The Stream API, How to Build Streams, First Patterns

José Paumard

@JosePaumard | blog.paumard.org

Agenda

What is a Stream?

Patterns to build streams

First Streams patterns

Simple reductions

What Is a Stream?

A new concept & API in Java 8

What Is a Stream?

- From a technical point of view: a typed interface

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    // interface
}
```

- Also: IntStream, LongStream & DoubleStream

What Is a Stream?

- From a developer point of view: a new concept in Java 8
- And since it is a new concept, we are free to put anything we want in it!

Stream: Definitions

- A Stream does not hold any data
  - It pulls the data it processes from a source
- A Stream does not modify the data it processes
  - Because we want to process the data in parallel with no visibility issues
- The source may be unbounded
  - Which can mean it is not finite
  - But most of the time, it only means that the size of this source is not known at build time

Stream: Definitions

- A Stream does not hold any data
  - It pulls the data it processes from a source
- A Stream does not modify the data it processes
  - Because we want to process the data in parallel with no visibility issues
- The source may be unbounded
  - Which can mean it is not finite
  - But most of the time, it only means that the size of this source is not known at build time

How to Build Streams

Many patterns!

How to Build Streams

- Of course the first pattern is:

```
// a list of Person
List<Person> people = ... ;
Stream<Person> stream = people.stream();
```

How to Build Streams

- There are many patterns for that

```
// an empty Stream
Stream.empty();
```

```
// a singleton Stream
Stream.of("one");
```

```
// a Stream with several elements
Stream.of("one", "two", "three");
```

How to Build Streams

- There are many patterns for that

```
// a constant Stream
Stream.generate(() -> "one");
```

```
// a growing Stream
Stream.iterate("+", s -> s + "+");
```

```
// a random Stream
ThreadLocalRandom.current().ints();
```

How to Build Streams

- There are many patterns for that

```
// a Stream on the letters of a String
IntStream stream = "hello".chars();
```

```
// a Stream on a regular expression
Stream<String> words =
   Pattern.compile("[^\\p{javaLetter}]")
          .splitAsStream(book);
```

```
// a Stream on the lines of a text file
Stream<String> lines = Files.lines(path);
```

How to Build Streams

- The StreamBuilder pattern

```
// first build a Stream.Builder
Stream.Builder<String> builder = Stream.builder();
```

- The add data in the builder

```
// by chaining the add() method
builder.add("one").add("two").add("three");
// or by calling accept()
builder.accept("four");
```

How to Build Streams

- Then build the stream

```
// call the build() method
Stream<String> stream = builder.build();
```

- And use the stream

```
stream.forEach(System.out::println);
```

- A built stream will throw an exception on an add() or accept() call

First Streams Patterns

Map / filter / reduce in action

First Patterns

- Map / filter / reduce on a stream of people

```
// a first way of writing it
persons.stream()
        .map(p -> p.getAge())
        .filter(age -> age > 20)
        .forEach(System.out::println);
```

- Prints out the age of the people older than 20

First Patterns

- Map / filter / reduce on a stream of people

```
// a first way of writing it
persons.stream()                                  // Stream<Person>
        .map(p -> p.getAge())
        .filter(age -> age > 20)
        .forEach(System.out::println);
```

- Prints out the age of the people older than 20

First Patterns

- Map / filter / reduce on a stream of people

```
// a first way of writing it
persons.stream()                          // Stream<Person>
       .map(p -> p.getAge())              // Stream<Integer>
       .filter(age -> age > 20)
       .forEach(System.out::println);
```

- Prints out the age of the people older than 20

First Patterns

- Map / filter / reduce on a stream of people

```
// a first way of writing it
persons.stream()                          // Stream<Person>
       .map(p -> p.getAge())              // Stream<Integer>
       .filter(age -> age > 20)           // Stream<Integer>
       .forEach(System.out::println);
```

- Prints out the age of the people older than 20
- What if we want the people themselves?

First Patterns

- Map / filter / reduce on a stream of people

```
// a second way of writing it
persons.stream()
        // .map(p -> p.getAge())
        .filter(p -> p.getAge() > 20)  // Stream<Person>
        .forEach(System.out::println);
```

- The map() call can change the type of a stream
- The filter() call does not change the type of a stream

Intermediate & Terminal Calls

- From the previous example

```
persons.stream()
        .map(p -> p.getAge())
        .forEach(System.out::println) // !!! DOES NOT COMPILE !!!
        .filter(age -> age > 20)
        .forEach(System.out::println);
```

- Suppose we want to display the elements processed by the map() call
- The forEach() does not return anything

Intermediate & Terminal Calls

- From the previous example

```
persons.stream()
        .map(p -> p.getAge())
        .peek(System.out::println)
        .filter(age -> age > 20)
        .forEach(System.out::println);
```

- The peek() call can be used for logging purposes
- Then why not use it instead of the forEach() call?

Intermediate & Terminal Calls

- Something like that:

```
persons.stream()
        .map(p -> p.getAge())
        .peek(System.out::println)
        .filter(age -> age > 20)
        .peek(System.out::println);
```

Intermediate & Terminal Calls

- The problem with this code is that…

```
persons.stream()
        .map(p -> p.getAge())
        .peek(System.out::println)
        .filter(age -> age > 20)
        .peek(System.out::println);
```

- It does not print anything!
- Why?

Intermediate & Terminal Calls

- Because
  - peek() is an intermediate operation
  - forEach() is a terminal operation

Terminal vs Intermediate Call

A terminal operation must be called to trigger the processing of a Stream

No terminal operation = no data is ever processed

How Can We Recognize a Terminal Call?

1) Read the Javadoc!

But there is also a trick…

2) A call that returns a Stream is an intermediate call
   A call that returns something else, or void is a terminal call that triggers the processing

Selecting Ranges of Data

Skip and limit

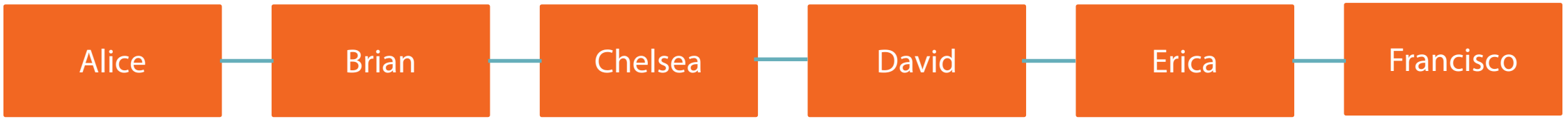
Methods skip() and limit()

- Used to select parts of a stream

```
persons.stream()
        .skip(2)
        .limit(3)
        .filter(person -> person.getAge() > 20)
        .forEach(System.out::println);
```

Methods `skip()` and `limit()`

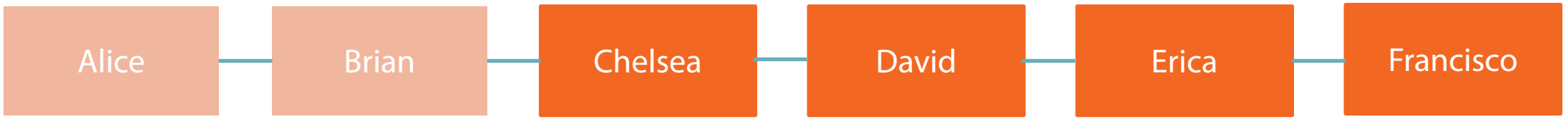
- Used to select parts of a stream




```
persons.stream()
```

Methods `skip()` and `limit()`

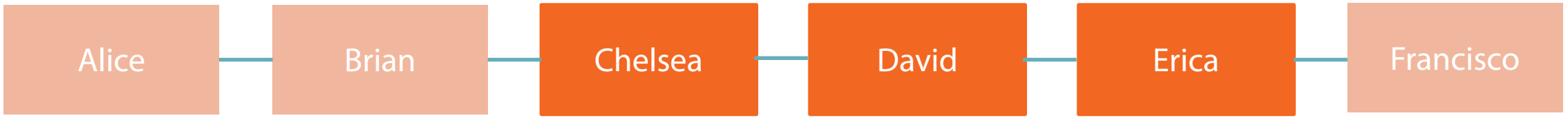
- Used to select parts of a stream




```
persons.stream()
       .skip(2)
```

Methods skip() and limit()

- Used to select parts of a stream




```
persons.stream()
       .skip(2)
       .limit(3)
```

Methods skip() and limit()

- Used to select parts of a stream


```
persons.stream()
        .skip(2)
        .limit(3)
        .filter(person -> person.getAge() > 20)
```

Methods skip() and limit()

- Used to select parts of a stream


```
persons.stream()
        .skip(2)
        .limit(3)
        .filter(person -> person.getAge() > 20)
        .forEach(System.out::println); // triggers the computation
```

Simple Reductions

Match, find, count, reduce

Match Reduction

- Three types of matchers: anyMatch(), allMatch() and noneMatch()
- They are terminal operations that return a boolean

Match Reduction

- Example of anyMatch():

```
List<Person> people = ...;

boolean b =
people.stream()
      .anyMatch(p -> p.getAge() > 20);
```

- Returns true if at least one element matches the predicate

Match Reduction

- Example of allMatch():

```
List<Person> people = ...;

boolean b =
people.stream()
      .allMatch(p -> p.getAge() > 20);
```

- Returns true if all the elements match the predicate

Match Reduction

- Example of noneMatch():

```
List<Person> people = ...;

boolean b =
people.stream()
      .noneMatch(p -> p.getAge() > 20);
```

- Returns true if no element matches the predicate

Match Reduction

- These three matchers may not evaluate the predicate for all the elements
- They are called *short-circuiting* terminal operations

Find Reduction

- There are two types of find reduction: findAll() and findAny()
- They might have nothing to return:
  - If the stream is empty
  - Or if there is no value that matches the predicate

- So they both return an Optional, that can be empty

Find Reduction

- Example of findFirst():

```
List<Person> people = ...;

Optional<Person> opt =
people.stream()
      .findFirst(p -> p.getAge() > 20);
```

- Returns the first person, if any, wrapped in an Optional
- The *first* person means the stream has an order, if not then any person is returned

Find Reduction

- Example of findAny():

```
List<Person> people = ...;

Optional<Person> opt =
people.stream()
      .findAny(p -> p.getAge() > 20);
```

- Returns any person, if it exists, wrapped in an Optional

Reduce Reduction

- There are three types of *reduce* reduction
- If no identity element is provided, then an Optional is returned
- Associativity is assumed for the reduction function, but not enforced

Reduce Reduction

- First version of reduce():

```
List<Person> people = ...;

int sumOfAges =
people.stream()
      .reduce(0, (p1, p2) -> p1.getAge() + p2.getAge());
```

- An identity element is provided, so the result is an int

Reduce Reduction

- First version of reduce():

```
List<Person> people = ...;

int maxOfAges =
people.stream()
      .reduce(0, (p1, p2) -> Integer.max(p1.getAge(), p2.getAge()));
```

- 0 is the identity element of the *max* reduction among positive integers

Reduce Reduction

- Second version of reduce():

```
List<Person> people = ...;

Optional<Integer> opt =
people.stream()
      .reduce((p1, p2) -> Integer.max(p1.getAge() + p2.getAge()));
```

- Here no identity element is provided, so the result is wrapped in an Optional

Reduce Reduction

- Third version of reduce(): used in parallel operations

```
List<Person> people = ...;

List<Integer> ages =
people.stream()
      .reduce(
         new ArrayList<Integer>(),
         (list, p) -> { list.add(p.getAge()) ; return list ;},
         (list1, list2) -> { list1.addAll(list2) ; return list1 ; }
      );
```

- Identity element, accumulator, combiner

Live Coding

Several patterns to create a Stream

Use of `limit()` and `forEach()`

Random streams

Live Coding Summary

- We saw how to build our first streams
- Simple ones, and less simple ones
- How to create random streams

Summary

- The Stream API
- Patterns to build streams on simple, regular cases
- First patterns to process data, map / filter / reduce
- Select ranges of data
- Simple reduction, agregations, optionals

Thank You

- Feel free to comment / ask questions!

@JosePaumard