Java 8

Stream Puzzlers

Angelika Langer & Klaus Kreft

http://www.AngelikaLanger.com/

what we will do in this talk

- look at some surprising / not so surprising behavior

- show some Java 8 stream source code
- have a vote about its behavior / output
- let the code run
- discuss the reasons / background

speakers' relationship to topic

- independent trainer / consultant / author
  - teaching C++ and Java for ~20 years
  - curriculum of some challenging seminars
  - providing consulting services for ~20 years
  - JCP observer and Java champion since 2005
  - authors of "Effective Java" column
  - author of Java Generics FAQ and Lambda Tutorial & Reference

let's get started ...

parallel `forEach()`

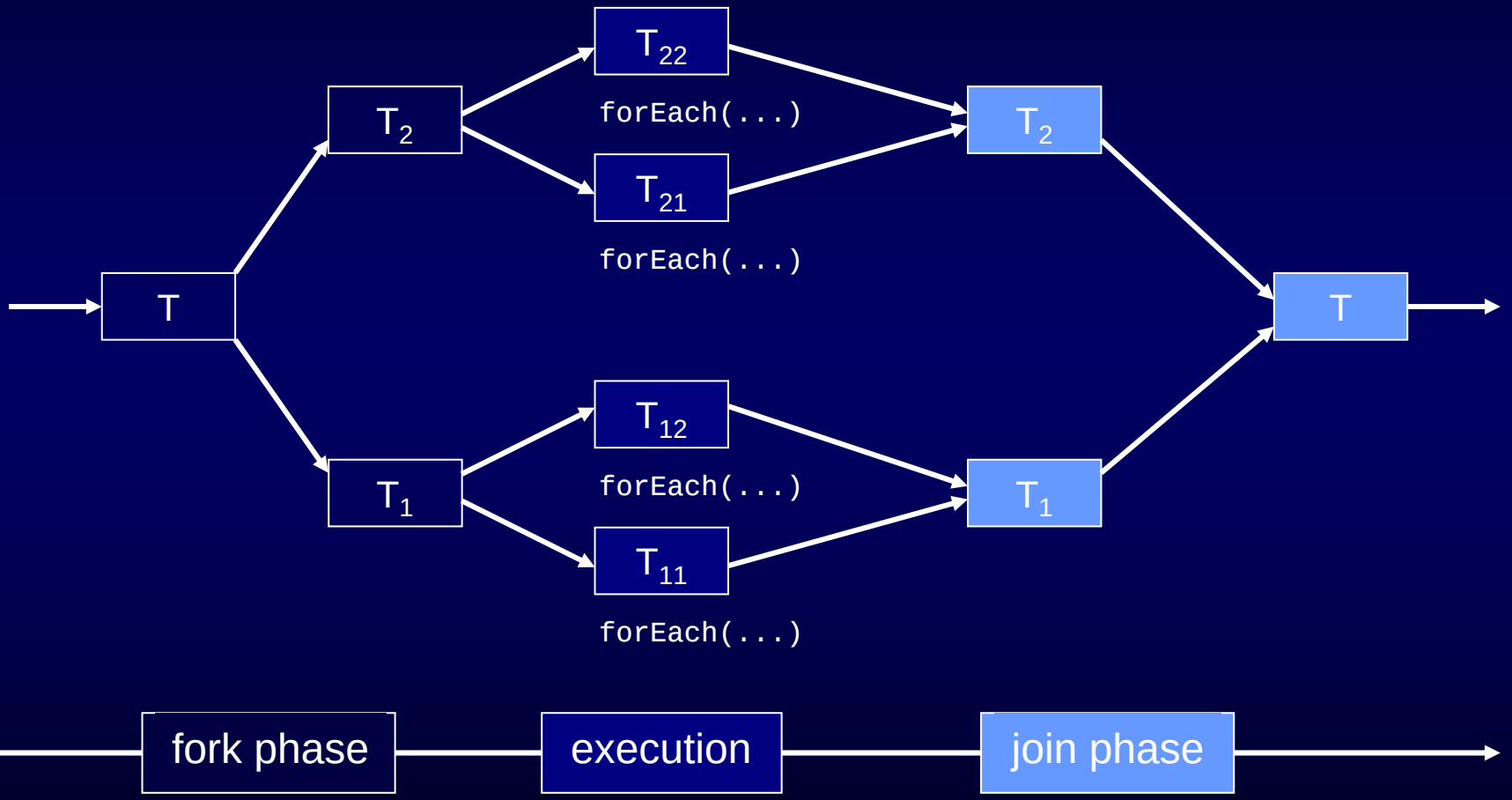
puzzler #1 – explained

- parallel !!!

- javadoc (`forEach()`):
  The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

parallel forEach()

```
forEach(s -> System.out.print(s))
```



let's use another stream operation ...

parallel `reduce()`

puzzler #2 – explained

- javadoc (`reduce()`):

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

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

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

but is not constrained to execute sequentially.

puzzler #2 – explained (cont.)

- javadoc also says:

  The `identity` value must be an identity for the `accumulator` function. This means that for all `t`, `accumulator.apply(identity, t)` is equal to `t`.

  The `accumulator` function must be an associative function.

- these requirements are important
  - ensure: order preserving when executed in parallel

puzzler #2 – identity

- *The `identity` value must be an identity for the `accumulator` function. This means that for all `t`, `accumulator.apply(identity, t)` is equal to `t`.*

- our example

```
reduce("", (s1, s2) -> s1 + s2)
```

because: `("" + s).equals(s)`
for all `String s`

puzzler #2 – associative

- *The `accumulator` function must be an associative function.*

- commutative: op(a,b) = op(b,a) or a○b = b○a
  - example: `max` for `int`, "`+`" for `int`

not commutative

  - associative: op(op(a,b),c) = op(a,op(b,c)) or
    ((a○b)○c) = (a○(b○c))
    - example: "`+`" for `String` (string concatenation)
      - `"hello"+"world"` **differs from** `"world"+"hello"`
      - `(("one"+"two")+"three")` **same as** `("one"+("two"+"three"))`

not associative

  - example: "`-`" for `int`
    `3-1` **differs from** `1-3`
    `((3-2)-1)` **differs from** `(3-(2-1))`

puzzler #2 - requirements ignored

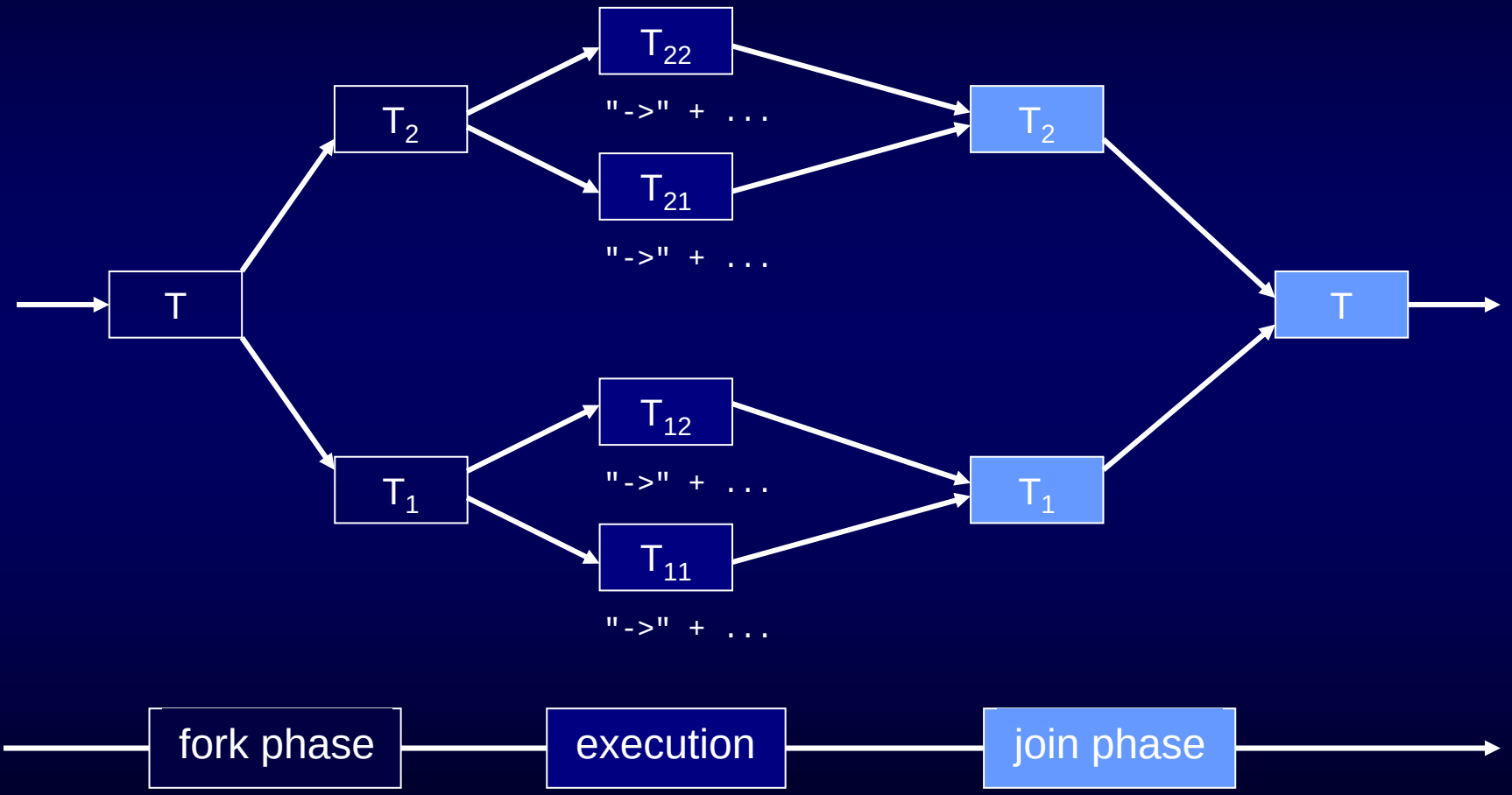
- what if we violate the requirements ... ?
  - use non-identity
  - use non-associative accumulator

let's ignore identity ...

parallel `reduce()` - with non-identity

parallel reduce()

```
reduce("->", (s1, s2) -> s1 + s2)
```



puzzler #2 – violate associativity

- use as reduction operation

```
reduce("", (s1, s2) -> toggle(s1) + s2)
```

  - where `toggle()` turns
    - the upper case characters from `s1` to lower case, and
    - the lower case characters form `s1` to upper case

```
String toggle(String in) {
  char[] chars = in.toCharArray();
  char[] buf = new char[chars.length];
  for (int i=0;i<chars.length;i++) {
    if (Character.isLowerCase(chars[i]))
      buf[i] = Character.toUpperCase(chars[i]);
    if (Character.isUpperCase(chars[i]))
      buf[i] = Character.toLowerCase(chars[i]);
  }
  return new String(buf);
}
```

puzzler #2 – violate associativity (cont.)

- use as reduction operation

```
reduce("", (s1, s2) -> toggle(s1) + s2)
```

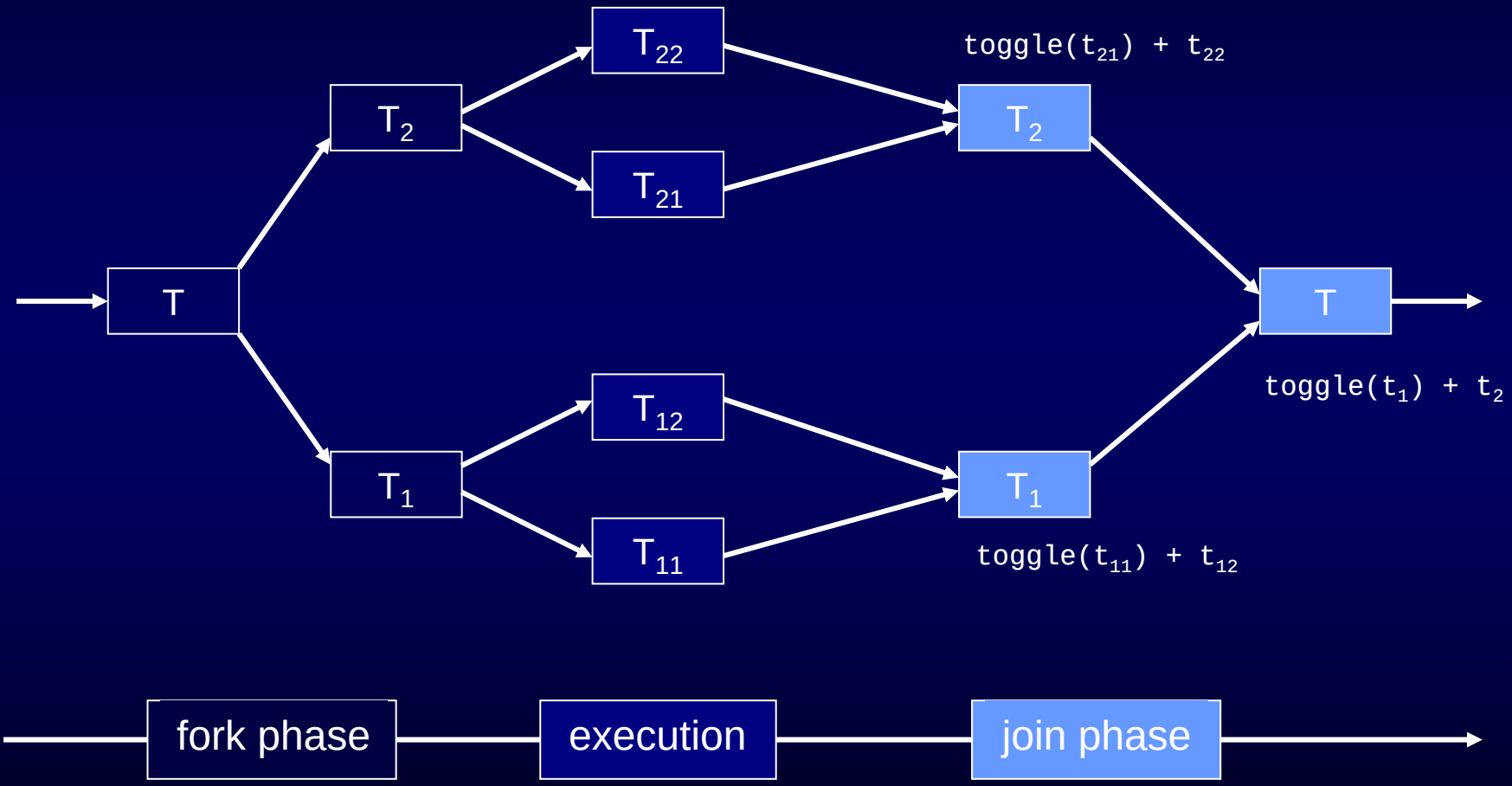
  - `toggle()` is not associative
    - (a○b)○c -> Ab○c -> aBc
    - a○(b○c) -> a○Bc -> ABc

let's ignore associativity ...

parallel `reduce()` - non-associative accumulator

parallel reduce()

```
reduce("", (s1, s2) -> toggle(s1) + s2)
```



parallel reduce()

- sequential reduce()

  (((((((a○b)○c)○d)○e)○f)○g)○h) => AbCdEfGh

- parallel reduce() with split in halves

  ((a○b)○c)○d) ○ ((e○f)○g)○h) => aBcDEfGh

- parallel reduce() with split in quarters

  ((a○b) ○ (c○d)) ○ ((e○f) ○ (g○h)) => AbcDeFGh

puzzler #2a/b – hint

- violating the `accumulator` requirements cause
  the results produced by parallel streams to be wrong

- but also not okay for sequential streams
- extremely fragile code
  - adding `parallel()` leads to wrong results
  - can easily happen when the responsibility for code is shared
    - typical for an agile process

other stream sources

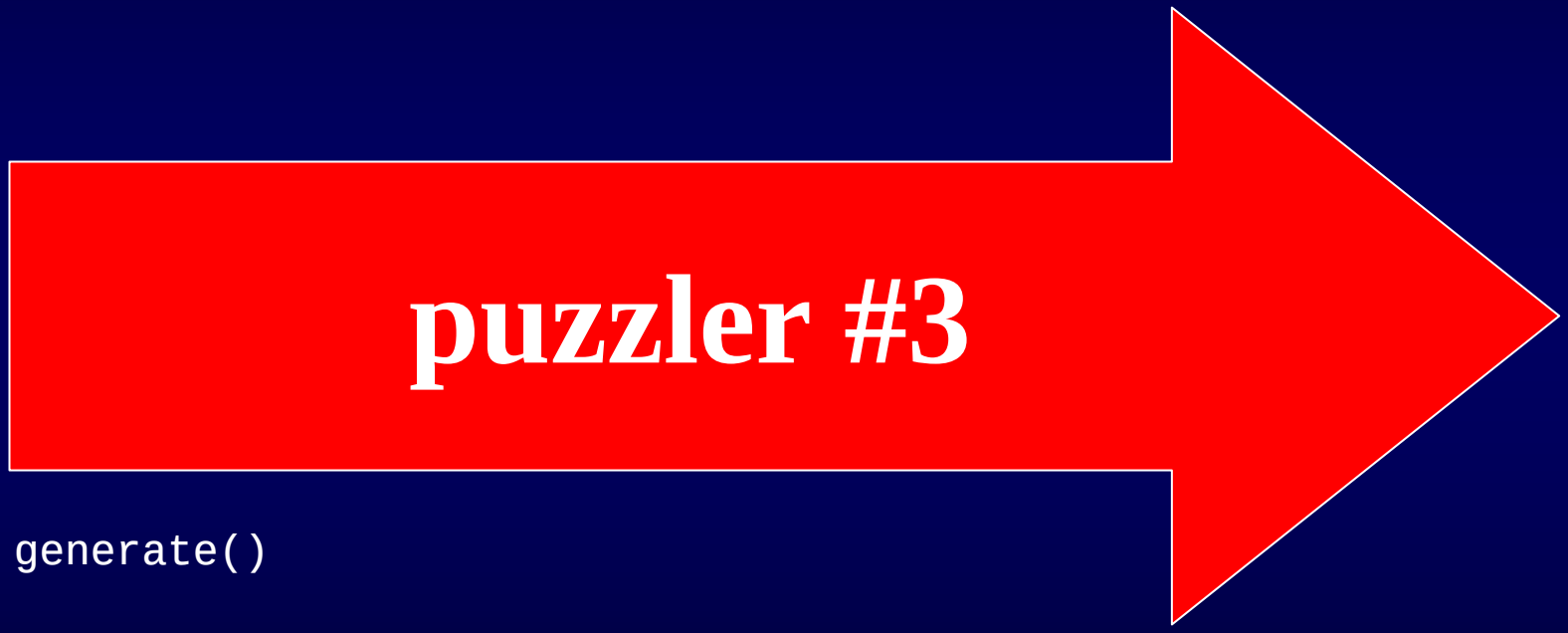
- streams can be generated

- stream operation `generate()`

```
static <T> Stream<T> generate(Supplier<T> s)
```

… each element is generated by the provided `Supplier`.

let's use a generated stream ...




```
generate()
```

puzzler #3 – explained

- javadoc (`generate()`):

  Returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

- we have used `generate()` incorrectly

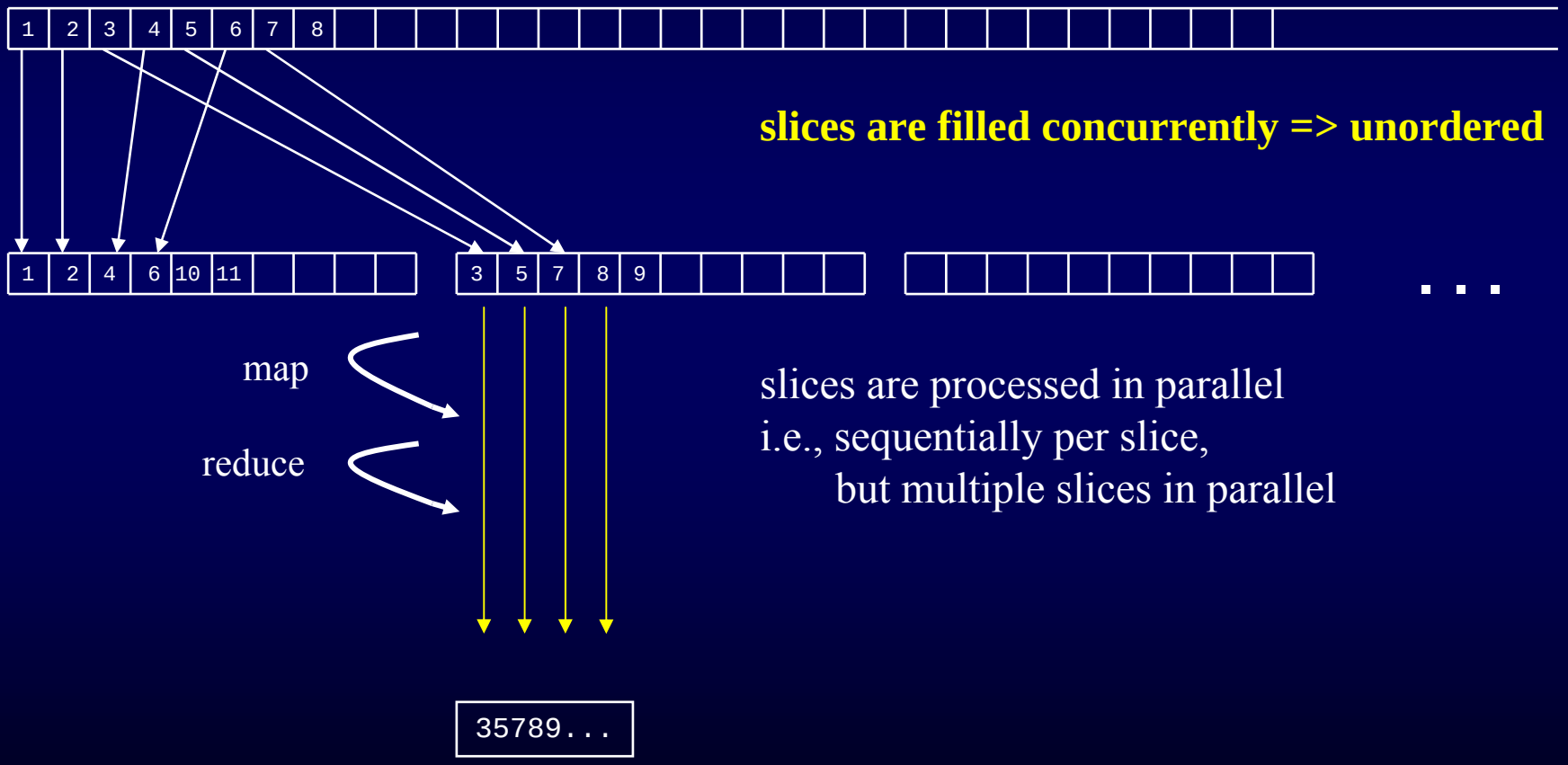
- as before: fails when executed in parallel
  - but sequential code is fragile and also not recommended

splitting unordered infinite streams

- unordered infinite streams have a special spliterator
  - of type `StreamSpliterators.UnorderedSliceSpliterator`
- creates stream *slices*
  - each slice is filled with generated elements
  - concurrently by several threads
- each task
  - iterates over a slice
  - applies intermediate/terminal operation

splitting infinite streams

```
generate(...).parallel().map(...).reduce(...)
```



slices are processed in parallel
i.e., sequentially per slice,
but multiple slices in parallel

another stream generator

- stream operation `iterate()`

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

… iterative application of a function `f` to an initial element `seed`, producing a Stream consisting of `seed`, `f(seed)`, `f(f(seed))`, etc.

let's use another stream generator ...


```
iterate()
```

puzzler #4 – explained

- javadoc (`iterate()`):

  Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.

  The first element (position 0) in the Stream will be the provided seed. For n > 0, the element at position n, will be the result of applying the function f to the element at position n - 1.

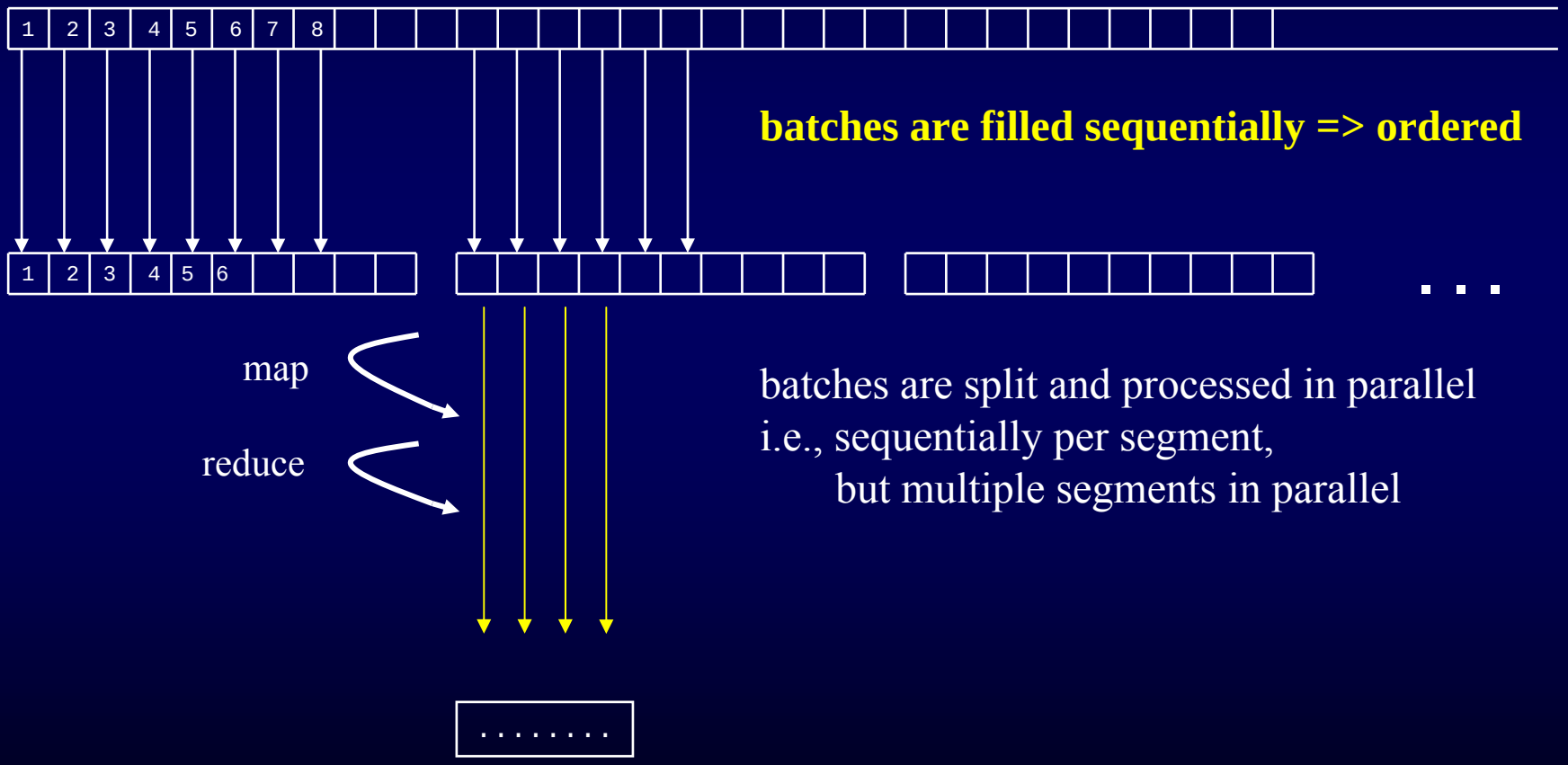
- this time we have done it correctly

splitting ordered infinite streams

- ordered infinite streams use a spliterator
  - of type `Spliterators.IteratorSpliterator`
- creates *batches*
  - each batch is filled with generated elements
  - sequentially by one thread
    - not necessarily always the same thread
      - next batch might be filled sequentially by another thread
- each task
  - iterates over a segment of a batch
  - applies intermediate/terminal operation

splitting infinite streams

```
iterate(...).parallel().map(...).reduce(...)
```



batches are filled sequentially => ordered

batches are split and processed in parallel
i.e., sequentially per segment,
but multiple segments in parallel

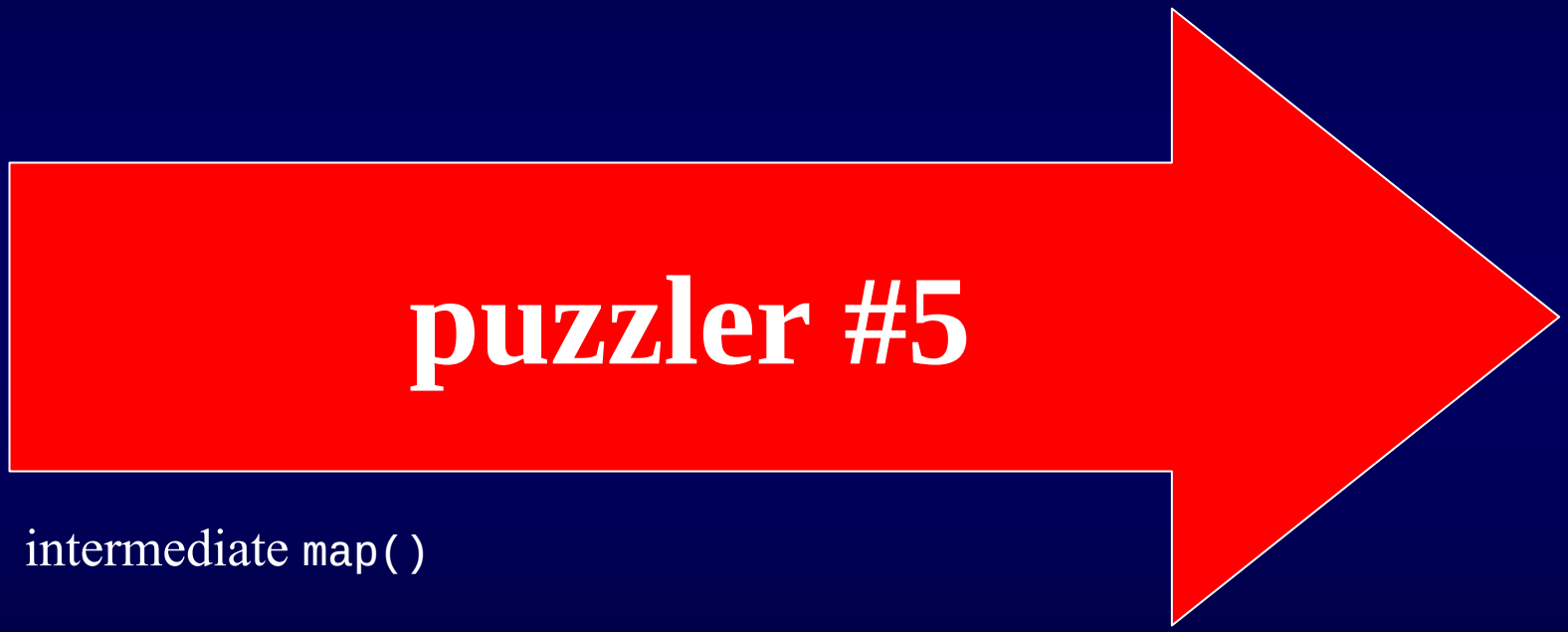
order hint

- stream source and terminal operation must be ordered

  then the result/effect is ordered

- figure that out from the javadoc, or (some simple rules):
  - arrays and all collections (except `HashSet`) are ordered
  - terminal operations
    - `reduce()`, `forEachOrdered()` are ordered
    - `forEach()` unordered
    - `collect()` depends on how Characteristics
      `Collector.Characteristics.UNORDERED` and
      `Collector.Characteristics.CONCURRENT`
      are set for the `Collector`

intermediate operations and order ...

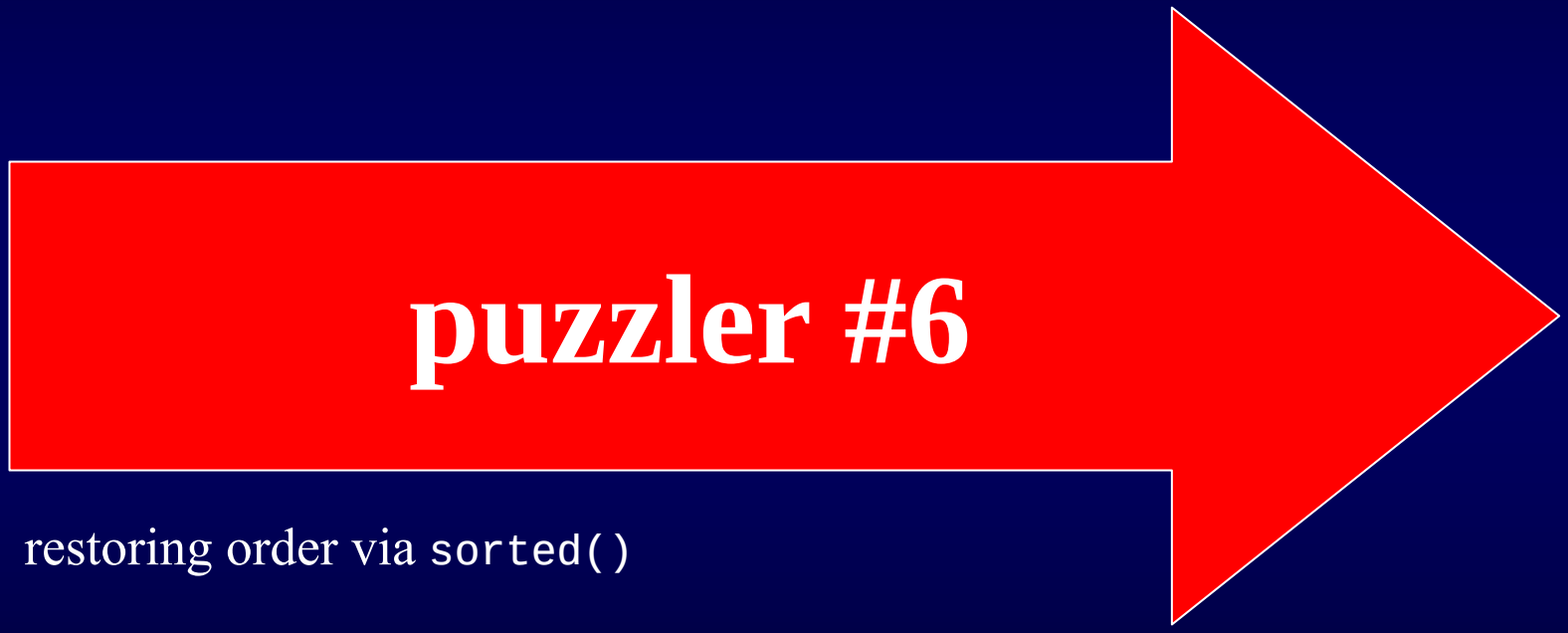


intermediate `map()`

order hint: intermediate operations

- intermediate operations have no effect on order
- except:
  - the intermediate operation `sorted()` restores order

let's put order into an unordered source ...



restoring order via `sorted()`

puzzler #6 – explained

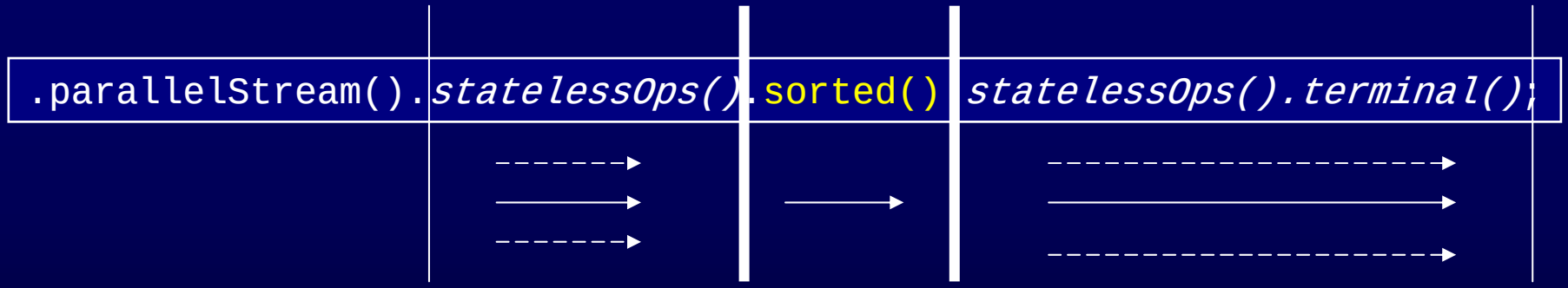
- unfortunately the javadoc (of `sorted()`) is not of much help

- there is only a small hint:

  This is a stateful intermediate operation.

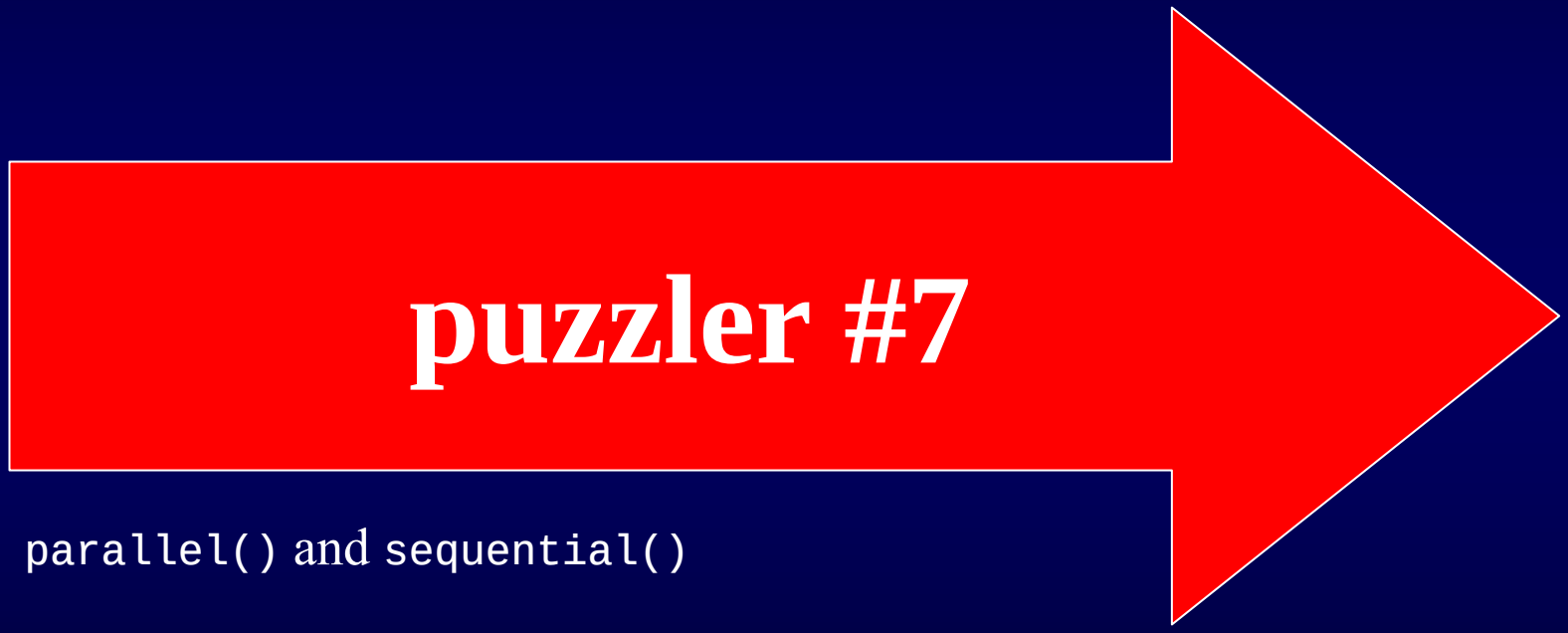
puzzler #6 – explained (cont.)

- need to have a look at the implemention
  - `sorted()` is implemented with two barriers, i.e.
    - stream elements are collected before the actual sort (1st barrier),
    - then the sort is performed with the collected elements, and then
    - the resulting elements are collected again after the sort (2nd barrier) ,
      - before the next operations start



- first barrier leads (already) to `OutOfMemoryError`
  - because it is an infinite stream that is `generated()`

let's explore parallel() / sequential() ...



`parallel()` and `sequential()`

puzzler #7 – explained

- unfortunately the javadoc of `sequential()` is not of much help

- there is only a small hint:

  May return itself, either because the stream was already sequential, or because the underlying stream state was modified to be sequential.

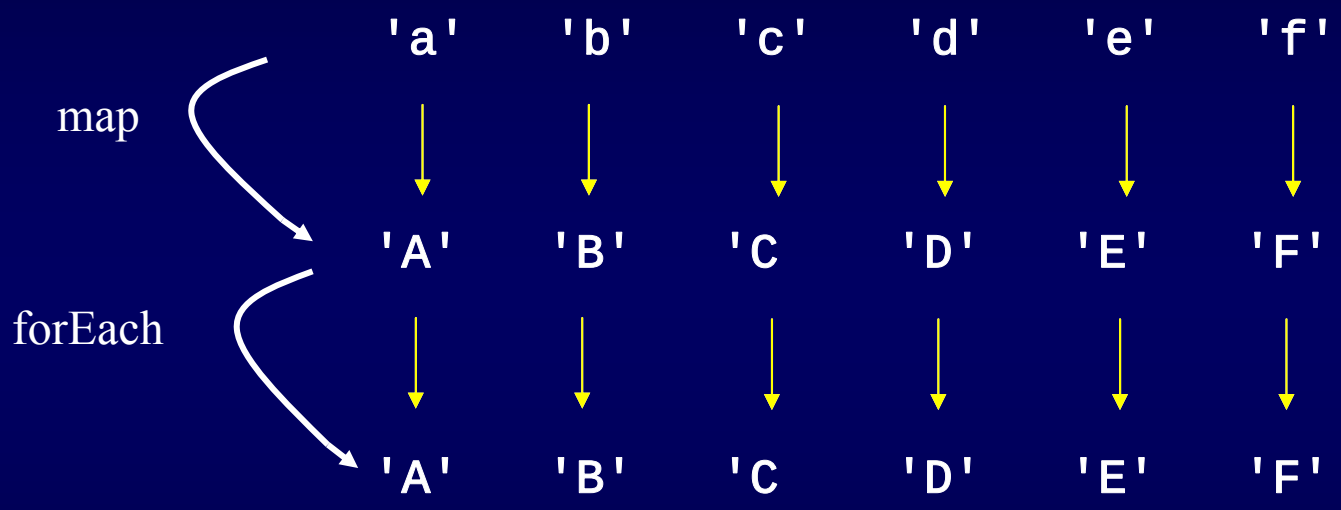
- that's how its done in (all) stream implementations
  - `parallel()` / `sequential()` flip a flag (stream state)

puzzler #7 – explained (cont.)

- and
  - intermediate operations are lazy
    - not executed immediately
  - terminal operations are eager
    - trigger the execution of all previous intermediate operations, and the terminal operation

pipeline

```
Arrays.stream(chars)
      .map(s -> toggle(s))
      .forEach(s -> System.out.print(s));
```



code looks like

really executed

puzzler #7 – explained (cont.)

```
Arrays.stream(chars).parallel()
      .map(s -> toggle(s))
      .sequential()
      .forEach(s -> System.out.print(s));
```

set parallel

set sequential

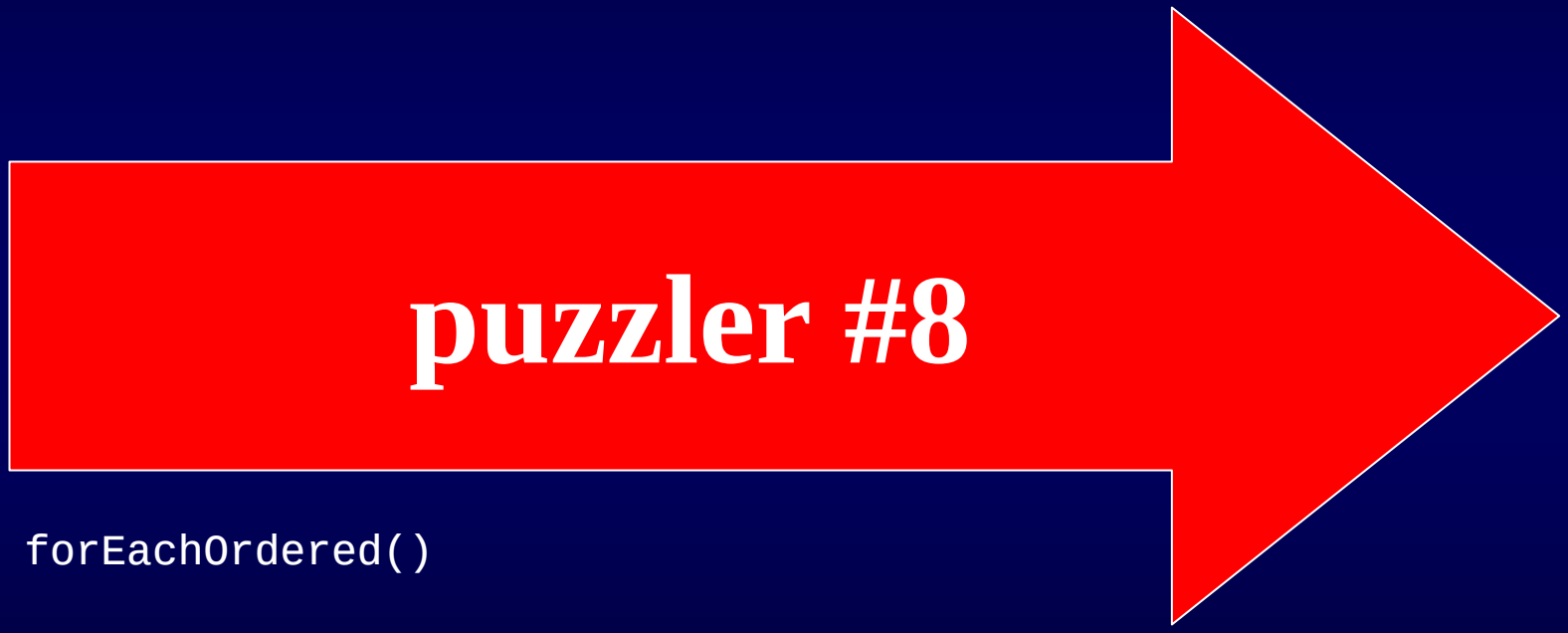
trigger all stream operations, with mode set to sequential

difference between ...

… stream operations and other methods

- stream operation act upon
  the elements of the underlying stream source
  - defined in `Stream<T>`, `IntStream`, `LongStream`, `DoubleStream`
- other operations
  - stream maintenance / management
    - e.g. `parallel()`, `sequential()`
  - defined in `BaseStream`
    - super-interface to `Stream<T>`, `IntStream`, `LongStream`, `DoubleStream`

let's explore forEachOrdered() ...



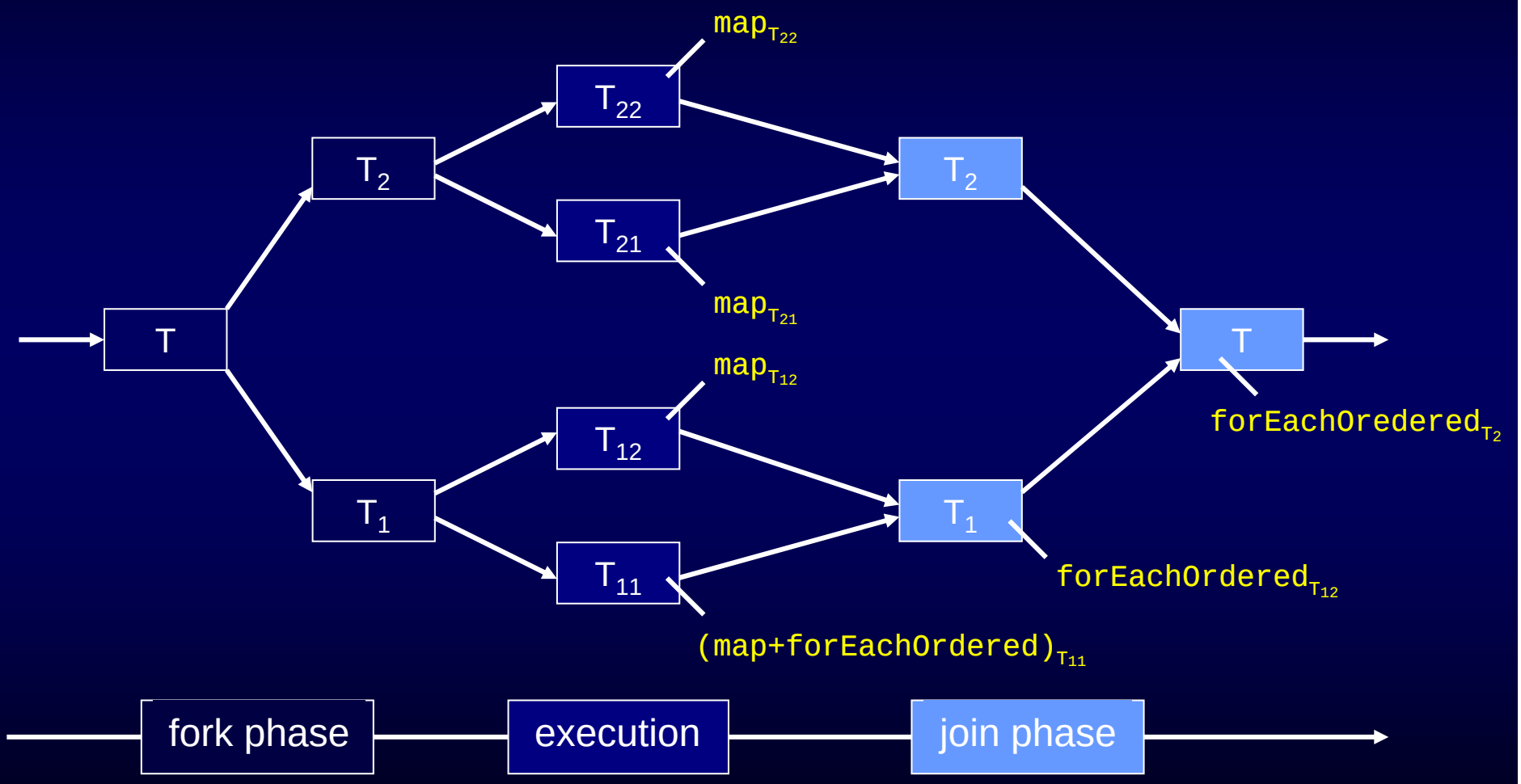

```
forEachOrdered()
```

puzzler08 – why is it so

- this time we have done it correctly
- javadoc (`forEachOdered()`):
  This operation processes the elements one at a time, in encounter order if one exists.
- what the javadoc does not say:
  often slower than `forEach()`

forEachOrdered()

```
map(s->toggle(s)).forEachOrdered(s->System.out.println(s));
```



wrap-up

- preservation of encounter order requires:
  - underlying stream source must be ordered, or
  - intermediate operation `sorted()` creates order, and
  - terminal operation must be ordered

wrap-up

- ordered stream sources
  - arrays, collections (exception HashSet), iterate(), ...
- unordered stream sources
  - HashSet, generate(), ...

- ordered terminal operations
  - reduce(), forEachOrdered(), collect(toCollection()), ...
- unordered terminal operations
  - forEach(), collect(toConcurrentMap()), ...

authors

Angelika Langer

Klaus Kreft

www.AngelikaLanger.com

Q & A

source code

www.AngelikaLanger.com\Conferences\Code\jDays2017\Main.java