strong typing

- Tricky debugging

Recruitment questions

- What will be the results of this stream?

- What is functional interface? Give me examples (Runnable, EventListener, Comparable)

- What kind of operations exist in streams? What are terminal and non-terminal operations?
- What does it mean that streams are lazy?

Sources

1. http://tutorials.jenkov.com/java-functionalprogramming/streams.html

2. https://www.samouczekprogramisty.pl/strumienie-w-jezyku-java/   (Polish)

3. https://www.baeldung.com/java-8-streams introduction

4. https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

That's All Folks!

Contact info

Patryk Drabiński

https://www.linkedin.com/in/
patryk-drabi%C5%84ski-1a6209a6/

https://github.com/t4upl

@T4Upl