# FUNCTIONAL AND CONCURENT PROGRAMMING

SI4

Pascal URSO

# OPTIONAL

# OPTION CONCEPT

- How to represent the asbence of result?
  - Nothing matches? (e.g. an even prime greater than 2)
  - Result not ready?
  - Object not present in database?
  - …
- null?
  - NullPointerException risk!!!
  - *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965"* (T. Hoare, 2009) – ALGOL W

# NULLPOINTEREXCEPTION

- Example
  - `String version = computer.getSoundcard().getUSB().getVersion();`
- What if computer has no sound card? no usb? no version?
  - ```
    String version = "UNKNOWN";
    if(computer != null){
        Soundcard soundcard = computer.getSoundcard();
        if(soundcard != null){
            USB usb = soundcard.getUSB();
            if(usb != null){
                version = usb.getVersion();
            }
        }
    }
    ```


    ```

# ALTERNATIVES

- Groovy, C#: operator ?.
  - `String version = computer?.getSoundcard()?.getUSB()?.getVersion();`
- Haskell, Scala, Python, Rust, …: a new type (called MayBe, Option, …)
  - Encapsulate an optional value
- In Java (Since 8)
  - java.util.Optional<T>
  - And OptionalInt, OptionalDouble, OptionalLong

# OPTIONAL : (VERY) BASIC EXAMPLE

- ```
  Optional<String> optional = ...
  if (optional.isPresent()) {
      System.out.println(optional.get());
  } else {
      System.out.println("The Optional is empty");
  }
  ```
- What is the interest?
  - Still quite verbose
  - Absence check at compilation (*Warning 'Optional.get()' without 'isPresent()' check)*
  ```

# OPTIONAL : MAIN METHODS

| Method | Effect | if empty |
| --- | --- | --- |
| static <T> Optional<T> empty() | returns an empty Optional instance | |
| static <T> Optional<T> of(T value) | encapsulate a non-null value | |
| static <T> Optional<T> ofNullable(T value) | encapsulate a value | empty if value is null |
| T orElse(T other) | returns the value | return other |
| void ifPresent(Consumer<? super T> action) | performs the action on the value | does nothing. |
| <U> Optional<U> map(Function<? super T,? extends U> mapper) | apply the mapper | returns an empty Optional |
| <U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper) | | |
| … SEE API | | |

- **To avoid : get(), isPresent(), isEmpty()**

# OPTIONAL : BETTER EXAMPLES

- ```
  Optional<String> optional = ...
  System.out.println(optional.orElse("The Optional is empty"));
  ```

- ```
  String name = computer.getSoundcard()
          .flatMap(Soundcard::getUSB)
          .flatMap(USB::getVersion)
          .orElse("UNKNOWN");
  ```

with:

- ```
  static class Computer {
      public Optional<Soundcard> getSoundcard()
  ```


  ```

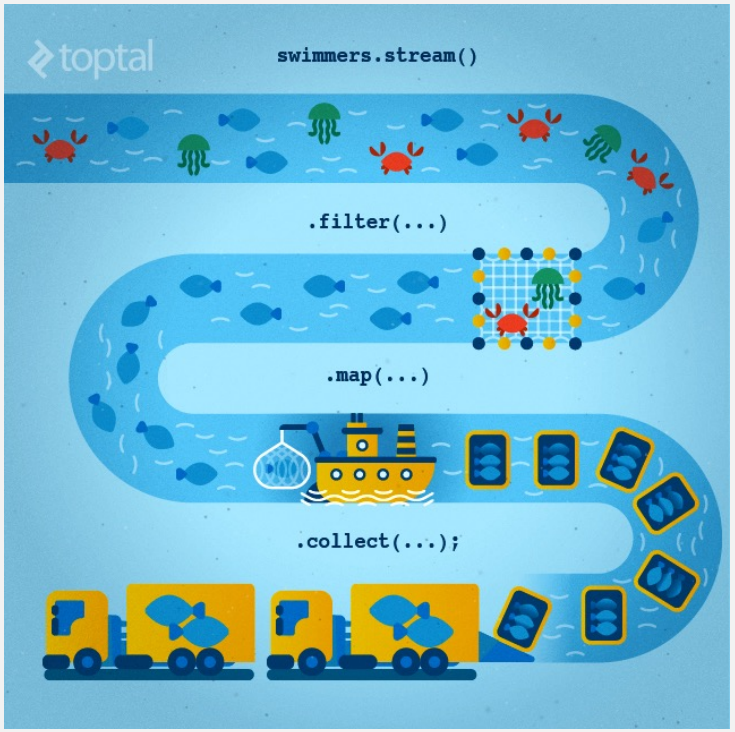
# OPTIONAL USAGE EXAMPLES

- Configuration
  - HttpClient::proxy -> Optional
  - Runtime.Version::build -> Optional
- **JEE (Distributed System) requests**
  - JAX-RS (Java API for RESTful Web Services): Response::get -> Optional
  - CDI (Contexts and Dependency Injection):  Instance::get -> Optional
  - EJB (Enterprise JavaBeans): EntityManager::find -> Optional
  - JPA (Java Persistence API): Query::getSingleResult -> Optional
- **Computation**
  - java.util.stream

# STREAMS

# STREAMS!

Source



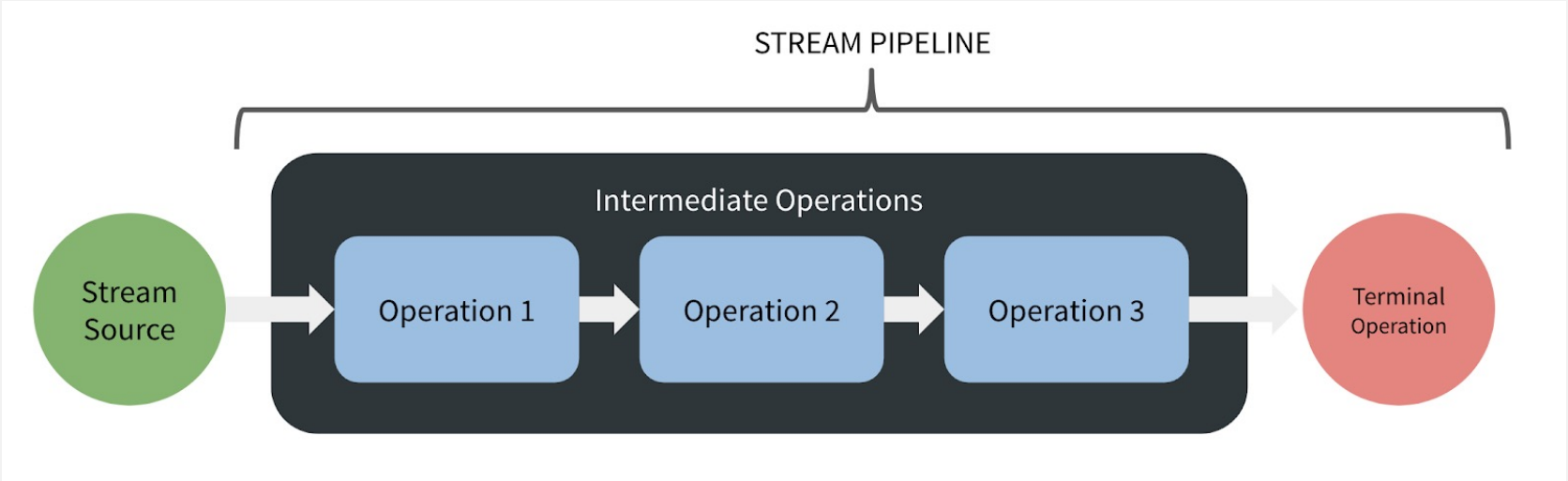
# STREAM PROPERTIES

- **Process** a sequence of elements
  - Closer to an iterator than a collection
- **Lazy** evaluation
  - Elements are processed on demand
  - And can even be produced on demand
- **Adapted to ditributed programming**
  - Immutable
  - Sequential or parrallel
  - Ordered or non ordered

# STREAM SOURCES

- Collections (java.util) / Arrays
  - Methods stream(), parrellelStream()
  - E.g. `Stream<Integer> stream = List.of(1,2,3,5,7).stream();`
- I/O channels: Files, BufferedReader, Scanner, …
  - E..g. `Stream<String> stream = Files.lines(Paths.get("file.txt"));`
- Generated values (methods of Stream<T>): **infinite streams!**
  - `static <T> Stream<T> generate(Supplier<? extends T> s)` (unordered)
  - `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)` (ordered)
- …

# STREAM OPERATIONS: PIPELINE



# INTERMEDIATE OPERATIONS

- Stateless method
  - map, filter, peek, flatMap, unordered, …
- Statefull method
  - sorted, limit, skip, distinct, …
- Applied to the stream
  - Modify its "internal state" but not its "source" (lazy)
  - `IntStream.iterate(0, x -> x + 1).filter(x -> x % 2 == 0); // OK`
- Final result may depand on stream (un)ordering **(see API)**

# INTERMEDIATE OPERATIONS: BUILDER PATTERN

- Intermediates methods return a stream (a new modified version)
  - ```
    DoubleStream s = DoubleStream.iterate(0, x -> x + Math.PI/2)
            .map(Math::cos)                    // [1,0,-1,0,1,0,-1,0,1...]
            .limit(6)                          // [1,0,-1,0,1,0]
            .filter(x -> x > 0);               // [1,1]
    ```
- But the original stream is closed, e.g.
  - ```
    IntStream s = IntStream.of(1, 2, 3, 5, 7);
    IntStream s1 = s.filter(x -> x > 4);    // s1 != s nor equals
    IntStream s2 = s.filter(x -> x < 6);    // s1.filter(x -> x < 6); is OK
    ```

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

at java.base/java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)

at java.base/java.util.stream.IntPipeline.<init>(IntPipeline.java:91)

# TERMINAL OPERATIONS

- Non-mutable operations
  - **reduce(f),** reduce(id, f), reduce(id, f, c)

short-circuit

  - **findFirst()**, **findAny()**, **anyMatch(predicate)**, allMatch(predicate), noneMatch(predicate)
  - forEach(consumer)

reductions

  - **max(comp)**, **min(comp)** / max(), min(), average() for IntStream, DoubleStream
  - count() / sum() for IntStream, DoubleStream
- Close the stream
- Optional result if needed

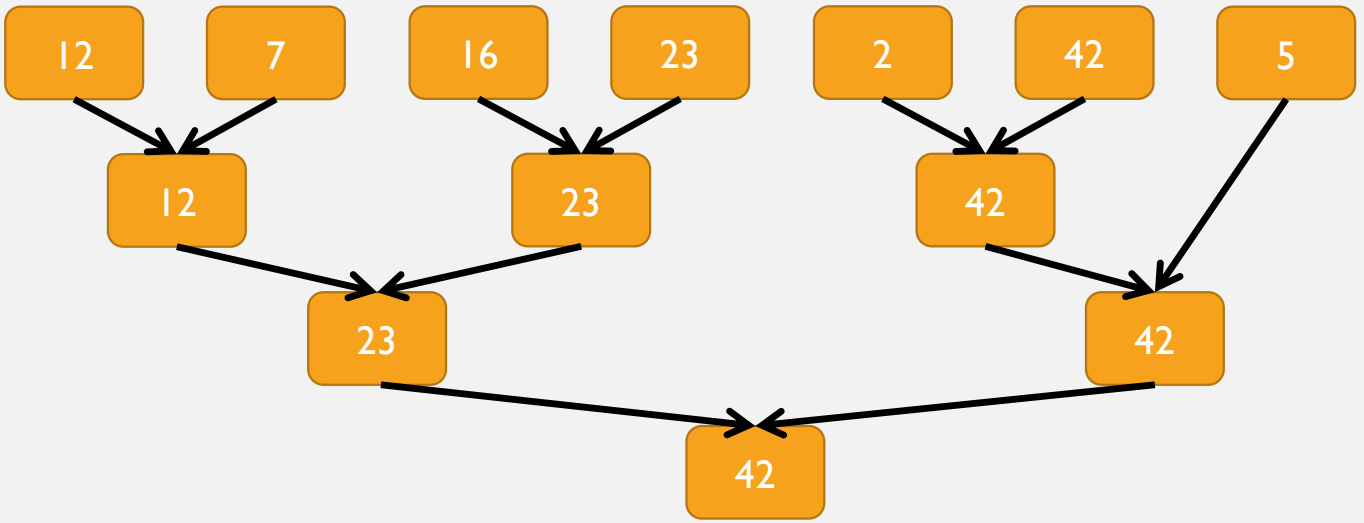
# COLLECT

- Mutable reduction
  - ```
    <R> R collect(Supplier<R> supplier,                    // Identity
                  BiConsumer<R, ? super T> accumulator,    // Adds an element
                  BiConsumer<R, R> combiner)               // Combine collections
    ```
  - ```
    List<T> l = stream.collect(ArrayList::new,
                               ArrayList::add,
                               ArrayList::addAll);
    ```
- Using collectors
  - ```
    <R,A> R collect(Collector<? super T,A,R> collector)
    ```
  - ```
    List<T> l = stream.collect(Collectors.toList());
    ```

# REDUCTION AND PARALLELISM

max

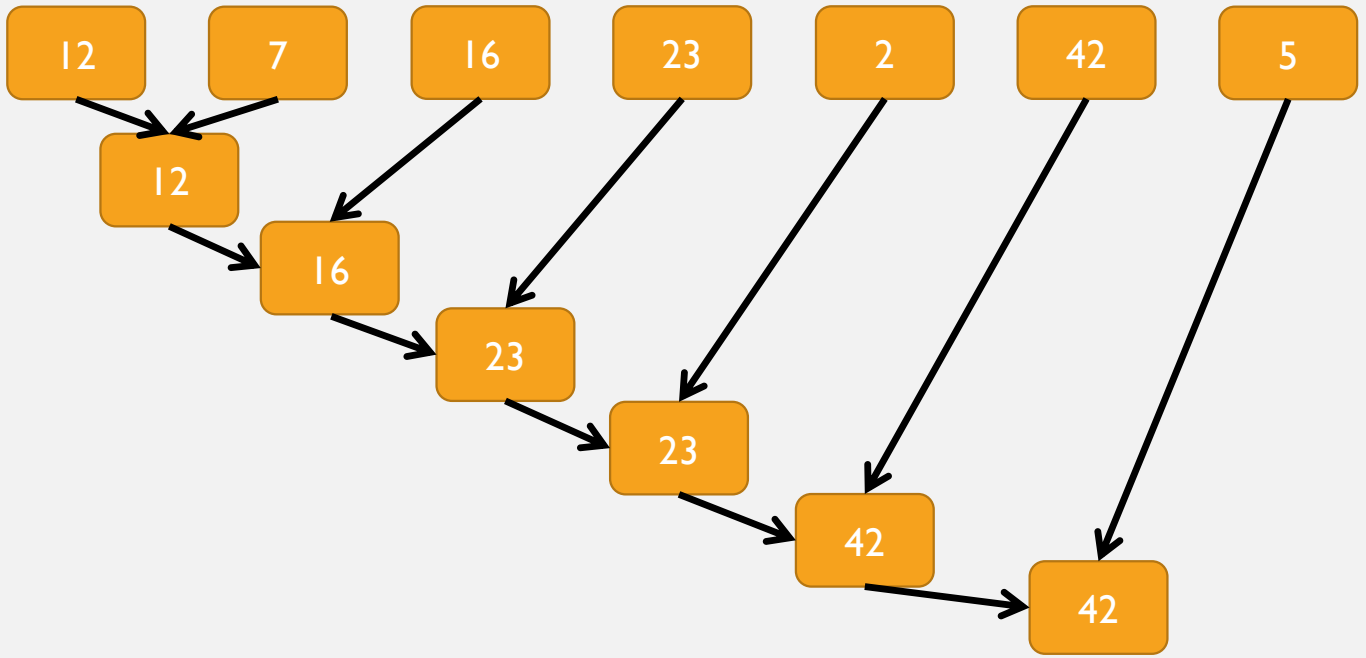
parallel
unordered
stream



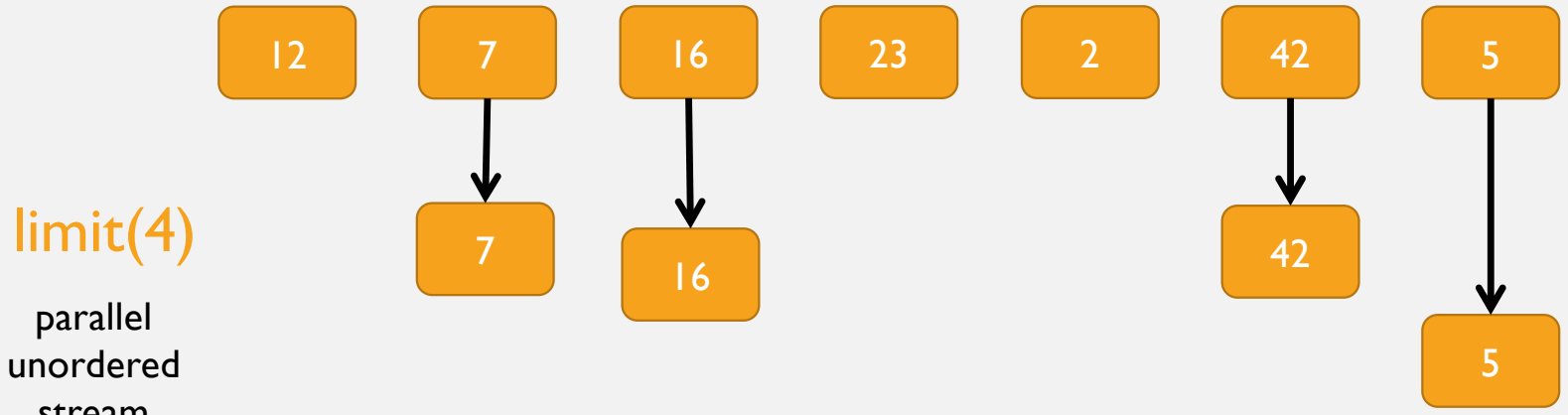
# REDUCTION AND PARALLELISM

max

sequential
**or** ordered
stream



# REDUCTION AND PARALLELISM

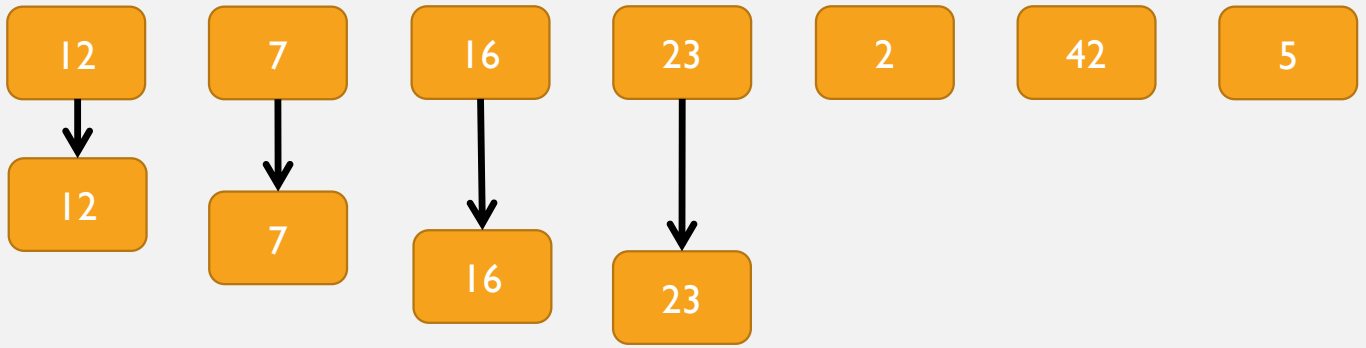



limit(4)

parallel
unordered
stream

possible result [16, 5, 42, 7]

# REDUCTION AND PARALLELISM




limit(4)

sequential
or ordered
stream

result [12, 7, 16, 23]