(4) Java Advanced: Streams – Part 1

Nico Ludwig (@ersatzteilchen)

TOC – Part 1

TOC – Part 2

- Cited Literature:
  - Java 8 in Action, Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft

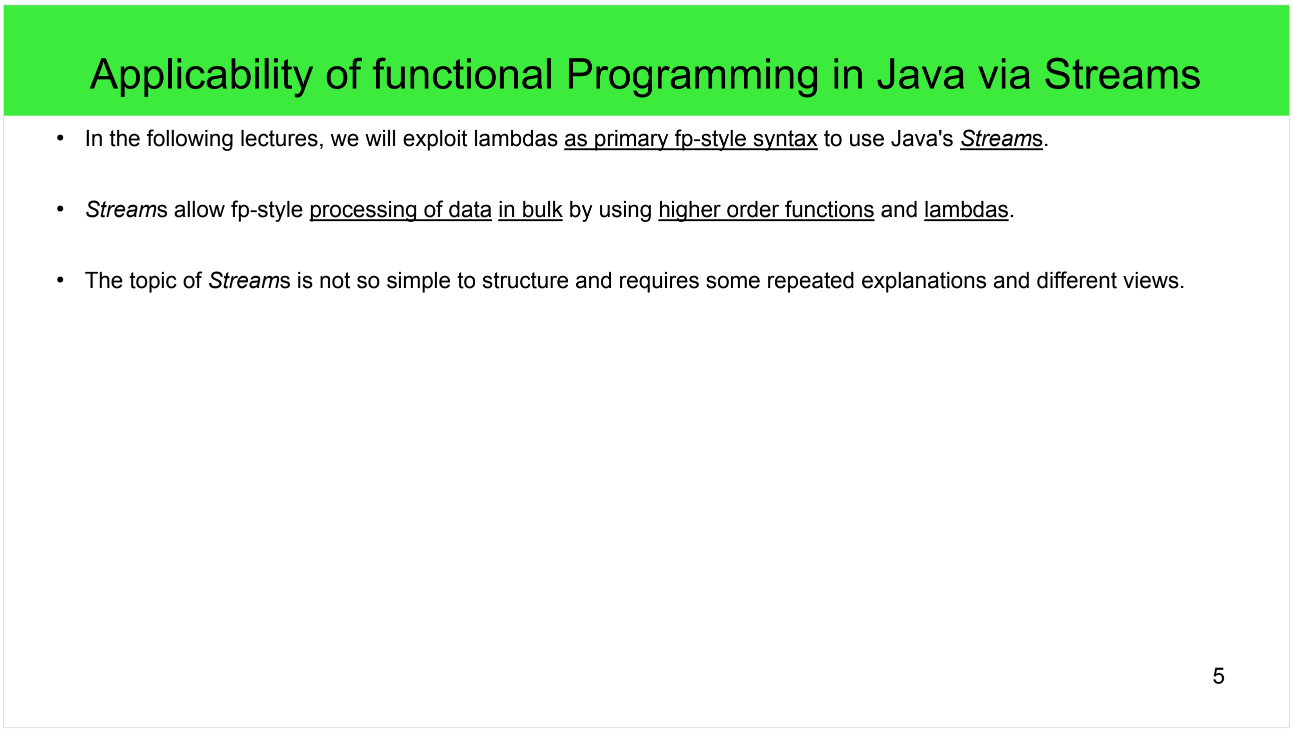
Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!



Applicability of functional Programming in Java via Streams

- In the following lectures, we will exploit lambdas as primary fp-style syntax to use Java's *Stream*s.
- *Stream*s allow fp-style processing of data in bulk by using higher order functions and lambdas.
- The topic of *Stream*s is not so simple to structure and requires some repeated explanations and different views.

- Other fp tools, which are getting added into modern languages: tuples and pattern matching.

Why Streams? – A Motivation – Part 1

- When we discussed lambdas, we showed, how algorithms using loops (external iteration) can be written as internal iteration:

```
List<String> names = List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
```

```
for (int i = 0; i < names.size(); ++i) {
        names.set(i, names.get(i).toUpperCase());
}
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```

$\longrightarrow$

```
names.replaceAll(name -> name.toUpperCase());
// names = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
```

- This works, because the method *List.replaceAll()* is a higher order function.
  - The method accepts another function, it does the iteration internally, but applies the passed function to all elements.

- But there are some problems with *List.replaceAll()*! – It is a higher order function, but does not really support fp:
  - (1) Its implementation is just using a loop, which is based on side effects:

```
// somewhere in the JDK
public interface List<E> { // simplified
        default void replaceAll(UnaryOperator<E> operator) {
              Objects.requireNonNull(operator);
              ListIterator<E> li = this.listIterator();
              while (li.hasNext()) { // Side effect: iterator status
                     li.set(operator.apply(li.next())); // Side effect: modification via iterator
              }
        }
}
```

  - (2) *List.replaceAll()* modifies the *List*, it is called on!

Why Streams? – A Motivation – Part 2

- So, the idea of methods like *List.replaceAll()* is fine, but not the way those are implemented:
  - Good: From an engineering standpoint it's useful: it separates iteration "fluff" from the relevant operation logic in the lambda.
  - Bad: Using side effects makes parallelization basically impossible and can bear concurrency problems.

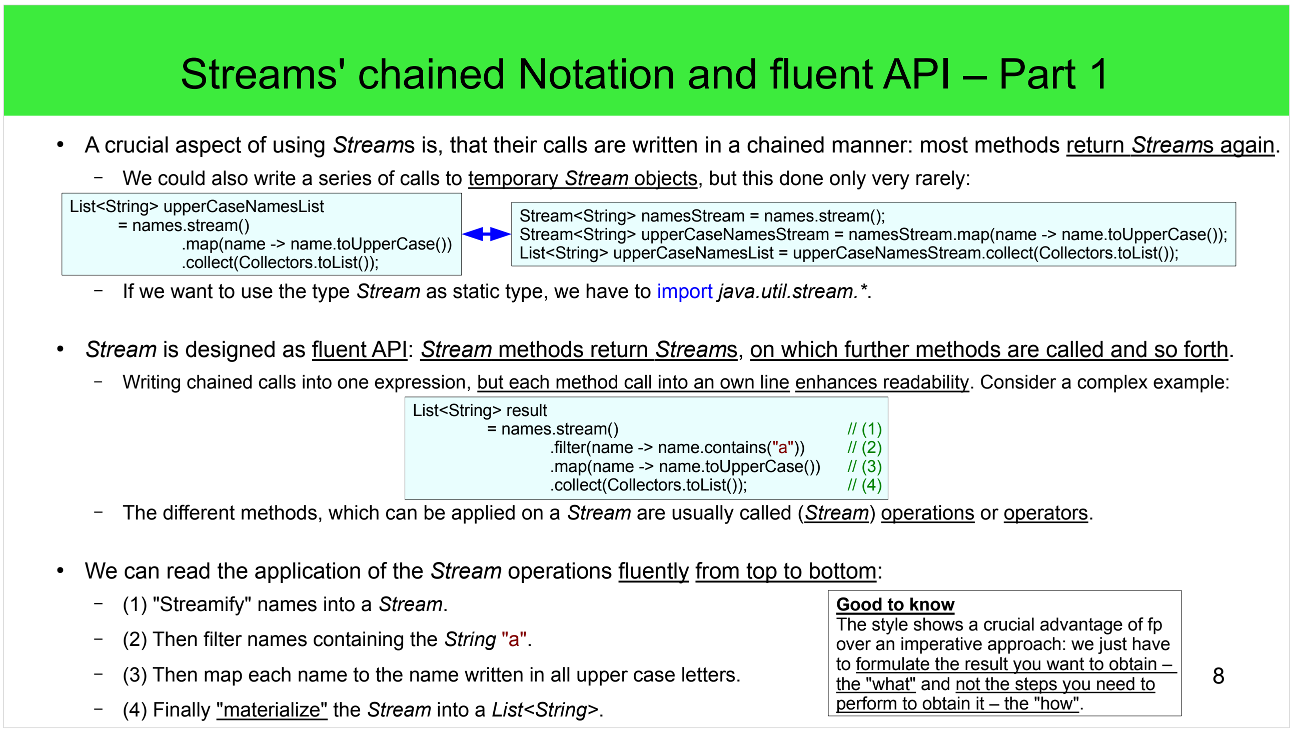
- Without further ado, this is the solution with the Java *Stream* API:

```
List<String> upperCaseNamesList = names.stream()
                                       .map(name -> name.toUpperCase())
                                       .collect(Collectors.toList());
// upperCaseNamesList = { "ASHLEY", "LISA", "SAMUEL", "PAT", "MARION" }
// names = {"Ashley", "Lisa", "Samuel", "Pat", "Marion" }
```

Good to know:
In fp-style processing the operations projection (map), selection (filter) and reduction (reduce or collect) are the most elemental ones. We can find them in most fp languages under differing names.

- The syntax of the solution does not contain any new aspect! It is "ordinary" syntax applied with some specific rules:
  - The interface *List* offers the method *List.stream()*, which returns an object of type *Stream*.
  - *Stream* offers the method *Stream.map()*, this method is a higher order function, it accepts a mapping-function.
    - This mapping-function accepts one argument of type *T* and returns another object of type *R*, i.e. it maps *T*s to *R*s.
    - In this case the mapping-function, i.e. the lambda, maps a *String* to its upper case form.
  - *Stream.map()* also returns a *Stream*. On the mapped *Stream* we call the method *Stream.collect()*.
  - *Stream.collect()* accepts a *Collector*, that collects a *Stream*'s result into a "materialized" form.
    - Here we pass a *List-Collector* (returned by the simple factory *Collectors.toList()*), that puts the mapped *Stream* into a new *List*.
  - Mind, that the resulting *List* is a new *List*! The original *List names* was not modified!

Good to know:
The Stream API also provides *Stream.reduce()*, which is similar to *Stream.collect()* but requires some discussion.



Streams' chained Notation and fluent API – Part 1

- A crucial aspect of using *Streams* is, that their calls are written in a chained manner: most methods return *Streams* again.
  - We could also write a series of calls to temporary *Stream* objects, but this done only very rarely:

```
List<String> upperCaseNamesList
        = names.stream()
                .map(name -> name.toUpperCase())
                .collect(Collectors.toList());
```

```
Stream<String> namesStream = names.stream();
Stream<String> upperCaseNamesStream = namesStream.map(name -> name.toUpperCase());
List<String> upperCaseNamesList = upperCaseNamesStream.collect(Collectors.toList());
```

  - If we want to use the type *Stream* as static type, we have to import *java.util.stream.**.

- *Stream* is designed as fluent API: *Stream* methods return *Streams*, on which further methods are called and so forth.
  - Writing chained calls into one expression, but each method call into an own line enhances readability. Consider a complex example:

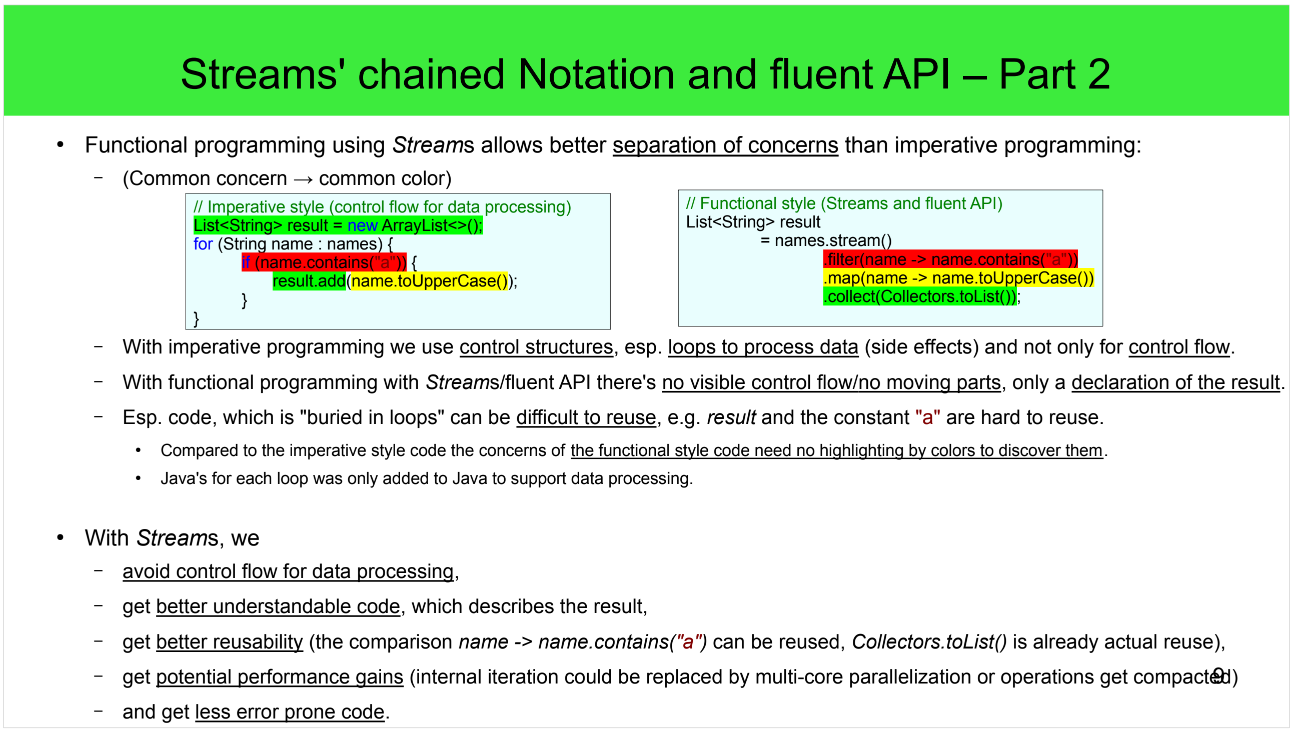
```
List<String> result
        = names.stream()                          // (1)
                .filter(name -> name.contains("a"))  // (2)
                .map(name -> name.toUpperCase())     // (3)
                .collect(Collectors.toList());       // (4)
```

  - The different methods, which can be applied on a *Stream* are usually called (*Stream*) operations or operators.

- We can read the application of the *Stream* operations fluently from top to bottom:
  - (1) "Streamify" names into a *Stream*.
  - (2) Then filter names containing the *String* "a".
  - (3) Then map each name to the name written in all upper case letters.
  - (4) Finally "materialize" the *Stream* into a *List<String>*.

Good to know
The style shows a crucial advantage of fp over an imperative approach: we just have to formulate the result you want to obtain – the "what" and not the steps you need to perform to obtain it – the "how".

- The fluent programming style is also used with JavaScript library jQuery.



Streams' chained Notation and fluent API – Part 2

- Functional programming using *Streams* allows better separation of concerns than imperative programming:
  - (Common concern → common color)

```
// Imperative style (control flow for data processing)
List<String> result = new ArrayList<>();
for (String name : names) {
        if (name.contains("a")) {
                result.add(name.toUpperCase());
        }
}
```

```
// Functional style (Streams and fluent API)
List<String> result
        = names.stream()
                .filter(name -> name.contains("a"))
                .map(name -> name.toUpperCase())
                .collect(Collectors.toList());
```

  - With imperative programming we use control structures, esp. loops to process data (side effects) and not only for control flow.
  - With functional programming with *Streams*/fluent API there's no visible control flow/no moving parts, only a declaration of the result.
  - Esp. code, which is "buried in loops" can be difficult to reuse, e.g. *result* and the constant "a" are hard to reuse.
    - Compared to the imperative style code the concerns of the functional style code need no highlighting by colors to discover them.
    - Java's for each loop was only added to Java to support data processing.

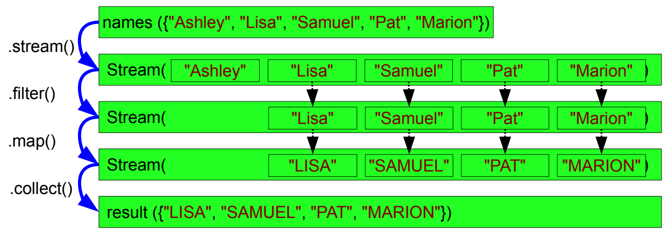
- With *Streams*, we
  - avoid control flow for data processing,
  - get better understandable code, which describes the result,
  - get better reusability (the comparison *name -> name.contains("a")* can be reused, *Collectors.toList()* is already actual reuse),
  - get potential performance gains (internal iteration could be replaced by multi-core parallelization or operations get compacted)
  - and get less error prone code.

- Chains of operations like *map()*, *filter()* and *collect()* and all the other *Stream* operations yet to be discussed, can get complex. To tackle down this complexity, some languages offer so called list comprehensions as alternative notation to bare chaining.

Stream Pipelines – Part 1

- The code using *Streams* and fluent programming reads as if it was executed from top to bottom, i.e. vertically.
  - But this is not the case! Instead it is executed horizontally. We'll now discuss how this works.

```
List<String> names = List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> result = names.stream()
                                             .filter(name -> name.contains("a"))
                                             .map(name -> name.toUpperCase())
                                             .collect(Collectors.toList());
```



.stream()
.filter()
.map()
.collect()
names ({"Ashley", "Lisa", "Samuel", "Pat", "Marion"})
Stream("Ashley" "Lisa" "Samuel" "Pat" "Marion")
Stream("Lisa" "Samuel" "Pat" "Marion")
Stream("LISA" "SAMUEL" "PAT" "MARION")
result ({"LISA", "SAMUEL", "PAT", "MARION"})

- The way processing in done in *Streams* is called pipelining:
  - The first *Stream* takes an element from *names*, which puts this element into the next *Stream*, if it contains "a".
  - The filtered *Stream* maps the element to its all upper case form.
  - The mapped *Stream* puts (collects) the element into the resulting *List*.
  - Then the next element is processed.

Stream Pipelines – Part 2

- Because the elements are processed in a pipeline, i.e. each element is processed individually, parallelization is possible:
  - Every element will only be pushed through the pipeline once.
  - No temporary result must be stored, thus no synchronization between threads is required.
- *Collection*s/arrays vs *Stream*s:
  - Both concepts abstract a sequence of elements.
  - A *Collection* abstracts management of elements in memory.
  - A *Stream* refers to a *Stream* source (maybe *Collection*) and remembers the so called pending operations, that are to be applied.
    - Pending operations are such operations like *Stream.filter()* or *Stream.map()*.
  - *Collection*s are finite, *Stream*s can be infinite.
- So, if a *Stream* refers a *Collection* and some pending operations, when are those operation actually executed?
  - *Stream*s offer two sorts of operations:
    - Intermediate operations append operations to be executed to the *Stream*.
    - Terminal operations execute all pending operations on the *Stream*.
  - Here the terminal operation *Stream.collect()* executes all pending operations.
  - Another view: We have a builder pattern starting with a *Stream* source, then intermediate operations and the terminal operation builds the result.
  - The pipeline is executed by a terminal operation.

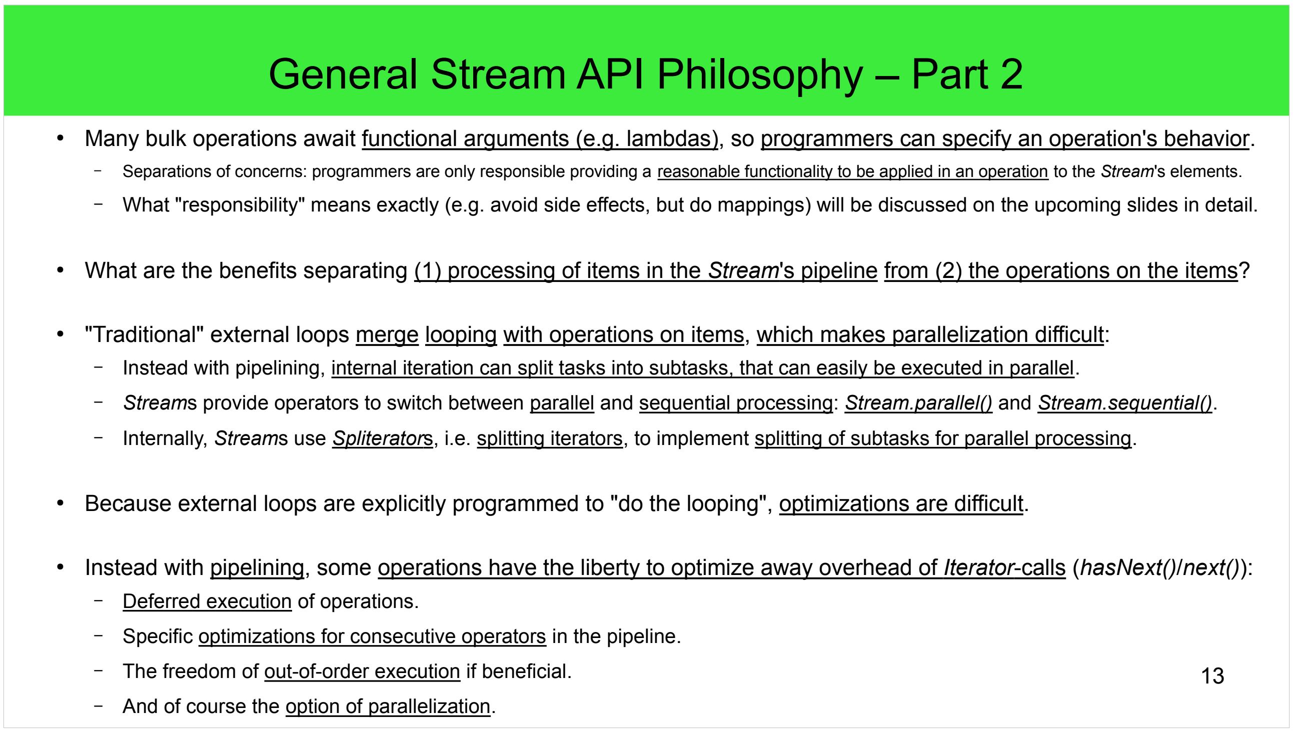
```
List<String> result
// Create a Stream from names:
= names.stream()
// Intermediate operation - apply the passed filter predicate:
.filter(name -> name.contains("a"))
// Intermediate operation - apply the passed mapping function:
.map(name -> name.toUpperCase())
// Terminal operation - execute the intermediate operations
// and collect the results into a new List:
.collect(Collectors.toList());
```

General Stream API Philosophy – Part 1

- Java 8's JDK adds the new abstraction *Stream<T>*, which is a view to a *Collection*/array or a generated sequence.
  - It provides bulk operations, so called *Stream* operations, that perform internal iteration.
  - Internal iteration allows the separation of data (*Stream*) and algorithms (functional arguments passed to *Stream* operations).
  - The *Stream* operations include *Stream.filter()*, *Stream.map()*, *Stream.collect()* and a lot more.
  - => From a practical standpoint the Stream API's idea is to enable the fundamental map/filter/collect operations in memory.
    - The general pattern of APIs similar to Java's *Stream*s is the chaining of the steps map/filter/reduce.
    - Each step accepts a lambda and data processing if controlled by the different objects returned by subsequent steps.
  - Internal iteration allows *Stream*s to support sequential and parallel execution in a convenient way.

- Instead of extending *Iterator*, the new type *Stream* was added. Why didn't they go this way?
  - One reason was, that *Iterator* should be there for external iteration only, whereas *Stream* is for internal iteration.

- *Stream* operators have no side effects on input *Stream*s, but produce new *Stream*s.
  - (*Iterator* also offers *Iterator.remove()*!)
  - Consecutive results are combined via the simple and uniformed chaining notation, because the *Stream* API is a fluent API.
  - It can be compared to the pipe functionality of different shells (e.g. bash or PSL).



General Stream API Philosophy – Part 2

- Many bulk operations await functional arguments (e.g. lambdas), so programmers can specify an operation's behavior.
  - Separations of concerns: programmers are only responsible providing a reasonable functionality to be applied in an operation to the *Stream*'s elements.
  - What "responsibility" means exactly (e.g. avoid side effects, but do mappings) will be discussed on the upcoming slides in detail.
- What are the benefits separating (1) processing of items in the *Stream*'s pipeline from (2) the operations on the items?
- "Traditional" external loops merge looping with operations on items, which makes parallelization difficult:
  - Instead with pipelining, internal iteration can split tasks into subtasks, that can easily be executed in parallel.
  - *Stream*s provide operators to switch between parallel and sequential processing: *Stream.parallel()* and *Stream.sequential()*.
  - Internally, *Stream*s use *Spliterator*s, i.e. splitting iterators, to implement splitting of subtasks for parallel processing.
- Because external loops are explicitly programmed to "do the looping", optimizations are difficult.
- Instead with pipelining, some operations have the liberty to optimize away overhead of *Iterator*-calls (*hasNext()*/*next()*):
  - Deferred execution of operations.
  - Specific optimizations for consecutive operators in the pipeline.
  - The freedom of out-of-order execution if beneficial.
  - And of course the option of parallelization.

- A *Stream* pipeline cannot be parallel and sequential. I.e. we cannot switch back and forth and execute one part of the pipeline parallel the other part of the pipeline sequential. -> The last concurrency specification before the terminal operation wins and is then valid for the full pipeline. In other words Java's *Stream*s don't support so called "segments", in opposite to reactive streams (via *observeOn()*).

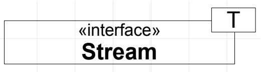
General Stream API Philosophy – Part 3

- Technically, a *Stream* houses two responsibilities:
  - it controls *Spliterator*s to partition and access the data
  - and it controls a *ReferencedPipeline* to handle the processing of data
  - Once again, we as programmers only provide the operations to be executed in the pipeline.

- The *Stream* concept allows data processing w/o intermediate collections.
  - map/filter/collect happen in memory but the *Stream* source and the destination could reside outside the process (e.g. as files).
  - *Stream*s don't hold data, instead they pull data from the *Stream* source when needed while processing.
  - *Stream*s should not (be forced to) modify data to allow parallel processing, we'll discuss an example in the upcoming slides.
  - Streams can work with unbounded sources, which we will also discuss in a minute.
  - => In contrary to collections, which hold data, can modify data and are bounded.

Stream is a Java interface

- *Stream* is a generic interface:



- Surprisingly, no public class in the JDK implements *Stream*!
  - On the other hand does *Stream* declare a lot of methods (about 40 (Java 8)).
  - However, *Stream* is not meant to be implemented by 3rd party developers (like us).

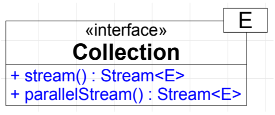
- If we need a *Stream* object, we have to get it from *Collection*s/arrays or use generator methods.
  - Intermediate *Stream* operations also create new *Stream*s (with pending operations) from a present *Stream*.
  - But the concrete implementation of those *Stream* is an implementation detail.

- Now, we'll discuss some ways to create *Stream*s from scratch.

How can we create Streams? – Part 1

- Most often, we need a *Stream* of a *Collection* (like a *List*), it's easy: the type *Collection* provides the method *Collection.stream()*.
  - So the method *Collection.stream()* is implemented by types implementing *Collection*, such as *ArrayList*:

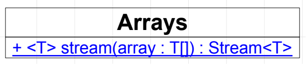
```
List<String> names = List.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
Stream<String> namesStream = names.stream();
```



  - *Collection.stream()* is a default method, each *Collection* implicitly offers this method.

- We can also create a *Stream* from an array, via the array companion class *Arrays* and the simple factory *Arrays.stream()*:

```
String[] names = {"Ashley", "Lisa", "Samuel", "Pat", "Marion"};
Stream<String> namesStream = Arrays.stream(names);
```



- When *Streams* are created from a *Collection* or array this *Collection*/array is officially called the *Stream* source.

- In a future lecture, we will introduce means to generate infinite *Streams*.
  - Then the *Stream* source is neither a *Collection* nor an array, but it is something of unknown size.

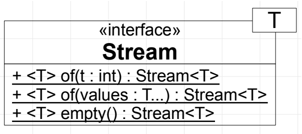
How can we create Streams? – Part 2

- Besides getting *Streams* from *Stream* sources, we can also generate new *Streams* from scratch:
  - The static method *Stream.of()* creates a *Stream* from its variable arguments list or array or of a single value:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
```

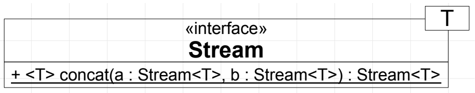
  - We can create an empty *Stream* with the static method *Stream.empty()*:

```
Stream<String> noNames = Stream.empty();
```


T
«interface»
Stream
+ <T> of(t : int) : Stream<T>
+ <T> of(values : T...) : Stream<T>
+ <T> empty() : Stream<T>

- Another interesting *Stream* operation is *Stream.concat()*, which concatenates two *Streams* into one *Stream*:

```
Stream<String> namesStream2 = Stream.of("Alex", "Peter", "Julian", "Martin", "Ruth");
Stream<String> allNames = Stream.concat(namesStream, namesStream2);
```



  - The *Stream* elements must be of the same type. If required the element types can be "adapted" via *Stream.map()*:

```
Stream<Person> personsStream = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
Stream<String> allNames = Stream.concat(namesStream, personsStream.map(person -> person.getName()));
```

- *Stream.concat()* can only be used to concatenate exactly two *Streams*!
  - The concatenation of more than two *Streams* using *Stream.concat()* must be done with cascaded *Stream.concat()* calls:

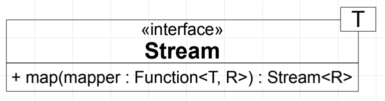
```
Stream<String> namesStream3 = Stream.of("Felix", "Olivia", "Phillip", "Viola", "David");
Stream<String> allNames = Stream.concat(namesStream, Stream.concat(namesStream2, namesStream3));
```

  - It should be said, that cascading *Stream.concat()*-calls can be very costly and can even lead to *StackOverflowExceptions*.
    - If we have more than two *Streams* to be concatenated, we're better off using *Stream.of()* and *Stream.flatMap()* like so (we'll discuss details soon):

```
Stream<String> allNames = Stream.of(namesStream, namesStream2, namesStream3).flatMap(Function.identity());
```

Projection: The Stream.map() Operation – Part 1

- The simplest operation on *Streams* is the projection. A projection maps each element in the *Stream* to another value.
- The most important projection operator is *Stream.map()*.



- *Stream.map()* creates a new *Stream*, which provides transformed, or mapped elements of the source *Stream*.
  - E.g. we can map each *String* of a *Stream<String>* to itself:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> names = namesStream.map(name -> name).collect(Collectors.toList());
// names = {"Ashley", "Lisa", "Samuel", "Pat", "Marion"}
```

  - Instead of the lambda *name -> name* as mapper-function, we can also use the simple factory *Function.identity()*:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> names = namesStream.map(Function.identity()).collect(Collectors.toList());
// names = {"Ashley", "Lisa", "Samuel", "Pat", "Marion"}
```

- A more interesting example maps the input values into different output values.
  - E.g. mapping each *String* of a *Stream<String>* to its upper-case variant:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> names = namesStream.map(name -> name.toUpperCase()).collect(Collectors.toList());
// names = {"ASHLEY", "LISA", "SAMUEL", "PAT", "MARION"}
```

  - Instead of the mapper function *name -> name.toUpperCase()*, we can also use the method reference *String::toUpperCase*:

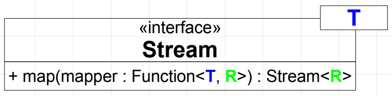
```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> names = namesStream.map(Stream::toUpperCase).collect(Collectors.toList());
// names = {"ASHLEY", "LISA", "SAMUEL", "PAT", "MARION"}
```

Projection: The Stream.map() Operation – Part 2

- Some points to consider, when using *Stream.map()*:
  - The order of the elements in the input elements needs not to correspond to the mapped elements in the output *Stream*!
  - This is true for most *Stream* operators. The order can be different due to optimization or parallelization.

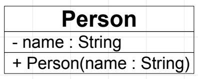
Good to know
The order of items in a *Stream* depends on the *Stream* source.

- *Stream.map()* can produce a new *Stream* of the same element type, or one of a different element type.
  - The trick: *Stream.map()* accepts a *Function<T, R>* as mapper function: the parameter type of the function can differ from the return type.



  - So, the mapper maps ***T***s to ***R***s, whereby ***T*** and ***R*** can be the same type or a different type.

- Following code maps a *Stream<String>* to a *Stream<Person>*, just by calling *Person*'s ctor:



```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<Person> persons = namesStream.map(name -> new Person(name)).collect(Collectors.toList());
// names = {Person("Ashley"), Person("Lisa"), Person("Samuel"), Person("Pat"), Person("Marion")}
```

  - Instead of the mapping-lambda, we can pass a constructor reference to *Stream.map()*:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<Person> persons = namesStream.map(Person::new).collect(Collectors.toList());
// names = {Person("Ashley"), Person("Lisa"), Person("Samuel"), Person("Pat"), Person("Marion")}
```

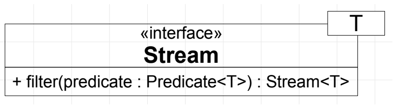
  - => We are using *Stream.map()* to convert elements as a projection.

Selection: The Stream.filter() Operation

- Another common operator is *Stream.filter()*, which is used to filter elements out of the pipeline.
  - In terms of functional programming we call such an operation selection.

- *Stream.filter()* accepts a *Predicate<T>*, which defines the test for the filtering:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> names = namesStream.filter(name -> name.contains("a")).collect(Collectors.toList());
// names = {"Lisa", "Samuel", "Pat", "Marion"}
```



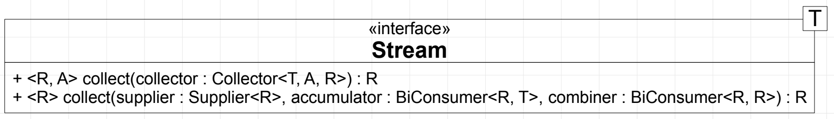
- We can easily append a projection with *Stream.map()* after the selection was applied with *Stream.filter()*:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<Person> persons = namesStream
                                .filter(name -> name.contains("a"))
                                .map(Person::new)
                                .collect(Collectors.toList());
// persons = {Person("Lisa"), Person("Samuel"), Person("Pat"), Person("Marion")}
```

  - Mind, how many loops and branches we have spared, just using this expression.

Reduction: The Stream.collect() Operation – Part 1

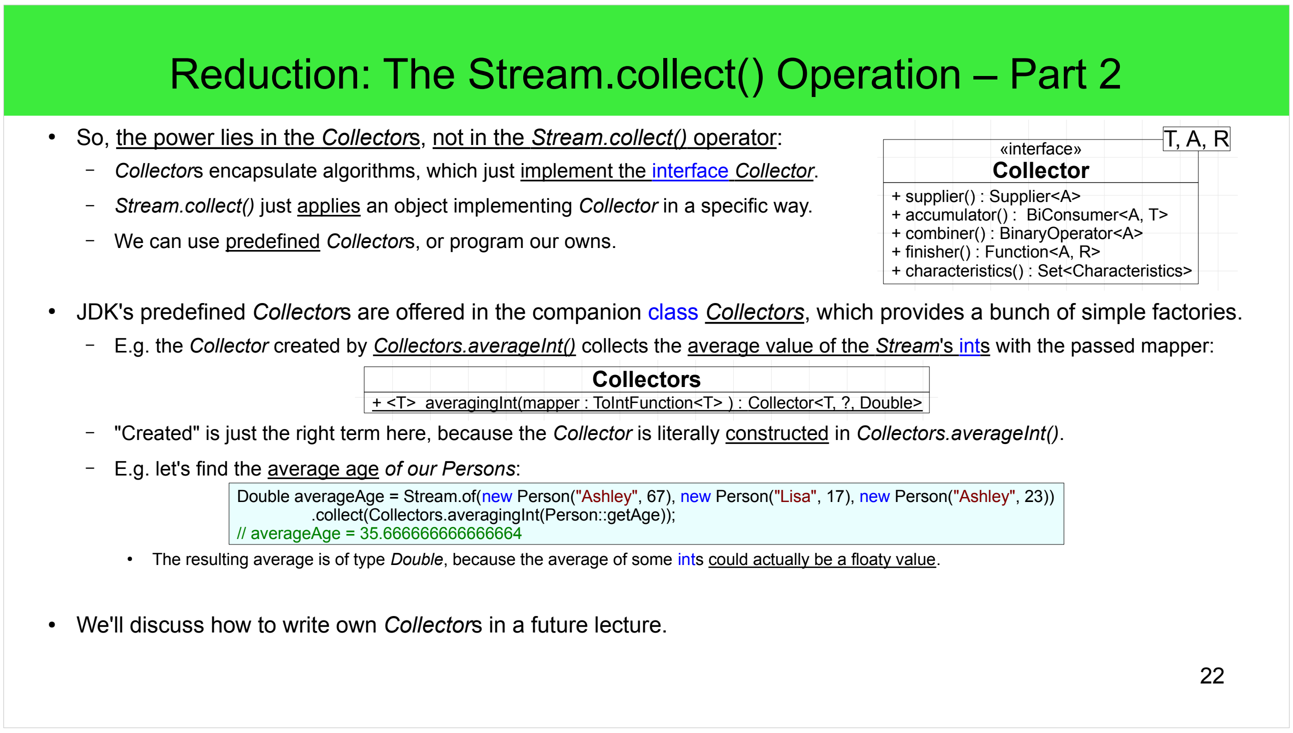
- For the time being we should shortly discuss the terminal operation *Stream.collect()*, it has following overloads:


T
«interface»
Stream
+ <R, A> collect(collector : Collector<T, A, R>) : R
+ <R> collect(supplier : Supplier<R>, accumulator : BiConsumer<R, T>, combiner : BiConsumer<R, R>) : R

  - We will only focus on the overload *Stream.collect(Collector<T, A, R>)*.
  - *Stream.collect()* is a very mighty operator and a lot of its power is in the passed *Collector* argument.
  - The overload *Stream.collect(Supplier<R>, BiConsumer<R, T>, BiConsumer<R, R>)* is yet more powerful, but also a complex beast!

- *Stream.collect()* reduces a *Stream* to another object, by collecting all elements of the *Stream*.
  - The reduction of data, also just called "reduction", is controlled by the *Collector*.
  - Because it is a terminal operation the *Stream* is consumed after *Stream.collect()* returns.

- However, besides only collecting elements into a *List*, we can do much more:
  - Generally collect to single value:
    - (1) Collect to a scalar value: counting, collecting averages, sums, extrema and concatenation of *String*s
    - (2) Collect to a *Collection*:
      - collecting elements into *List*s, *Map*s, *Set*s and other modifiable/unmodifiable or concurrent/non-concurrent *Collection*s.
      - grouping and partitioning



Reduction: The Stream.collect() Operation – Part 2

- So, the power lies in the *Collector*s, not in the *Stream.collect()* operator:
  - *Collector*s encapsulate algorithms, which just implement the interface *Collector*.
  - *Stream.collect()* just applies an object implementing *Collector* in a specific way.
  - We can use predefined *Collector*s, or program our owns.

```
                              T, A, R
«interface»
Collector
+ supplier() : Supplier<A>
+ accumulator() :  BiConsumer<A, T>
+ combiner() : BinaryOperator<A>
+ finisher() : Function<A, R>
+ characteristics() : Set<Characteristics>
```

- JDK's predefined *Collector*s are offered in the companion class *Collectors*, which provides a bunch of simple factories.
  - E.g. the *Collector* created by *Collectors.averageInt()* collects the average value of the *Stream*'s ints with the passed mapper:

```
Collectors
+ <T>  averagingInt(mapper : ToIntFunction<T> ) : Collector<T, ?, Double>
```

  - "Created" is just the right term here, because the *Collector* is literally constructed in *Collectors.averageInt()*.
  - E.g. let's find the average age *of our Persons*:

```
Double averageAge = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.averagingInt(Person::getAge));
// averageAge = 35.666666666666664
```

    - The resulting average is of type *Double*, because the average of some ints could actually be a floaty value.

- We'll discuss how to write own *Collector*s in a future lecture.

22

- All *averagingXXX Collector*s produce the arithmetic mean.

Reduction: The Stream.collect() Operation – Part 3

- For the examples in this slide deck, the *Collector* created via *Collectors.toList()* is most important to us.
  - As the name indicates, it just collects all elements of the *Stream* into a *List*.

```
Collectors
+ <T> toList() : Collector<T, ?, List<T>>
```

- So we can basically put all elements of a *Stream<String>* into a *List<String>* just like so:

```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
List<String> namesList = namesStream.collect(Collectors.toList());
// namesList = {"Ashley", "Lisa", "Samuel", "Pat", "Marion"}
```

- If there are no elements in the source *Stream*, the resulting *List* will be empty:

```
List<String> emptyList = Stream.<String>empty().collect(Collectors.toList());
// emptyList = {}
```

```
Collectors
+ toMap() : Collector
+ toSet() : Collector
+ toCollection() : Collector
...
```

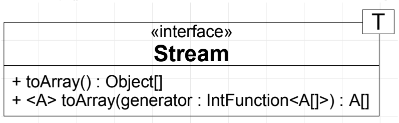
- *Collectors*' simple factories with the "*to*"-prefix can also be understood as materialization of the *Stream*.
  - After this materialization, the resulting *Collection* is independent from the input *Stream* with shallow copies of the elements.
  - Even if the input sequence is backed by a *Collection*/array, always a new *Collection*/array will be returned (hence the "*to*"-prefix).
  - Modifications (adding/removing) to the input *Collection*/array won't reflect in the result afterwards, the resulting *Collection* is a snapshot!

- Also the available predefined *Collector*s are very powerful and we have to discuss them in a future lecture.

Terminal Operations: Stream.collect() and Stream.toArray()

- *Stream.collect()* is a so called terminal operation:
  - Terminal operations collect the input of the *Stream* and return a type different from *Stream* or return nothing at all.
  - I.e. a terminal operator terminates the *Stream* pipeline. Some say, the *Stream* is consumed by a terminal operation.
  - In opposite to intermediate operators, like *Stream.map()*, which return yet other *Streams* and continue the pipeline.
  - Thus, all *Stream* operators, which return no *Stream* are terminal operations.

- Another terminal operator is *Stream.toArray()*. It consumes the *Stream* and puts its elements into an array:



Good to know
With Java 16 we got the terminal operation *Stream.toList()*. But in opposite to the *List*-collector it returns an unmodifiable *List*.

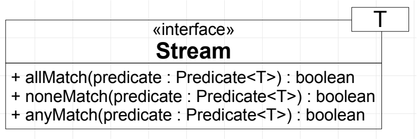
```
Stream<String> namesStream = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion");
String[] namesArray = namesStream.toArray(String[]::new);
// namesArray = {"Ashley", "Lisa", "Samuel", "Pat", "Marion"}
```

  - *Stream.toArray()*'s parameterless overload just returns an *Object[]*, so that also elements of heterogeneous type can be handled.
  - The other overload accepts an *IntFunction<A[]>*, which should provide an empty array of the passed size.
    - Mind, that the array-constructor reference *A[]::new* is compatible to *IntFunction<A[]>*. So, the matching array-constructor reference can just be used.

- *Stream.toArray()* is a bit more efficient than *Stream.collect()*, but it only produces an unmodifiable array.

Quantifier Operations – Part 1

- There are more terminal operators of interest we should discuss.
- A very interesting set of operations are quantifiers, which check the validity of a predicate for all elements at once.


T
«interface»
Stream
+ allMatch(predicate : Predicate<T>) : boolean
+ noneMatch(predicate : Predicate<T>) : boolean
+ anyMatch(predicate : Predicate<T>) : boolean

  - *Stream.allMatch()* checks if all elements match the passed predicate:

```
boolean allPersonsAreAdult
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .allMatch(person -> person.getAge() >= 18);
// allPersonsAreAdult = false
```

  - *Stream.noneMatch()* checks if no element matches the passed predicate:

```
boolean noPersonsNameStartsWithX
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .noneMatch(person -> person.getName().startsWith("X"));
// noPersonsNameStartsWithX = true
```

  - *Stream.anyMatch()* checks if any element matches the passed predicate:

```
boolean anyPersonIsAdult
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .anyMatch(person -> person.getAge() >= 18);
// anyPersonIsAdult = true
```

Quantifier Operations – Part 2

- The idea of quantifiers is directly taken from maths. It is part of the predicate logic, also called first-order logic.
  - In simple terms predicate logic means, that we define logical clauses using predicates.
  - The predicates are connected with quantifiers like "all", "some", "many", "few", "most" and "no".

- But in formalized predicate logic there are only two quantifiers:
  - "There exists" denoted with the symbol ∃. E.g. "there exists (at least) one x, for which f is valid": $\exists x(f(x))$
  - "For all" denoted with the symbol ∀. E.g. "for all x f is valid": $\forall x(f(x))$
  - We can also negate quantifiers by prefixing the respective symbol with ¬ or just striking them out.
    - "There exists no x, for which f is valid": $\neg\exists x(f(x))$or $\nexists x(f(x))$

- We can combine quantifiers with the lambda calculus and formulate predicates as lambdas:

```
boolean allPersonsAreAdult
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                   .allMatch(person -> person.getAge() >= 18);
```

$\forall person((\lambda p.p.getAge()\geq 18)(person))$

```
boolean noPersonsNameStartsWithX
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                   .noneMatch(person -> person.getName().startsWith("X"));
```

$\nexists person((\lambda p.p.getName().startsWith("X"))(person))$

```
boolean anyPersonIsAdult
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                   .anyMatch(person -> person.getAge() >= 18);
```

$\exists person((\lambda p.p.getAge()\geq 18)(person))$

Stream Termination – Part 1

- Before we go on, we should discuss, what "terminal operation" actually means.

- To make the issue clear, we will use a *Stream<Person>* as source:

```
Stream<Person> personStream = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
```

  - Then, we call the terminal operation *Stream.allMatch()* on this *Stream*:

```
boolean allPersonsAreAdult = personStream.allMatch(person -> person.getAge() >= 18);
// allPersonsAreAdult = false
```

  - One could say: "Yes, as expected!" – *allPersonsAreAdult* evaluates to false.

- However, when we call another terminal operation on the same source *Stream*, we'll get an *IllegalStateException*:

```
boolean noPersonsNameStartsWithX = personStream.noneMatch(person -> person.getName().startsWith("X"));
// Invalid! IllegalStateException: stream has already been operated upon or closed
```

- The bottom-line: when a terminal operation is called on a *Stream* it will be proverbially consumed.
  - On a consumed *Stream*, we cannot call another terminal operation, this would raise an *IllegalStateException*.
  - It means that generally, a *Stream* object can only be used once to produce a result!
  - Alas, we cannot check if a *Stream* was already consumed/terminated!

Stream Termination – Part 2

- Source *Stream*s are also getting consumed, when a terminal operation completes an intermediate *Stream*.

- To make the issue clear, we will again use a *Stream<Person>* as source:

```
Stream<Person> personStream = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
```

  - Then, we append a *Stream.filter()* to the pipeline and then call the terminal operation *Stream.allMatch()* on the intermediate *Stream*:

```
boolean personsAreAdult = personStream.filter(person -> person.getName().equals("Ashley"))
                                      .allMatch(person -> person.getAge() >= 18);
// personsAreAdult = true
```

  - Still as expected, *personsAreAdult* evaluates to true (regarding the additional intermediate *Stream.filter()* operation).

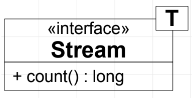
- We call another terminal operation on another intermediate *Stream*, but the same source *Stream* and get an *IllegalStateException*:

```
boolean noAdultPersonsNameStartsWithX = personStream.filter(person -> person.getAge() >= 18)
                                                    .noneMatch(person -> person.getName().startsWith("X"));
// Invalid! IllegalStateException: stream has already been operated upon or closed
```

- The bottom-line: when a terminal operation is called on an intermediate *Stream*, also its source *Stream* will be consumed.
  - And … on a consumed *Stream*, we cannot call another terminal operation, this would raise an *IllegalStateException*.

Counting Stream Elements

- We can also get the count of elements in a *Stream* with the terminal operation *Stream.count()*:



  - Although we've already used some complex operations on *Stream*s, we never need to know the actual count of elements.
  - We never need it, because, you may remember, *Stream* operations are based on internal iteration.
  - (The count of elements is mostly used in data processing via external loops, that are discouraged in fp.)

- But *Stream.count()* can be useful, because it is one way to check, if a *Stream* contains elements at all:

```
boolean noPersons = 0 == personStream.count();
```

  - It means that checking if a *Stream* is empty is necessarily a terminal operation, which consumes the *Stream*.
  - So, this line will raise an *IllegalStateException* for sure, because *personStream* was consumed by *Stream.count()*:

```
List<Person> personList = personStream.collect(Collectors.toList()); // Invalid! throws IllegalStateException
```

- In many cases collecting a *Stream* into a *Collection* and then checking for emptiness is the better way to do it:

```
List<Person> personList = personStream.collect(Collectors.toList());
boolean noPersons = personList.isEmpty(); // OK!
```

  - Mind, that *List.isEmpty()* is called on the collected *List*, which is decoupled from the source *Stream*.
  - But nevertheless *personStream* is now terminated and further operations must be done from the collected *List*.

Primitive Stream Specializations – Part 1

- Up to here, we have only used *Stream*s with reference types, i.e. UDTs, esp. types like *String*.

- The reason is simple: *Stream<T>* is a generic type and type args of generic types can only be reference types in Java.

- Of course we can use *Stream*s with types like int and double, but have to use their wrapper types effectively:

```
List<Integer> adultPersonsAges = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .map(person -> person.getAge())
                .filter(age -> age > 18)
                .collect(Collectors.toList());
// adultPersonsAges = {Integer(67), Integer(23)}
```

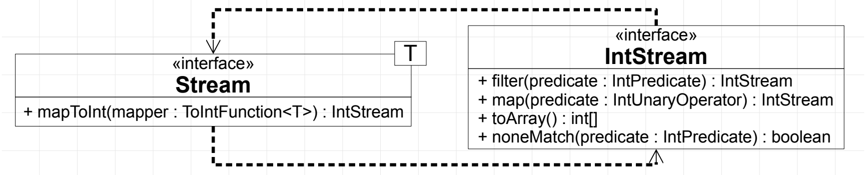
- But this solution is pretty inefficient, because each age of type int is boxed into an *Integer* object.

- There is a more efficient solution in sight: the *Stream* API supports primitive specializations of *Stream*s.
  - E.g. with the *Stream* operator *Stream.mapToInt()* we can create an *IntStream* from the mapping instead of a *Stream<Integer>*:

```
int[] adultPersonsAges = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .mapToInt(person -> person.getAge())
                .filter(age -> age > 18)
                .toArray();
// adultPersonsAges = {67, 23}
```

Primitive Stream Specializations – Part 2

- *IntStream* provides specialized operations based on int and not on a type parameter: it never needs to box ints!
  - Pretty all known *Stream* operators in *IntStream* carry int-specialized functional interface parameter types and return *IntStreams*:


«interface»
Stream
T
+ mapToInt(mapper : ToIntFunction<T>) : IntStream
«interface»
IntStream
+ filter(predicate : IntPredicate) : IntStream
+ map(predicate : IntUnaryOperator) : IntStream
+ toArray() : int[]
+ noneMatch(predicate : IntPredicate) : boolean

  - Many operators are optimized for int, e.g. *IntStream.toArray()* directly returns an int[].
    - Mind, that we cannot have a *List<int>* and if we want to avoid boxing, arrays can be a good compromise.

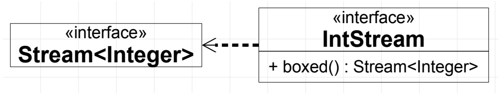
```
int[] adultPersonsAges = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
            .mapToInt(person -> person.getAge())
            .filter(age -> age > 18)
            .toArray();
// adultPersonsAges = {67, 23}
```

- Besides *IntStream*, there also exist *LongStream* and *DoubleStream* as primitive specializations.
  - E.g. *Stream* provides *Stream.mapToLong()* and *Stream.mapToDouble()* respectively.

- As can be seen in the class diagram, *Stream* and primitive specializations like *IntStream* have a mutual dependency.
  - The idea is that primitive specialized *Stream*s can be transformed back into generic *Streams*, let's see how this works...

Primitive Stream Specializations – Part 3

- We can project an *IntStream* back to a *Stream<Integer>* with the operation *IntStream.boxed()*:



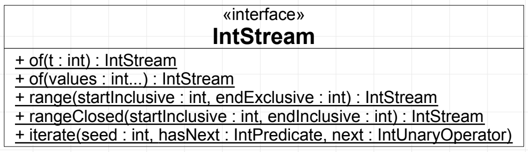
```
List<Integer> adultPersonsAges = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .mapToInt(person -> person.getAge())
                .filter(age -> age > 18)
                .boxed()
                .collect(Collectors.toList());
// adultPersonsAges = {Integer(67), Integer(23)}
```

- Boxing is a mayor bad influence on *Stream*-performance, if primitive *Stream* specializations are available, they should be used.
  - Esp. if a *Stream* represents many elements.

- In this and following lectures, it will be pointed out, if primitive *Stream*s and specialized operations are handy.

- Our discussing of primitive *Stream* specializations is not yet complete, the creation of those unleashes powerful features.

Primitive Stream Specializations – Part 4



«interface»
IntStream

+ of(t : int) : IntStream
+ of(values : int...) : IntStream
+ range(startInclusive : int, endExclusive : int) : IntStream
+ rangeClosed(startInclusive : int, endInclusive : int) : IntStream
+ iterate(seed : int, hasNext : IntPredicate, next : IntUnaryOperator)

- *IntStream.of()* and its overloads create an *IntStream* from passed int values.

```
int[] greaterThan5 = IntStream.of(5, 10, 4, 3, 8)
        .filter(number -> 0 == number > 5)
        .toArray();
// greaterThan5 = {10, 8}
```

- *IntStream.range()* and *IntStream.rangeClosed()* create … yes, sequences of ints according the specified range:

```
int[] evenNumbers = IntStream.range(0, 10)
        .filter(number -> 0 == number % 2)
        .toArray();
// evenNumbers = {0, 2, 4, 6, 8}
```

```
int[] evenNumbers = IntStream.rangeClosed(0, 10)
        .filter(number -> 0 == number % 2)
        .toArray();
// evenNumbers = {0, 2, 4, 6, 8, 10}
```

  – The operators correspond to following mathematical symbolism:

`IntStream.range(0, 10)` ➡ $[0,10[$ `IntStream.rangeClosed(0, 10)` ➡ $[0,10]$

Primitive Stream Specializations – Part 5

- *IntStream.iterate()* is a general way to create int-sequences with a start value (seed), an end condition and an increment:

```
int[] multipliesOf3 = IntStream.iterate(3, number -> number < 50, number -> number + 3)
        .toArray();
// multipliesOf3 = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48}
```

  - *IntStream.iterate()*'s params correspond to the initialization expression, conditional expression and update statement of a for loop.

- The update expression will be repeatedly applied on the current int, until the end condition evaluates to false.
  - We can use the lambda calculus to express the value of the int after, say the third iteration:

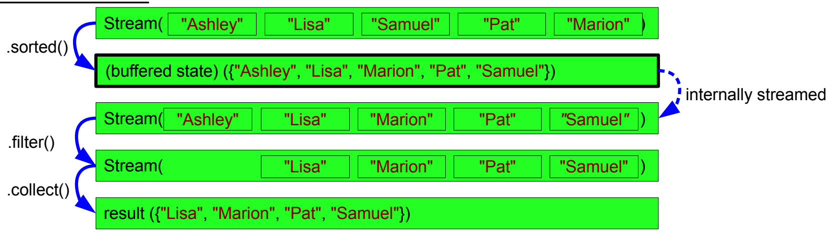
$$((\lambda number\,.\,number+3)((\lambda number\,.\,number+3)((\lambda number\,.\,number+3)3)))$$

Stateful Operations

- Some *Stream* operators require the pipeline to keep a state during processing.
- It sounds a bit strange at first, but keeping a state is required for operations like *Stream.sorted()*:

```
List<String> sortedNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
          .sorted()
          .filter(name -> name.contains("a"))
          .collect(Collectors.toList());
// sortedNames = {"Lisa", "Marion", "Pat", "Samuel"}
```

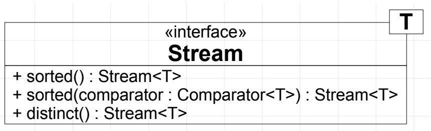
- Sorting is an operation, which must "see" all elements in the pipeline to do the sorting: the elements must be buffered.
  - And this buffer is the actual state.



Stream("Ashley" "Lisa" "Samuel" "Pat" "Marion")
.sorted()
(buffered state) ({"Ashley", "Lisa", "Marion", "Pat", "Samuel"})
internally streamed
Stream("Ashley" "Lisa" "Marion" "Pat" "Samuel")
.filter()
Stream("Lisa" "Marion" "Pat" "Samuel")
.collect()
result ({"Lisa", "Marion", "Pat", "Samuel"})

- Such operations are called stateful operations in the *Stream* API.

Other stateful Operations



- Facts about *Stream.sorted()*:
  - By default, *Stream.sorted()* sorts the elements of the *Stream* by using the elements' *Comparable* implementation.
  - There exists an overload accepting a *Comparator* to control sorting from outside. (The *Comparator* can be a lambda.)
  - *Stream.sorted()* is stable for ordered *Stream*s, for unordered *Stream*s no stability guarantees are made.
  - Stability means, that the relative order of elements is kept, if they appear to be equivalent.

- Another important stateful operation is *Stream.distinct()*, which removes duplicate elements from the source *Stream*.

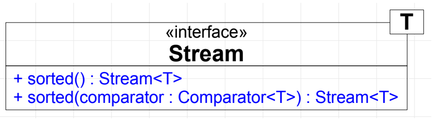
```
List<String> distinctNames = Stream.of("Ashley", "Lisa", "Ashley", "Lisa")
                .distinct()
                .collect(Collectors.toList());
// distinctNames = {"Ashley", "Lisa", "Lisa"}
```

  - It must also be stateful: an operation that removes duplicates must also "record" the history of already seen elements to do its work.
  - *Stream.distinct()* bases on the equality of elements, i.e. it calls *equals()* on the elements.
  - *Stream.distinct()* is stable for ordered *Stream*s.

Stream Sorting in Focus – Part 1

- Sorting a *Stream* is very simple using *Stream.sorted()*:

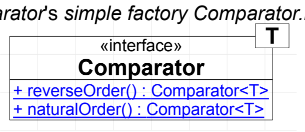
```
List<String> result = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
                            .sorted()
                            .collect(Collectors.toList());
// result = List{"Ashley", "Lisa", "Marion", "Pat", "Samuel"}
```



  - *Stream.sorted()* calls *Comparable.compareTo()* of each object to do the sorting.
  - => The types of objects to be sorted with *Stream.sorted()* must implement *Comparable*!

- We can customize sorting by passing a specific *Comparator* to *Stream.sorted()*'s overload.
  - E.g. to sort the names in reversed order, we can use the *Comparator* from *Comparator*'s *simple factory Comparator.reverseOrder()*:

```
List<String> result = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
                            .sorted(Comparator.reverseOrder())
                            .collect(Collectors.toList());
// result = List{"Samuel", "Pat", "Marion", "Lisa", "Ashley"}
```


«interface»
Comparator
T
+ reverseOrder() : Comparator<T>
+ naturalOrder() : Comparator<T>

  - *Comparator.naturalOrder()* creates a *Comparator* representing the way in which *Comparable* objects would normally get compared:
    - The natural order *Comparator* just calls the implementation of *Comparable.compareTo()* of the objects being processed (*String*s in our case).

```
List<String> result = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
                            .sorted()
                            .collect(Collectors.toList());
// result = List{"Ashley", "Lisa", "Marion", "Pat", "Samuel"}
```

⇔

```
List<String> result = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
                            .sorted(Comparator.naturalOrder())
                            .collect(Collectors.toList());
// result = List{"Ashley", "Lisa", "Marion", "Pat", "Samuel"}
```

    - => *Comparator.naturalOrder()* creates "the" *Comparator* just comparing using the implementation of *Comparable* of the respective UDT:

```
Comparator<String> stringComparator = Comparator.<String>naturalOrder();
```

    - *Comparator.naturalOrder()* can be used as basis for more complex *Comparator*s.

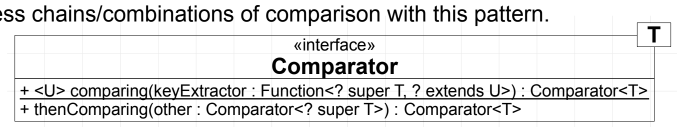
Stream Sorting in Focus – Part 2

- However, sometimes it is need to do more elaborate sorting, e.g. sort primary for the name and secondary for the age.
  - Following *Stream.sorted()*'s design we must create a *Comparator* with a comparison to put this multi-level-sorting into effect.
  - Per se such a *Comparator* is not difficult to implement, but is kind of dull and one could add bugs.
  - To safe us from dullness and failure, *Comparator* provides some default methods to create and combine other *Comparators* easily:

```
List<Person> result = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                                       .sorted(Comparator
                                                      .comparing(Person::getFirstName)
                                                      .thenComparing(Person::getAge))
                                       .collect(Collectors.toList());
// result = List{Person{"Ashley", 23}, Person{"Ashley", 67} , Person{"Lisa", 17}}
```

  - The simple factory *Comparator.comparing()* creates a *Comparator* from the accessor-method we pass to it.
    - The official *Comparator*-API calls the parameters accepting the accessor-methods "key-extractors".
    - => *Comparator.comparing(Person::getFirstName)* creates a *Comparator<Person>* which compares *Person*s using the result of *Person.getFirstName()*.
  - *Comparator.thenComparing()* creates a new *Comparator* from this then comparing using the *Comparator* created from its accessor method.
    - => *Comparator.thenComparing(Person::getAge)* creates a *Comparator<Person>* which compares *Person*s using the result of *Person.getAge()*, but before comparing *Person* with *Person.getAge()* it compares Persons with the result of *Person.getFirstName()*.

- We could create endless chains/combinations of comparison with this pattern.


T
«interface»
Comparator
+ <U> comparing(keyExtractor : Function<? super T, ? extends U>) : Comparator<T>
+ thenComparing(other : Comparator<? super T>) : Comparator<T>

  - The API pattern features a fluent API (*Comparator*-methods return *Comparator*s) the oo design pattern applies decorator.

Stream Sorting in Focus – Part 3

- We can use *Comparator*'s simple factories to create *Comparator*s from types which do not implement *Comparable*.
  - Remember when we sorted *Car*s for driven distances, we had to implement a dedicated *Comparator*. – This is no longer needed:

```
// <CarDrivenDistanceComparator.java>
public class CarDrivenDistanceComparator implements Comparator<Car> {
        @Override
        public int compare(Car lhs, Car rhs) {
                return Double.compare(lhs.getDrivenDistance(), rhs.getDrivenDistance());
        }
}
```

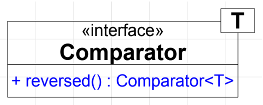
```
List<Person> result = cars.stream()
                .sorted(new CarDrivenDistanceComparator())
                .collect(Collectors.toList());
```

Wow!

```
List<Person> result = cars.stream()
                .sorted(Comparator.comparing(Car::getDrivenDistance))
                .collect(Collectors.toList());
```

- We can also simply descend any part of the sorting by reversing the responsible part of the comparison.
  - E.g. order the *Person*s primary for the name and secondary for the age, but descending. That it super-easy now:

```
List<Person> result = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                                        .sorted(Comparator
                                                        .comparing(Person::getFirstName)
                                                        .thenComparing(Comparator.comparing(Person::getAge).reversed()))
                                        .collect(Collectors.toList());
// result = List{Person{"Ashley", 67}, Person{"Ashley", 23} , Person{"Lisa", 17}}
```

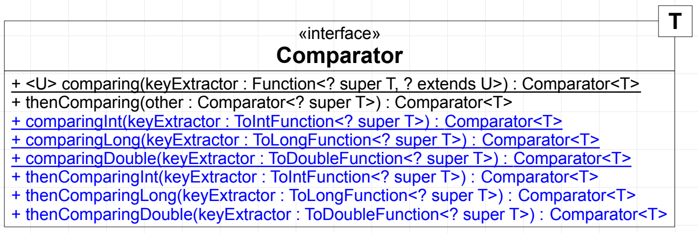


  - => In the *thenComparing()*-call we just have to pass a *Comparator* comparing *Person*s by age and reverse it with *reverse()*.

Stream Sorting in Focus – Part 4

- Creating *Comparator*s for primitive types using *Comparator.comparing()* produces *Comparator*s boxing primitive objects.
  - Boxing takes place internally, because accessor-methods are of type *Function<T, U>*, where *U* is the boxed return type.

- *Comparator* provides primitive specializations of its simple factories creating *Comparator*s for primitive accessor-methods.
  - With those, accessor-methods are of type *ToXXXFunction<T>*, where *XXX* is the primitive return type.
  - The internal comparison is the done with specialized methods (e.g. *Integer.compare()*), which yields even more performance:



```
T
«interface»
Comparator
+ <U> comparing(keyExtractor : Function<? super T, ? extends U>) : Comparator<T>
+ thenComparing(other : Comparator<? super T>) : Comparator<T>
+ comparingInt(keyExtractor : ToIntFunction<? super T>) : Comparator<T>
+ comparingLong(keyExtractor : ToLongFunction<? super T>) : Comparator<T>
+ comparingDouble(keyExtractor : ToDoubleFunction<? super T>) : Comparator<T>
+ thenComparingInt(keyExtractor : ToIntFunction<? super T>) : Comparator<T>
+ thenComparingLong(keyExtractor : ToLongFunction<? super T>) : Comparator<T>
+ thenComparingDouble(keyExtractor : ToDoubleFunction<? super T>) : Comparator<T>
```

- Let's use *Comparator.thenComparingInt()* to create a more efficient partial comparison of ages:

```
List<Person> result = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                             .sorted(Comparator
                                         .comparing(Person::getFirstName)
                                         .thenComparingInt(Person::getAge))
                             .collect(Collectors.toList());
// result = List{Person{"Ashley", 23}, Person{"Ashley", 67} , Person{"Lisa", 17}}
```

Streams and Side Effects

- As we asserted in the beginning of the discussion of fp, side effects are not desired in fp algorithms.
  - As all lambdas in Java can have side effects, this is also true for lambdas used as argument for *Stream* operators.
- Technically, Java forbids one kind of side effect from lambdas: we cannot assign captured locals from within a lambda:

```
String lastNameInFilter;
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
          .filter(name -> {
                lastNameInFilter = name; // Invalid! local variables referenced from a lambda expression must be final or effectively final
                return name.contains("s");
          })
          .collect(Collectors.toList());
```

- On the other hand, we cannot force immutability in Java nicely. The explicit or implicit "finality" is pretty weak.
  - Immutability is inappropriate in oo languages, oo is about mutable, but encapsulated state of objects.
  - Sometimes such a mismatch of paradigms is called impedance mismatch.
  - To cross the mismatch, the *Stream* API makes compromises to allow side effects in some operations.
- However, which kinds of side effects can we have at all in a lambda of a *Stream* operation? We concentrate on those:
  - (1) Changing the operational state of the program, e.g. modifying (not assigning) a captured variable or writing to the console.
  - (2) Mutating the state of a *Stream*'s element, e.g. calling a setter.
  - (3) Modifying the *Stream* source.

Side Effects: Changing the Operational state of a Program – Part 1

- Consider this code, it prints a message to the console each time the lambda passed to *Stream.filter()* is executed:

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .filter(name -> {
                    System.out.printf("just filtering %s%n", name);
                    return name.contains("s");
                })
                .collect(Collectors.toList());
```

```
Terminal
just filtering Ashley
just filtering Lisa
just filtering Samuel
just filtering Pat
```

  - Sure, it works ...

- We can print messages to the console each time the lambda for *Stream.map()* and *Stream.filter()* is called in the same pipeline:

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .map(name -> {
                    System.out.printf("just mapping %s%n", name);
                    return name;
                })
                .filter(name -> {
                    System.out.printf("just filtering %s%n", name);
                    return name.contains("s");
                })
                .collect(Collectors.toList());
```

```
Terminal
just mapping Ashley
just filtering Ashley
just mapping Lisa
just filtering Lisa
just mapping Samuel
just filtering Samuel
just mapping Pat
just filtering Pat
```

  - This one also works, the output nicely shows how the pipeline works.
  - The remarkable point is, that the order of *System.out.printf()* calls in the code doesn't match the order in the output.
    - => The iteration is not visible, it is an internal iteration.

Side Effects: Changing the Operational state of a Program – Part 2

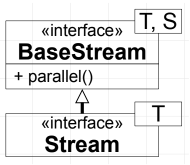
- Mind, that lambdas and *Streams* are taken from fp, in which side effects are a kind of disturbance.
  - As soon as we introduce side effects into our code, the order of side effects may differ from their written order in the code.
  - The order of execution of expressions is generally irrelevant in fp, remember that this is how terms work in maths.

- Here the order of side effects for console output is harmless, but a good indicator to what will happen with other side effects.
  - Regard, that writing a file or database could be the side effect we talk about, and then the order of activity can be crucial!

- But, this is not all to this story, in the last pipeline we could predict, that *Stream.map()* and *Stream.filter()* alternate…
  - A real eye-opener, that side effects can be a disaster when using *Streams*, is when we switch to parallel processing:

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
                .parallel()
                .map(name -> {
                    System.out.printf("just mapping %s%n", name);
                    return name;
                })
                .filter(name -> {
                    System.out.printf("just filtering %s%n", name);
                    return name.contains("s");
                })
                .collect(Collectors.toList());
```

Terminal
```
just mapping Samuel
just mapping Pat
just filtering Pat
just mapping Marion
just filtering Marion
just mapping Lisa
just mapping Ashley
just filtering Ashley
just filtering Lisa
just filtering Samuel
```



  - The operator *Stream.parallel()* enables parallel processing.
  - If parallel processing is active, the pipeline executes subsequent operators and elements in an unpredictable execution order.

Side Effects: stateful Lambdas

- We should shed a light on how situations occur, which are more problematic with side effects.

- The *Stream* API calls lambdas, which involve side effects stateful lambdas.
  - But most *Stream* lambdas must be stateless.
  - Mind, this is different from the idea of stateful *Stream* **operations**.
    - In those the operation itself has a state independent of a passed lambda (*Stream.sorted()* and *Stream.distinct()*).

- For the *Stream* API statelessness means, that the execution of lambdas on elements must not depend on each other.
  - More clearly: we are not allowed to modify or accumulate data!
  - Only, if such a dependency is not given, the order of execution of work in the pipeline doesn't matter.

When are Side Effects getting a Problem? – Part 1

- Following code mimics *Stream.distinct()* on an *IntStream* and sums the distinct numbers then repeatedly:

```
while (true) {
        Set<Integer> known = new HashSet<>();
        int sum = IntStream.of(11, 12, 11, 12, 13, 14, 14, 15)
                    .map(number -> // simulation of Stream.distinct():
                          known.add(number) // checks and keeps a state
                                ? number
                                : 0)
                    .sum();
        System.out.println(sum);
        Thread.sleep(300);
}
```

```
Terminal
65
65
65
65
65
65
```

  - It works and it looks harmless, but checking the content of the *Set* and collecting elements into the *Set* as side effect is dangerous.

- When we change this algorithm to work with *Stream.parallel()*, we can see some problems:

```
while (true) {
        Set<Integer> known = new HashSet<>();
        int sum = IntStream.of(11, 12, 11, 12, 13, 14, 14, 15)
                    .parallel()
                    .map(number -> // simulation of Stream.distinct():
                          known.add(number) // checks and keeps a state
                                ? number
                                : 0)
                    .sum();
        System.out.println(sum);
        Thread.sleep(300);
}
```

```
Terminal
65
77
65
79
65
77
```

  - As can be seen, when we repeat this way of summing in the loop, we'll get different results for the same input!

When are Side Effects getting a Problem? – Part 2

```
while (true) {
        Set<Integer> known = new HashSet<>();
        int sum = IntStream.of(11, 12, 11, 12, 13, 14, 14, 15)
                    .parallel()
                    .map(number -> // simulation of Stream.distinct():
                          known.add(number) // checks and keeps a state
                                ? number
                                : 0)
                    .sum();
        System.out.println(sum);
        Thread.sleep(300);
}
```

```
Terminal
65
77
65
79
65
77
```

- OK? What's the problem? What's going on?

- The problem is, that <u>*HashSet* is no thread safe collection</u>!
  - *HashSet.add()* is a method, which sets a value <u>and</u> checks the existence of an element.
  - It looks like the method *HashSet.add()* does the <u>setting and checking in an atomic way</u>, and this also correct, <u>but only in one thread</u>.
  - But in this case we execute multiple calls of *HashSet.add()* <u>from multiple threads</u> implicitly, <u>because the *Stream* is parallel</u>.
  - <u>Each thread can interrupt the work of each other thread at any time.</u>
  - Thread A may <u>only execute the checking-part of *HashSet.add()*</u>, <u>while it is interrupted by thread B, which already adds the value</u>.
  - The problem: <u>depending on the order of access to the *HashSet* and the state of the *HashSet* elements might be added or skipped</u>.
  - The effect: <u>the result (sum) can vary between some mostly incorrect values in a **non-deterministically** way</u>.

When are Side Effects getting a Problem? – Part 3

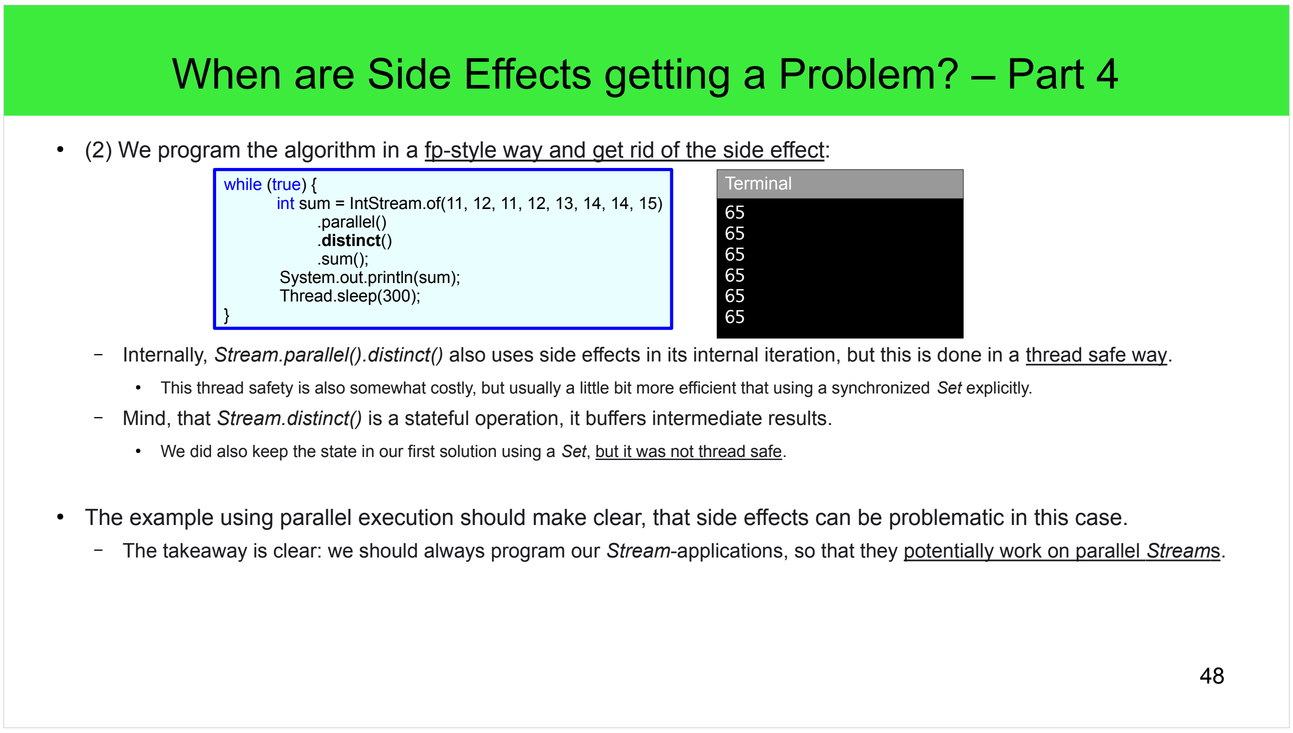
- How can we fix our algorithm to work deterministically in a parallel execution scenario?

- (1) We can use a *Set* implementation, that is thread safe, e.g. a synchronized *Set*:

```
while (true) {
        Set<Integer> known = Collections.synchronizedSet(new HashSet<>());
        int sum = IntStream.of(11, 12, 11, 12, 13, 14, 14, 15)
                    .parallel()
                    .map(number ->
                          known.add(number)
                                ? number
                                : 0)
                    .sum();
        System.out.println(sum);
        Thread.sleep(300);
}
```

```
Terminal
65
65
65
65
65
65
```

  – The idea to make concurrent access to *Set.add()* atomic for multiple threads. (In Java) we say we synchronize access in *Set.add()*.
  – The simple factory *Collections.synchronizedSet()* decorates a present *Set* with a synchronized wrapper.
  – Internally, a synchronized *Set* decorates some critical methods with a lock, so that only one thread can enter the method at once.
    - Notice, this means, that in a synchronized *Set* methods like *Set.add()* are really atomic then!
    - Java allows to synchronize whole methods or only so called critical sections with user defined locks, e.g. with the synchronized keyword.
  – Of course, the downside is, that the code is trickier.
    - Another downside is that using synchronized access is pretty costly, but this is required operationally.



When are Side Effects getting a Problem? – Part 4

- (2) We program the algorithm in a fp-style way and get rid of the side effect:

```
while (true) {
        int sum = IntStream.of(11, 12, 11, 12, 13, 14, 14, 15)
                .parallel()
                .distinct()
                .sum();
        System.out.println(sum);
        Thread.sleep(300);
}
```

```
Terminal
65
65
65
65
65
65
```

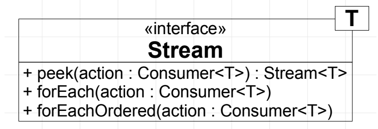
  - Internally, *Stream.parallel().distinct()* also uses side effects in its internal iteration, but this is done in a thread safe way.
    - This thread safety is also somewhat costly, but usually a little bit more efficient that using a synchronized *Set* explicitly.
  - Mind, that *Stream.distinct()* is a stateful operation, it buffers intermediate results.
    - We did also keep the state in our first solution using a *Set*, but it was not thread safe.

- The example using parallel execution should make clear, that side effects can be problematic in this case.
  - The takeaway is clear: we should always program our *Stream*-applications, so that they potentially work on parallel *Streams*.

- The decision when to use parallel execution to get better performance utilizing multiple processing units is not easy. Its effect depends on the count of input data to process and on the complexity/time consumption of the operation to be executed on multiple threads. For only a few ints as we did here, execution could even be less performant than sequential execution, because the splitting/joining/synchronization costs overweight the parallelization gain.
  - Parallel Streams do not make sense to handle blocking operations in general. This is because the common fork join pool (CFJP) in its default configuration is pretty limited and its threads are shared among other operations, e.g. CompletableFutures. – In the worst case all threads of the CFJP could block, which leads to a halt of the JVM.

Stream Operations, which allow Side Effects – Part 1

- So, side effects in *Stream* operators are discouraged, but as already mentioned, the *Stream* API opens the gate a little:
  - (1) There is a simple rule: only perform one operation per lambda for an operator.
  - (2) Three operators, *Stream.peek()*, *Stream.forEach()* and *Stream.forEachOrdered()* are allowed to accept stateful lambdas:


T
«interface»
Stream
+ peek(action : Consumer<T>) : Stream<T>
+ forEach(action : Consumer<T>)
+ forEachOrdered(action : Consumer<T>)

  - (1) and (2) allow to put work on the pipeline in a structured way and to tell potentially stateful activities from others.

- Some initial words on those operations:
  - All three operators accept a *Consumer*, as you remember, *Consumer*s are meant to hold side effects.
  - *Stream.peek()* is an intermediate operation.
  - *Stream.forEach()* and *Stream.forEachOrdered()* are terminal operations.

- Mind, that the *Stream* API itself provides stateful operations like *Stream.distinct()*, but limits the freedom to define own.
  - The *Stream* API cannot hinder programmers from having side effects in operator's lambdas, instead it defines rules.
  - … and those rules esp. mean only using the above mentioned three operations to perform side effects.

Stream Operations, which allow Side Effects – Part 2

- *Stream.peek()* is used to literally peek into the *Stream*. The passed lambda will be called for each element in the *Stream*.
  - Its parameter is of type *Consumer<T>*: the "current" element of the pipeline is passed to the *Consumer<T>* respectively.
  - *Stream.peek()* is excellent to print something to the console, e.g. to monitor the processing of the pipeline for debugging purposes:

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .filter(name -> {
                    System.out.printf("just filtering %s%n", name);
                    return name.contains("s");
                })
                .collect(Collectors.toList());
```

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .peek(name -> System.out.printf("just peeking %s%n", name))
                .filter(name -> name.contains("s"))
                .collect(Collectors.toList());
```

- E.g. with *Stream.peek()* we have a "legal" way to monitor the *Stream*'s elements before and after *Stream.filter()*:

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .peek(name -> System.out.printf("before filter %s%n", name))
                .filter(name -> name.contains("s"))
                .peek(name -> System.out.printf("after filter %s%n", name))
                .collect(Collectors.toList());
```

Terminal
```
before filter Ashley
after filter Ashley
before filter Lisa
after filter Lisa
before filter Samuel
before filter Pat
```

- Sure, we can also change "operation states" of a program with *Stream.peek()*, but we should think twice before doing so.
  - E.g. we should not modify captured variables or collect data (with *Stream*s we do this with … *Streams*, not with side effects).
  - However, writing logs for debugging purposes, e.g. to the console is appropriate for *Stream.peek()*.

Stream Operations, which allow Side Effects – Part 3 – Warning about Stream.peek()

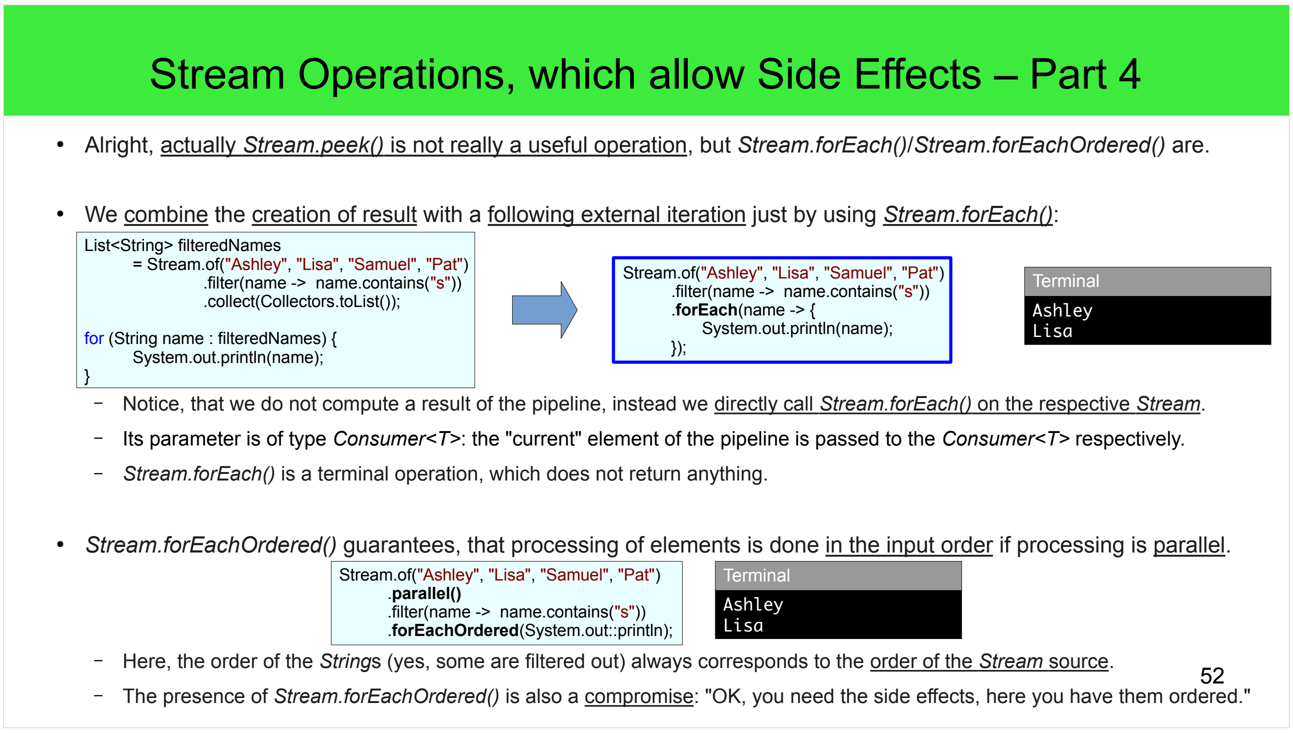
- **Warning: The side effects specified in the lambda of *Stream.peek()* can be optimized away by the JRE!**
  - So, if we have code, which relies on the actual execution of side effects in *Stream.peek()*, we might be in a bad luck:

```
List<Integer> values = Arrays.asList(1, 2, 3);
List<Integer> squaredValues = new ArrayList<>(values.size());
long count = values.stream()
        .map(number -> number * number)
        .peek(squaredValues::add)
        .count();
// count = 3              // Sure!
// squaredValues = {} // Oops!
```

- The Java standard (>= 9) states, that:
  - An implementation can elide operations (or stages) from a pipeline if it can prove that it won't affect the **result** of the computation.
  - It can therefore remove the invocation of behavioral parameters (lambdas).
  - => Effectively, this means that side effects of lambdas may not always be executed!

- This thesis makes clear, that *Stream*s are for fp! Side effects don't play a role and can be optimized away!

- *Stream.peek()* must be used carefully or should be never used at all.
  - Combinations of relying on certain operation orders together with terminal operators and parallel processing can lead to a mess.
  - **=> Don't use *Stream.peek()* in productive code, only for debugging.**



Stream Operations, which allow Side Effects – Part 4

- Alright, actually *Stream.peek()* is not really a useful operation, but *Stream.forEach()*/*Stream.forEachOrdered()* are.
- We combine the creation of result with a following external iteration just by using *Stream.forEach()*:

```
List<String> filteredNames
        = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                    .filter(name ->  name.contains("s"))
                    .collect(Collectors.toList());

for (String name : filteredNames) {
        System.out.println(name);
}
```

```
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
        .filter(name ->  name.contains("s"))
        .forEach(name -> {
             System.out.println(name);
        });
```

```
Terminal
Ashley
Lisa
```

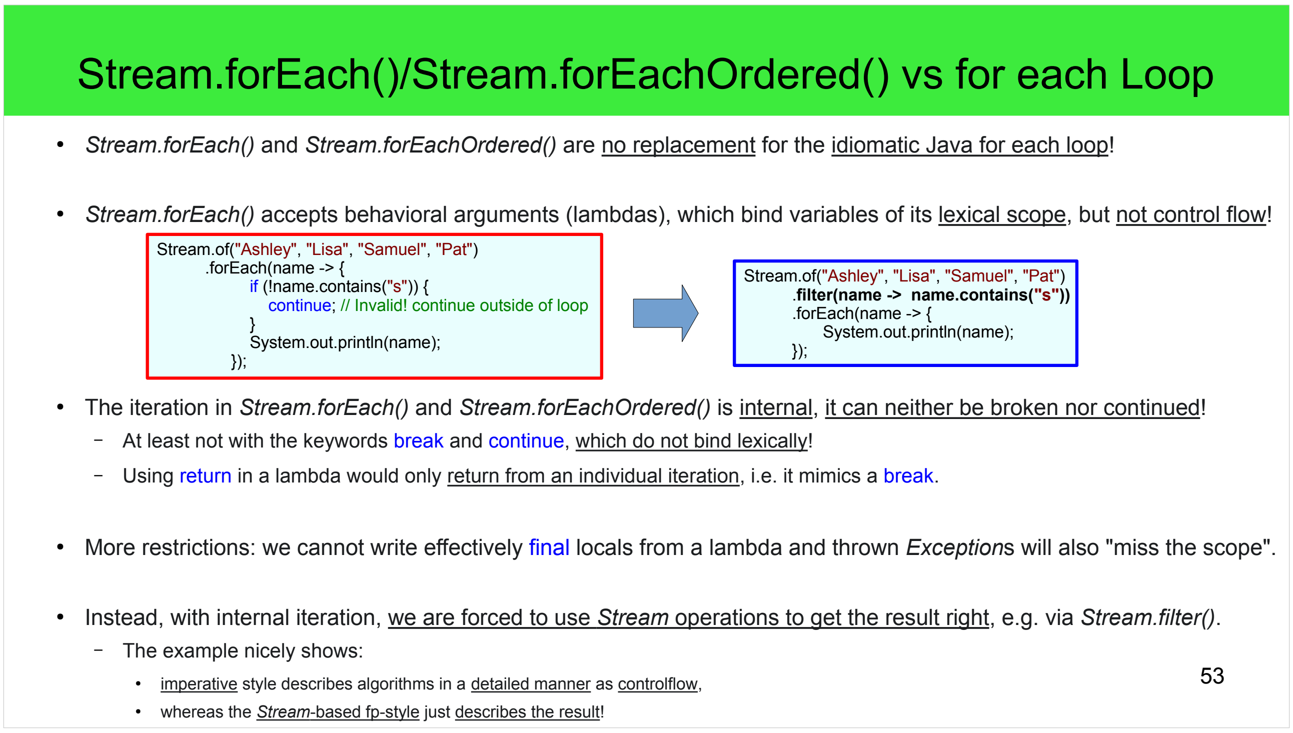
  - Notice, that we do not compute a result of the pipeline, instead we directly call *Stream.forEach()* on the respective *Stream*.
  - Its parameter is of type *Consumer<T>*: the "current" element of the pipeline is passed to the *Consumer<T>* respectively.
  - *Stream.forEach()* is a terminal operation, which does not return anything.
- *Stream.forEachOrdered()* guarantees, that processing of elements is done in the input order if processing is parallel.

```
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
        .parallel()
        .filter(name ->  name.contains("s"))
        .forEachOrdered(System.out::println);
```

```
Terminal
Ashley
Lisa
```

  - Here, the order of the *String*s (yes, some are filtered out) always corresponds to the order of the *Stream* source.
  - The presence of *Stream.forEachOrdered()* is also a compromise: "OK, you need the side effects, here you have them ordered."

- *Stream.forEachOrdered()* can only re-establish ordering, if the *Stream* source is ordered. E.g. a *List* is ordered but a *Set* in not necessarily ordered. *Stream.of()* always returns an ordered *Stream*.
- The relation between *Stream.forEachOrdered()* and *Stream.forEach()* corresponds to the relation between *Stream.findAny()* and *Stream.findFrist()* in parallel execution. One could say that *Stream.findFrist()* is always ordered.



Stream.forEach()/Stream.forEachOrdered() vs for each Loop

- *Stream.forEach()* and *Stream.forEachOrdered()* are no replacement for the idiomatic Java for each loop!
- *Stream.forEach()* accepts behavioral arguments (lambdas), which bind variables of its lexical scope, but not control flow!

```
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
        .forEach(name -> {
                if (!name.contains("s")) {
                    continue; // Invalid! continue outside of loop
                }
                System.out.println(name);
            });
```

```
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
        .filter(name -> name.contains("s"))
        .forEach(name -> {
                System.out.println(name);
        });
```

- The iteration in *Stream.forEach()* and *Stream.forEachOrdered()* is internal, it can neither be broken nor continued!
  - At least not with the keywords break and continue, which do not bind lexically!
  - Using return in a lambda would only return from an individual iteration, i.e. it mimics a break.
- More restrictions: we cannot write effectively final locals from a lambda and thrown *Exception*s will also "miss the scope".
- Instead, with internal iteration, we are forced to use *Stream* operations to get the result right, e.g. via *Stream.filter()*.
  - The example nicely shows:
    - imperative style describes algorithms in a detailed manner as controlflow,
    - whereas the *Stream*-based fp-style just describes the result!

53

- Remember, that there was on of the few alternative design proposals for lambdas in Java, namely BGGA. BGGA's (Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé) idea: introduce a function type incl. lexical binding of this, return, continue and break.

Allowed Side Effects Sailing close to the Wind – Part 1

- Now its time to see how *Stream.forEach()*/*Stream.forEachOrdered()* can be used with parallel processing.

- E.g. let's use a *StringBuilder* to join all *String*s of a *Stream<String>* together using *Stream.forEach()* and in parallel:

```
StringBuilder result = new StringBuilder();
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
		.parallel()
		.forEach(result::append);
System.out.println(result);
```

```
Terminal
SamuelPatNULNULNULAshleyLisa
```

- But … it doesn't work! At least, the result is not as expected …
  - (1) The order of the *String*s in the result is somehow random.
  - (2) There are NUL ('\0') characters somewhere in the middle of the result *String*!
  - (3) The effects (1) and (2) look differently each time this code is executed.

- So, what went wrong? Several things:
  - (1) When executed in parallel, the pipeline delivers results in an unpredictable order.
    - => The order of sub-*String*s in result doesn't correspondent to the input *String*s in the *Stream* source.
  - (2) *StringBuilder* isn't thread safe! *Stream.parallel()* leads to a situation, in which *result* is accessed concurrently, damaging its state.
    - => NUL characters indicate, that the status of *StringBuilder*'s internal buffer is screwed up.

Allowed Side Effects Sailing close to the Wind – Part 2

- We should solve the problem with output order first, *Stream.forEachOrdered()* just fixes this issue:

```
StringBuilder result = new StringBuilder();
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
            .parallel()
            .forEachOrdered(result::append);
System.out.println(result);
```

```
Terminal
AshleyLisaNULNULNULSamuelPat
```

  - *Stream.forEachOrdered()* processes the elements in the "encountered" order of predecessor source *Stream*.
  - "Encountered" means, that a predecessor source *Stream* might not have an order, but in our case we have an encountered order.
  - But, we still have problems with the thread unsafe buffer in *StringBuilder* (see the NULs).

- We can solve this problem by using the thread safe alternative to *StringBuilder*: the *StringBuffer*:

```
StringBuffer result = new StringBuffer();
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
            .parallel()
            .forEachOrdered(result::append);
System.out.println(result);
```

```
Terminal
AshleyLisaSamuelPat
```

- So after a relatively long way crossing some traps we have a functional solution:
  - (1) But the code is relatively "special" for a pretty easy problem: we have to use the *StringBuffer* and *Stream.forEachOrdered()*.
  - (2) We only wrote this code, because we made it work for parallel processing (virtually, the processing strategy shouldn't matter).
  - (3) We only found the problems of parallel processing, because we looked at them very closely while testing.

Allowed Side Effects Sailing close to the Wind – Part 3

- Java allows stateful lambdas as behavioral argument for *Stream.peek()*, *Stream.forEach()* and *Stream.forEachOrdered()*.
  - It just means, that Java allows side effects in those lambdas.
  - But if the pipeline is processed in parallel, we must care for thread safety ourselves (e.g. use *StringBuffer* instead of *StringBuilder*).

- Esp. the concatenation of *String*s is a so common use case, that it shouldn't be complicated.
  - The *Stream* API supports this operation via a dedicated *Collector* provided via an overload of *Collectors.joining()*:

Collectors
+ joining() : Collector<CharSequence, ?, String>
+ joining(delimiter : CharSequence) : Collector<CharSequence, ?, String>
+ joining(delimiter : CharSequence, prefix : CharSequence, suffix : CharSequence) : Collector<CharSequence, ?, String>

  - It means we can use *Stream.collect()* to do the *String* concatenation without any stateful and complicated code:

```
StringBuffer result = new StringBuffer();
Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                .parallel()
                .forEachOrdered(result::append);
System.out.println(result);
```

```
String result // Excellent!
          = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
                        .parallel()
                        .collect(Collectors.joining());
System.out.println(result);
```

Terminal
```
AshleyLisaSamuelPat
```

- We can draw important conclusions from the example of *String* concatenation we have discussed up to here:
  - (1) The combination of side effects an parallel processing is tricky and can be dangerous (also independently of *Streams*).
  - (2) In most cases there are solutions without statefulness and side effects supporting fp principles in Java (with or without *Streams*).
  - (3) Knowledge of the possibilities with Java and *Streams* is the key of finding ways to "do it right".

Side Effects and the Rule of Non-Interference

- Besides stateful lambdas there exists another kind of side effect in the *Stream*s API, the so called interference.

- Interference means, that the code in a lambda modifies the *Stream* source.
  - Mind, that a *Stream* is no *Collection*, instead it can rather refer a backing *Collection* or array as *Stream* source.

- For interference with *Stream*s the rule is simple: **all lambdas passed to *Stream* operations must be non-interfering**!
  - **In other words: lambdas passed to *Stream* operations must not modify the *Stream* source!**

- If the non-interference rule is hurt, processing often ends with (fail fast) *Exceptions*, or strange results might occur:

```java
List<String> names = new ArrayList<>(List.of("Ashley", "Lisa", "Samuel", "Pat"));
names.stream()
        .map(String::toUpperCase)
        .forEach(names::add); // Invalid! Throws ConcurrentModificationException
```

```java
List<String> immutableStreamSource = List.of("Ashley", "Lisa", "Samuel", "Pat"); // Produces an immutable List
immutableStreamSource.stream()
          .map(String::toUpperCase)
          .forEach(immutableStreamSource::add); // Of course invalid! Throws UnsupportedOperationException
```

  - It should be said, that both modifications on the source *Collection*s would also not work in (for each) loops in Java!
  - However, there can be situations, in which this violation is not easy to find, because the source and the side effect are on very different locations in code and timely locations because of deferred execution.

Modifying Stream Elements

- Is it allowed to perform side effects on the elements of a *Stream*? Consider this:

```
Person
+ Person(name : String, age : int)
+ getAge() : int
+ setAge(age : int)
+ getName() : String
+ setName(name : String)
```

```
List<Person> persons = List.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
List<Person> persons2 = persons.stream()
        .map(person -> {
            person.setAge(person.getAge() + 1); // modifies the age of a person in the Stream
            return person;
        })
        .collect(Collectors.toList());
// persons = {new Person("Ashley", 68), new Person("Lisa", 18), new Person("Ashley", 24)}
```

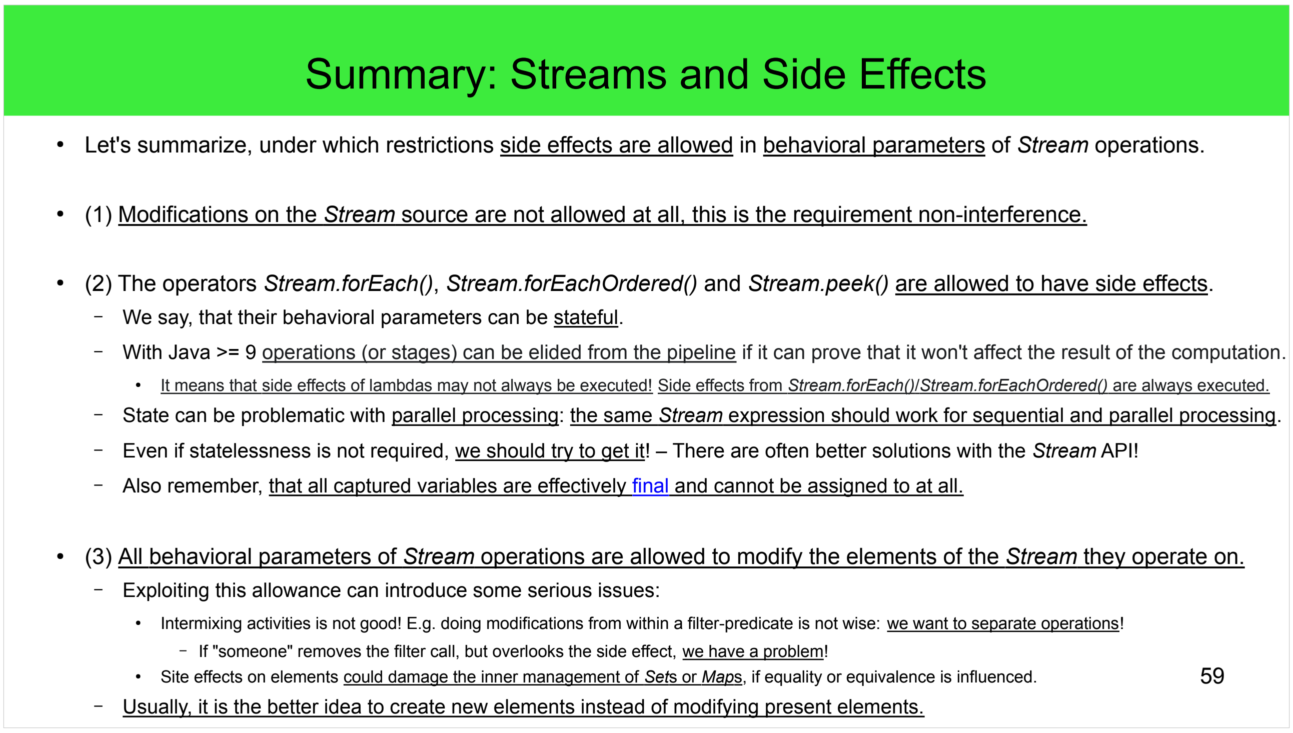
  - The new aspect in the code is, that the lambda passed to *Stream.map()* modifies all *Person*s it's operating on (age is incremented).

- This is ok! In any *Stream* operator accepting a lambda, this lambda is allowed to perform side effects on the *Stream* elements.

- But fp means, that we should avoid any side effects, also such on the elements (which *Stream*s allow per se).
  - Instead we can create new *Person* objects from the old ones with an incremented age:

```
List<Person> olderPersons = persons
            .stream()
            .map(person -> new Person(person.getName(), person.getAge() + 1))
            .collect(Collectors.toList());
// persons = {new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23)}
// olderPersons = {new Person("Ashley", 68), new Person("Lisa", 18), new Person("Ashley", 24)}
```

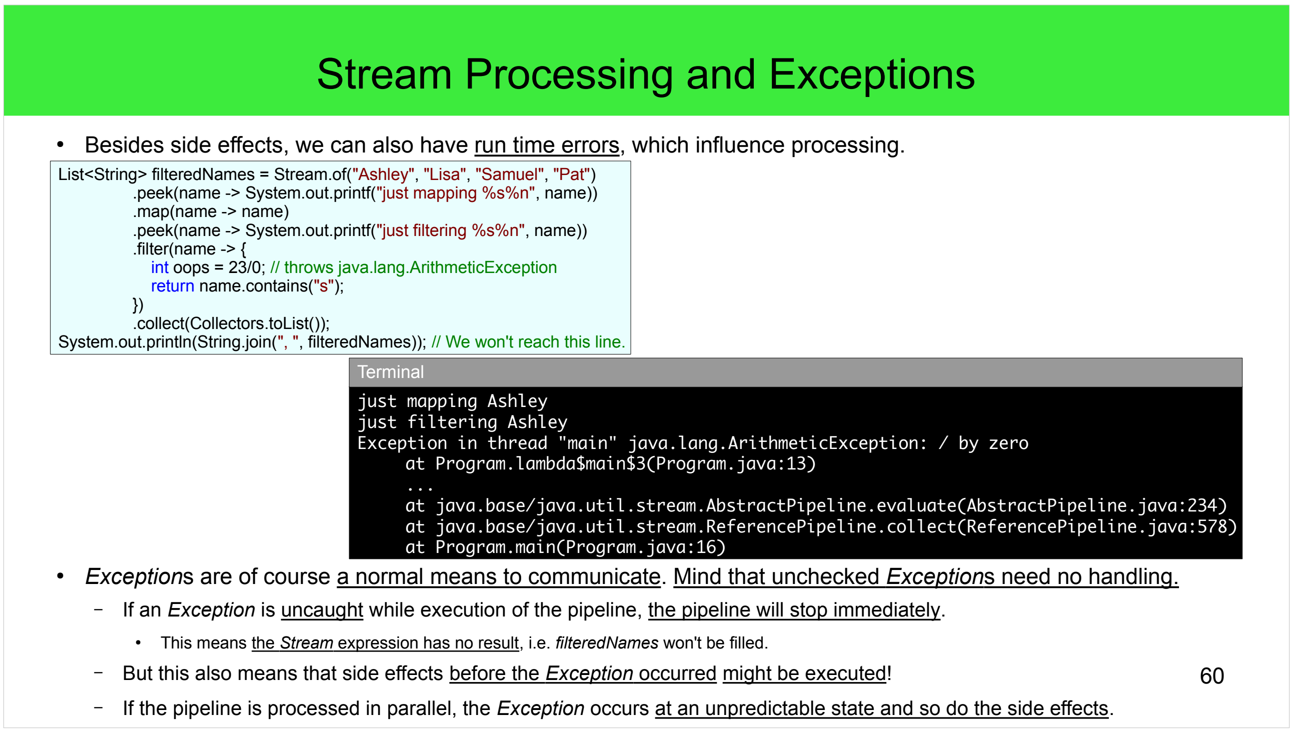
  - It perfectly embraces the idea of side effect free programming and immutable objects!



Summary: Streams and Side Effects

- Let's summarize, under which restrictions side effects are allowed in behavioral parameters of *Stream* operations.
- (1) Modifications on the *Stream* source are not allowed at all, this is the requirement non-interference.
- (2) The operators *Stream.forEach()*, *Stream.forEachOrdered()* and *Stream.peek()* are allowed to have side effects.
  - We say, that their behavioral parameters can be stateful.
  - With Java >= 9 operations (or stages) can be elided from the pipeline if it can prove that it won't affect the result of the computation.
    - It means that side effects of lambdas may not always be executed! Side effects from *Stream.forEach()*/*Stream.forEachOrdered()* are always executed.
  - State can be problematic with parallel processing: the same *Stream* expression should work for sequential and parallel processing.
  - Even if statelessness is not required, we should try to get it! – There are often better solutions with the *Stream* API!
  - Also remember, that all captured variables are effectively final and cannot be assigned to at all.
- (3) All behavioral parameters of *Stream* operations are allowed to modify the elements of the *Stream* they operate on.
  - Exploiting this allowance can introduce some serious issues:
    - Intermixing activities is not good! E.g. doing modifications from within a filter-predicate is not wise: we want to separate operations!
      - If "someone" removes the filter call, but overlooks the side effect, we have a problem!
    - Site effects on elements could damage the inner management of *Sets* or *Maps*, if equality or equivalence is influenced.
  - Usually, it is the better idea to create new elements instead of modifying present elements.

- Internally, *Stream.distinct()* uses a *HashSet* to hold the buffer/state. – The *Stream* operations provided by the JDK are allowed to do this, but this is an implementation detail. But we as programmers are not allowed to do this e.g. in a behavioral argument (lambda). Programmers are in charge to obey the rules, esp. to avoid side effects.



Stream Processing and Exceptions

- Besides side effects, we can also have run time errors, which influence processing.

```
List<String> filteredNames = Stream.of("Ashley", "Lisa", "Samuel", "Pat")
            .peek(name -> System.out.printf("just mapping %s%n", name))
            .map(name -> name)
            .peek(name -> System.out.printf("just filtering %s%n", name))
            .filter(name -> {
                int oops = 23/0; // throws java.lang.ArithmeticException
                return name.contains("s");
            })
            .collect(Collectors.toList());
System.out.println(String.join(", ", filteredNames)); // We won't reach this line.
```

Terminal

```
just mapping Ashley
just filtering Ashley
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at Program.lambda$main$3(Program.java:13)
     ...
     at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
     at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:578)
     at Program.main(Program.java:16)
```

- *Exception*s are of course a normal means to communicate. Mind that unchecked *Exception*s need no handling.
  - If an *Exception* is uncaught while execution of the pipeline, the pipeline will stop immediately.
    - This means the *Stream* expression has no result, i.e. *filteredNames* won't be filled.
  - But this also means that side effects before the *Exception* occurred might be executed!
  - If the pipeline is processed in parallel, the *Exception* occurs at an unpredictable state and so do the side effects.

- Because *Stream* processing ends immediately after an *Exception* is thrown, it might be a better option to handle errors as data in *Stream* pipelines. This is the way it is done in Spring WebFlux and it can be mimicked easily in Java as well, but it must be done explicitly.

Avoid Side Effects and "States"

- The parallelization of operations on items in the pipeline only leads to valid results, if they have no side effects.
  - I.e. no side effects on the *Stream* source and none on the elements itself.

- The reason: If we avoid side effects, the order of execution of split subtask doesn't matter.

- *Exception*s seem not to fit into the word of functional programming, at least how Java implements *Exception*s.
  - Checked *Exception*s are per se problematic with lambdas.
  - A program can be in an "exceptional state", and in fp we don't like states. – Mathematical formulas don't have state as well.
  - The stacktrace is also not a fp concept (just mind the memory consumption with recursive call stacks).

Deferred/Lazy Execution – Part 1

- Now we have to discuss an important phenomenon of *Streams*: deferred or lazy execution. Consider this code:

```
Stream<Person> adultPersons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Mary", 23))
            .filter(person -> {
                boolean isAdult = person.getAge() >= 18;
                if (isAdult) {
                    System.out.printf("%s is adult %n", person.getName());
                }
                return isAdult;
            });
```

  - But, we get no output:

```
Terminal
```

- Is there a bug in our code? No! – The code in the lambda was just not yet executed.
  - We can force the execution of the pipeline with a terminal *Stream* operation:

```
List<Person> adultPersonsList = adultPersons.collect(Collectors.toList());
```

  - All right:

```
Terminal
Ashley is adult
Mary is adult
```

Deferred/Lazy Execution – Part 2

- All right, this is the effect of lazy execution: as long as the pipeline is not terminated, it is in a "standby" mode.
  - The operations and thus the lambdas are executed in a deferred manner, i.e. deferred from the location they're written in the code.

- The laziness gets visible, when we put the *Stream*-expression/pipeline-creation in a method separate from the execution:

pipeline creation

```
public static Stream<Person> getAdultPersons() {
        Stream<Person> adultPersons
              = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Mary", 23))
                          .filter(person -> {
                                    boolean isAdult = person.getAge() >= 18;
                                    if (isAdult) {
                                            System.out.printf("%s is adult %n", person.getName());
                                    }
                                    return isAdult;
                            });
        return adultPersons;
}
```

pipeline execution

```
Stream<Person> adultPersonsStream = getAdultPersons();
List<Person> adultPersonsList = adultPersonsStream.collect(Collectors.toList());
```

Terminal

```
Ashley is adult
Mary is adult
```

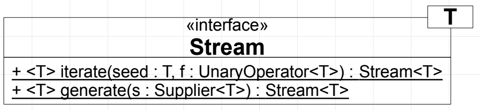
- *Streams* separate processing of items from the operations on items and the combination of operations from execution!

Unbounded Streams – Part 1

- Deferred execution (the separation of the combination of operations from execution) enables two important features:
  - (1) The execution of the pipeline can be optimized at the latest point in time, when all operations in the pipeline are determined.
    - Optimizations could be the coalescing of *Stream*s or skipping of stages, when they are not required for the computation of the result.
    - Lazy evaluation is required for dynamic parallelization to work properly.
  - (2) We can build unbounded *Stream*s of data (produce data), which usually doesn't even exist at the point the *Stream* is created.
    - An unbounded *Stream* can be potentially indefinite from the "producer's" perspective.

- Unbounded *Stream*s make a mighty concept, but it is just a concept: we cannot handle infinite data in a computer!
  - Remember, that infinity is a mathematical concept not a computer science concept.
  - Unbounded doesn't mean infinite, but that we have no direct access to elements, we do not know the count of elements a *Stream* will produce, even if we know the size of the source (e.g. the file-size is known, but the count of its lines is not known).

- The *Stream* interface provides some simple factories to produce infinite *Steams*:

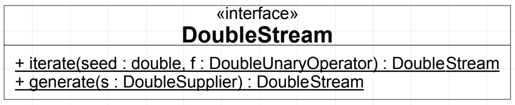

«interface»
Stream
T
+ <T> iterate(seed : T, f : UnaryOperator<T>) : Stream<T>
+ <T> generate(s : Supplier<T>) : Stream<T>

  - We already know *Stream.iterate()* from *IntStream*, but this overload offers no parameter to stop the iteration like *Predicate<T> hasNext*.
    - I.e. it allows to specify iteration similar to a for loop, but without an end condition.

  - Methods to create infinite *Stream*s are esp. among mathematicians/fp aficionados the most important methods in the *Stream* API.

Unbounded Streams – Part 2

- Instead of creating infinite generic *Stream*s, we create *DoubleStream*s, which also provide the relevant *Stream* operators.

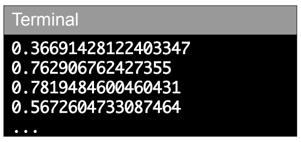

«interface»
DoubleStream
+ iterate(seed : double, f : DoubleUnaryOperator) : DoubleStream
+ generate(s : DoubleSupplier) : DoubleStream

  - We can simply create an infinite *DoubleStream* of random numbers:

```
DoubleStream randomNumbers = DoubleStream.generate(Math::random);
```

  - But, when we execute this pipeline to output the random numbers to the console, it'll take an unlimited amount of time:

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



```
Terminal
0.36691428122403347
0.762906762427355
0.7819484600460431
0.5672604733087464
...
```

- Terminal operations called on infinite *Streams* are catastrophic!
  - Terminal operations must consume the full *Stream*, i.e. read all elements of the *Stream*.
  - *Stream.forEach()*, *Stream.count()*, *Stream.collect()* or *Stream.toArray()* on an infinite *Stream* must take infinitely long.
  - Calling stateful operations is also catastrophic *Stream.sorted()*, *Stream.distinct()* on an infinite *Stream* must also take infinitely long.

Unbounded Streams – Short out Streams – Part 3

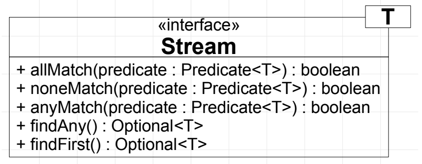
- Consider following terminal operations, which will block the program forever:

```
DoubleStream randomNumbers = DoubleStream.generate(Math::random);
```

```
// Will block: randomNumbers is an infinite Stream.
long count = randomNumbers.count();
```

```
// Will block: randomNumbers is an infinite Stream.
double[] randomNumbersArray = randomNumbers.toArray();
```

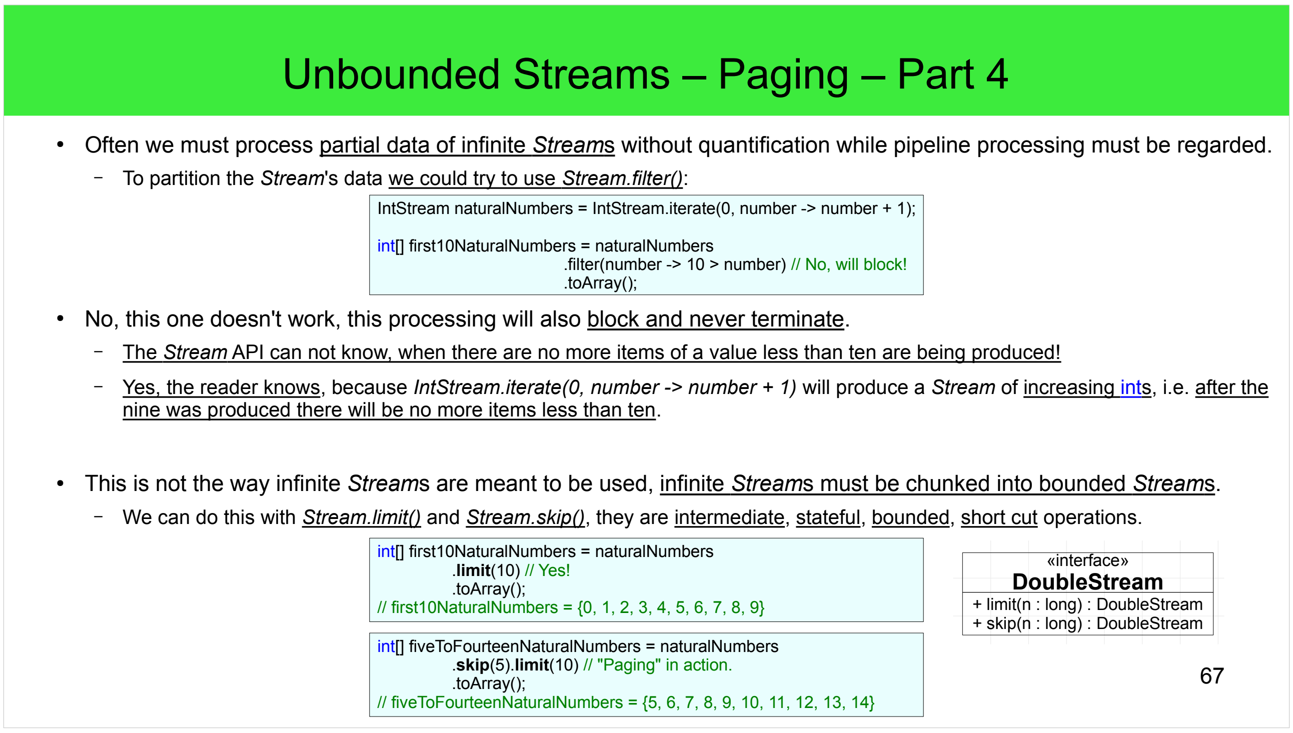
- The *Stream* API provides some terminal operations, which support cutting the execution short:


T
«interface»
Stream
+ allMatch(predicate : Predicate<T>) : boolean
+ noneMatch(predicate : Predicate<T>) : boolean
+ anyMatch(predicate : Predicate<T>) : boolean
+ findAny() : Optional<T>
+ findFirst() : Optional<T>

- The quantifiers *Stream.allMatch()*, *noneMatch()* and *anyMatch()* stop execution as soon as the terminal result is clear.
  - E.g. *Stream.anyMatch()* cuts execution short and returns true, as soon as an encountered element matches the predicate:

```
boolean hasGreaterThanPointFive = randomNumbers.anyMatch(number -> 0.5 > number);
// hasGreaterThanPointFive = true
```

- *Stream.findAny()* and *.findFirst()* combine quantifiers with returning matched elements, we'll discuss those in a future lecture.
  - Esp, their return type *Optional<T>* requires further discussion.



Unbounded Streams – Paging – Part 4

- Often we must process partial data of infinite *Streams* without quantification while pipeline processing must be regarded.
  - To partition the *Stream*'s data we could try to use *Stream.filter()*:

```
IntStream naturalNumbers = IntStream.iterate(0, number -> number + 1);

int[] first10NaturalNumbers = naturalNumbers
                                    .filter(number -> 10 > number) // No, will block!
                                    .toArray();
```

- No, this one doesn't work, this processing will also block and never terminate.
  - The *Stream* API can not know, when there are no more items of a value less than ten are being produced!
  - Yes, the reader knows, because *IntStream.iterate(0, number -> number + 1)* will produce a *Stream* of increasing ints, i.e. after the nine was produced there will be no more items less than ten.

- This is not the way infinite *Streams* are meant to be used, infinite *Streams* must be chunked into bounded *Streams*.
  - We can do this with *Stream.limit()* and *Stream.skip()*, they are intermediate, stateful, bounded, short cut operations.

```
int[] first10NaturalNumbers = naturalNumbers
                .limit(10) // Yes!
                .toArray();
// first10NaturalNumbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
int[] fiveToFourteenNaturalNumbers = naturalNumbers
                .skip(5).limit(10) // "Paging" in action.
                .toArray();
// fiveToFourteenNaturalNumbers = {5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

«interface» **DoubleStream**
+ limit(n : long) : DoubleStream
+ skip(n : long) : DoubleStream

67

- *Stream.skip()* and *Stream.limit()* do if you will access elements based on their virtual index in the *Stream*.

Unbounded Streams – Stateful operations – Part 5

- We should also discuss the effects of infinite *Streams* with stateful operations.
  - Following call to *DoubleStream.distinct()* followed by *DoubleStream.toArray()* will block:

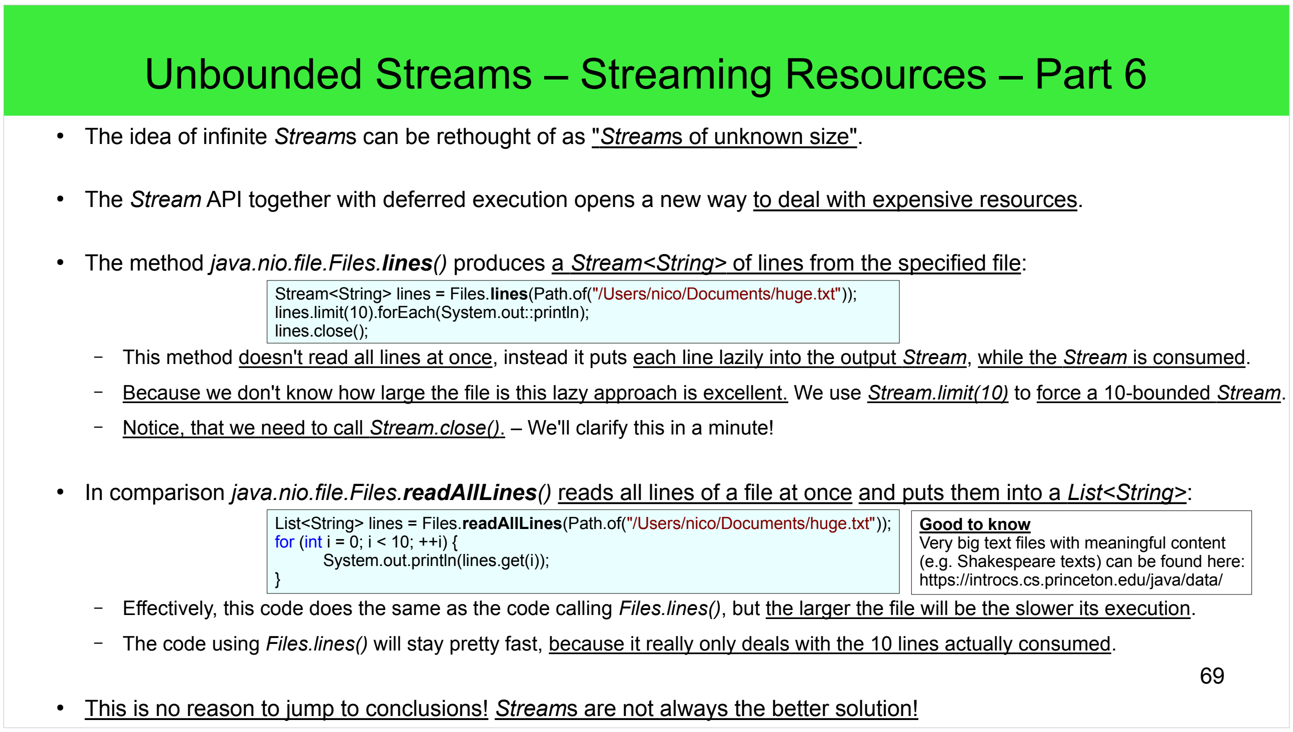
```
DoubleStream randomNumbers = DoubleStream.generate(Math::random);

DoubleStream distinctRandomNumbers = randomNumbers.distinct();
double[] distinctRandomNumbersArray = distinctRandomNumbers.toArray(); // No, will block!
```

  - Although *DoubleStream.distinct()* is an intermediate operation, it must buffer all elements, before it can create its output *Stream*!
  - And buffering an infinite count of elements, takes an infinite amount of time.

- Operations, which buffer all elements of its input *Stream* before the output *Stream* is created, are called intermediate unbounded stateful operations.

- In opposite, *Stream.limit()* and *Stream.skip()* are intermediate bounded stateful operations.

- Infinite *Streams* allow mighty yet simple algorithms, but we have to be careful with terminal and stateful operations.
  - When we use primitive *Stream* specializations, the operators *range()*/*rangeClosed()* should be preferred over unbounded operators.



Unbounded Streams – Streaming Resources – Part 6

- The idea of infinite *Stream*s can be rethought of as "*Stream*s of unknown size".
- The *Stream* API together with deferred execution opens a new way to deal with expensive resources.
- The method *java.nio.file.Files.**lines**()* produces a *Stream<String>* of lines from the specified file:

```
Stream<String> lines = Files.lines(Path.of("/Users/nico/Documents/huge.txt"));
lines.limit(10).forEach(System.out::println);
lines.close();
```

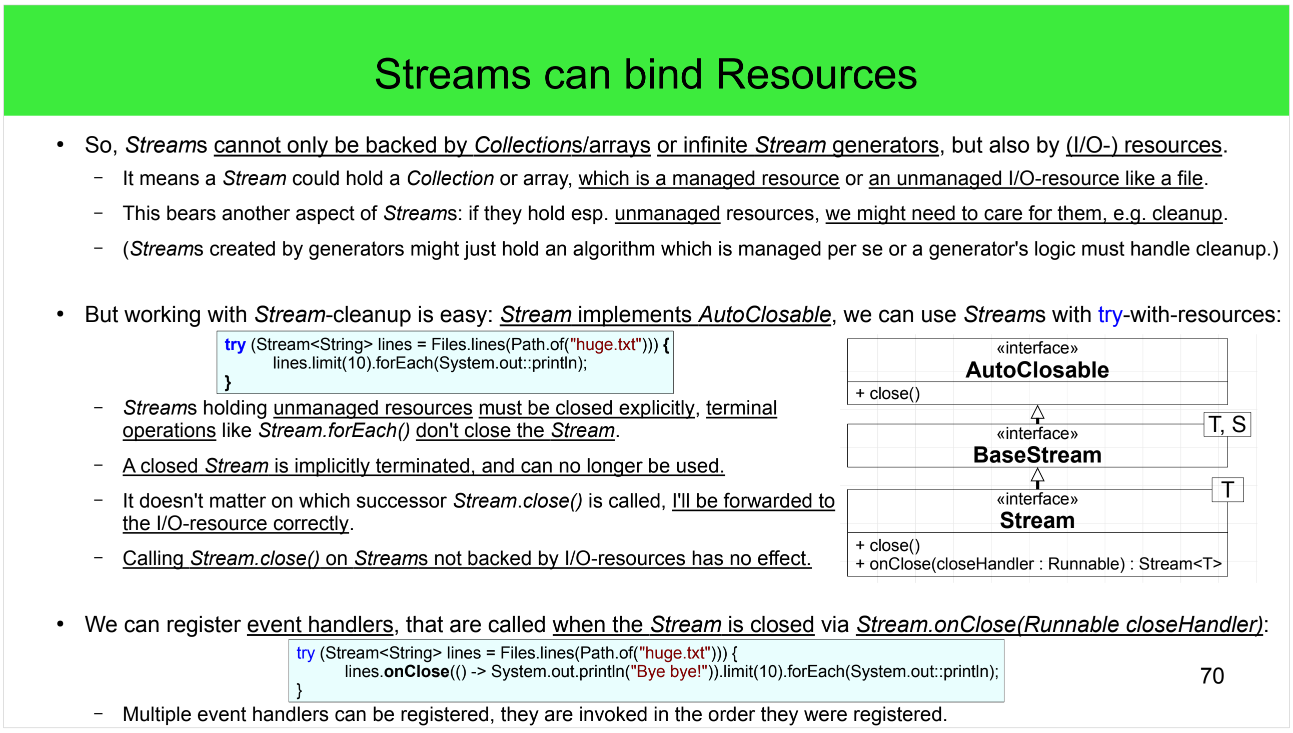
  - This method doesn't read all lines at once, instead it puts each line lazily into the output *Stream*, while the *Stream* is consumed.
  - Because we don't know how large the file is this lazy approach is excellent. We use *Stream.limit(10)* to force a 10-bounded *Stream*.
  - Notice, that we need to call *Stream.close()*. – We'll clarify this in a minute!
- In comparison *java.nio.file.Files.**readAllLines**()* reads all lines of a file at once and puts them into a *List<String>*:

```
List<String> lines = Files.readAllLines(Path.of("/Users/nico/Documents/huge.txt"));
for (int i = 0; i < 10; ++i) {
        System.out.println(lines.get(i));
}
```

Good to know
Very big text files with meaningful content (e.g. Shakespeare texts) can be found here: https://introcs.cs.princeton.edu/java/data/

  - Effectively, this code does the same as the code calling *Files.lines()*, but the larger the file will be the slower its execution.
  - The code using *Files.lines()* will stay pretty fast, because it really only deals with the 10 lines actually consumed.
- This is no reason to jump to conclusions! *Stream*s are not always the better solution!

- E.g. if a program needs to read the full file, e.g. to read data in specific locations in the file it can be better to read the file asynchronously (e.g. in a background thread).
  - Also the available RAM capacity to hold the file in memory plays a role for a reasonable solution.
  - => There is no single good solution, but the *Streams* API offers new tools for certain situations.
- Other *Streams* that must be closed:
  - *Streams* that result from Spring's *org.springframework.jdbc.core.JdbcTemplate. queryForStream()* also need to be closed because they hold a reference to the data source.
  - The same is true for *Streams* returned from Jakarta Persistence API *Querys*. *jakarta.persistence.Query* can provide a *Stream* via the method *Query.getResultStream()*, which must be closed.

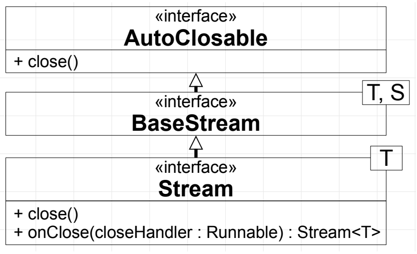


Streams can bind Resources

- So, *Stream*s cannot only be backed by *Collections*/arrays or infinite *Stream* generators, but also by (I/O-) resources.
  - It means a *Stream* could hold a *Collection* or array, which is a managed resource or an unmanaged I/O-resource like a file.
  - This bears another aspect of *Streams*: if they hold esp. unmanaged resources, we might need to care for them, e.g. cleanup.
  - (*Stream*s created by generators might just hold an algorithm which is managed per se or a generator's logic must handle cleanup.)
- But working with *Stream*-cleanup is easy: *Stream* implements *AutoClosable*, we can use *Stream*s with try-with-resources:

```
try (Stream<String> lines = Files.lines(Path.of("huge.txt"))) {
        lines.limit(10).forEach(System.out::println);
}
```

  - *Stream*s holding unmanaged resources must be closed explicitly, terminal operations like *Stream.forEach()* don't close the *Stream*.
  - A closed *Stream* is implicitly terminated, and can no longer be used.
  - It doesn't matter on which successor *Stream.close()* is called, I'll be forwarded to the I/O-resource correctly.
  - Calling *Stream.close()* on *Stream*s not backed by I/O-resources has no effect.



- We can register event handlers, that are called when the *Stream* is closed via *Stream.onClose(Runnable closeHandler)*:

```
try (Stream<String> lines = Files.lines(Path.of("huge.txt"))) {
        lines.onClose(() -> System.out.println("Bye bye!")).limit(10).forEach(System.out::println);
}
```

  - Multiple event handlers can be registered, they are invoked in the order they were registered.

- Because the origin, i.e. the source of *Streams* should be transparent of the consumer of the *Stream* handling *Stream.close()* is not so obvious.
  - Maybe, this is an indication, that *Streams* should rather be used locally, instead of being returned from a method or accepted as parameter.

Remember the Idiosyncrasies of Streams

- (1) Mind, that we can't reuse a *Stream* when it was terminated!
  - It forces us to write algorithms to process *Stream*s only once, the main reason is performance.

- (2) Understand deferred/lazy execution!
  - Esp. lambdas make *Stream* expressions look like imperative code, which executes as written, but this is not the case.
  - It is an easy task to introduce bugs due to deferred execution.
  - It is a hard task to debug code applying deferred execution.
  - Also *Exception*s are raised in a deferred manner!

- (3) Do not perform side effects in behavioral arguments (e.g. lambdas)!
  - Side effects do not work well with parallel execution.
  - Side effects can also happen in a deferred manner, which can introduce goofy and difficult to track down bugs.

- Tips:
  - *Stream*s should only be used in method implementations, not as field-, parameter- or return-types.
  - Mind or rather remember to use primitive *Stream* specializations if appropriate to avoid excessive boxing.

Streams and Performance

- *Stream*s are at most as performant as loops.
  - With few elements in a *Stream* the overhead is much higher when compared to loops.
  - The more elements and the higher the costs of the operation per element the less relevant than the cost of the overhead.

- Excessive boxing has significant performance costs, remember using primitive *Stream* specializations in case.

- We can also use parallel *Stream* processing to get better performance.
  - It only makes sense, if the overhead to split the work and join the results is less than the cost of the "real" operation in the pipeline.
  - Memory can become a bottleneck: when splitting a *LinkedList* with 10000 elements, the first 5000 must be iterated for the first split.
    - Splitting should result in equal chunks of work.
    - Splitting an array of primitive element-type has less overhead, it can also be easily projected into the CPU cache.
    - (With non-primitive arrays or *Collection*s further dereferencings are required for the operations to happen, which is costly.)
  - Of course, concrete benefits depend on the configuration of the system: #CPUs, cache, common threadpool parallelism etc.

- *Stream*s are rather not available for performance, but for productivity and expressibility.
  - Classical loops are generally faster than sequential *Stream*s, but parallel *Stream*s could be faster than classical loops.
  - Put simple, parallel processing is beneficial, if we have many data (large *Stream* source) and/or costly operations in the pipeline.

Thank you!