Functional-style data processing with stream

Streams

Have a full understanding of what streams are.

How you can use them concisely and efficiently.

Introducing streams

Stream operations

Parallel streams

Advantages and disadvantages of streams

Introducing Streams

What is a stream?

Collections versus streams

Working with stream

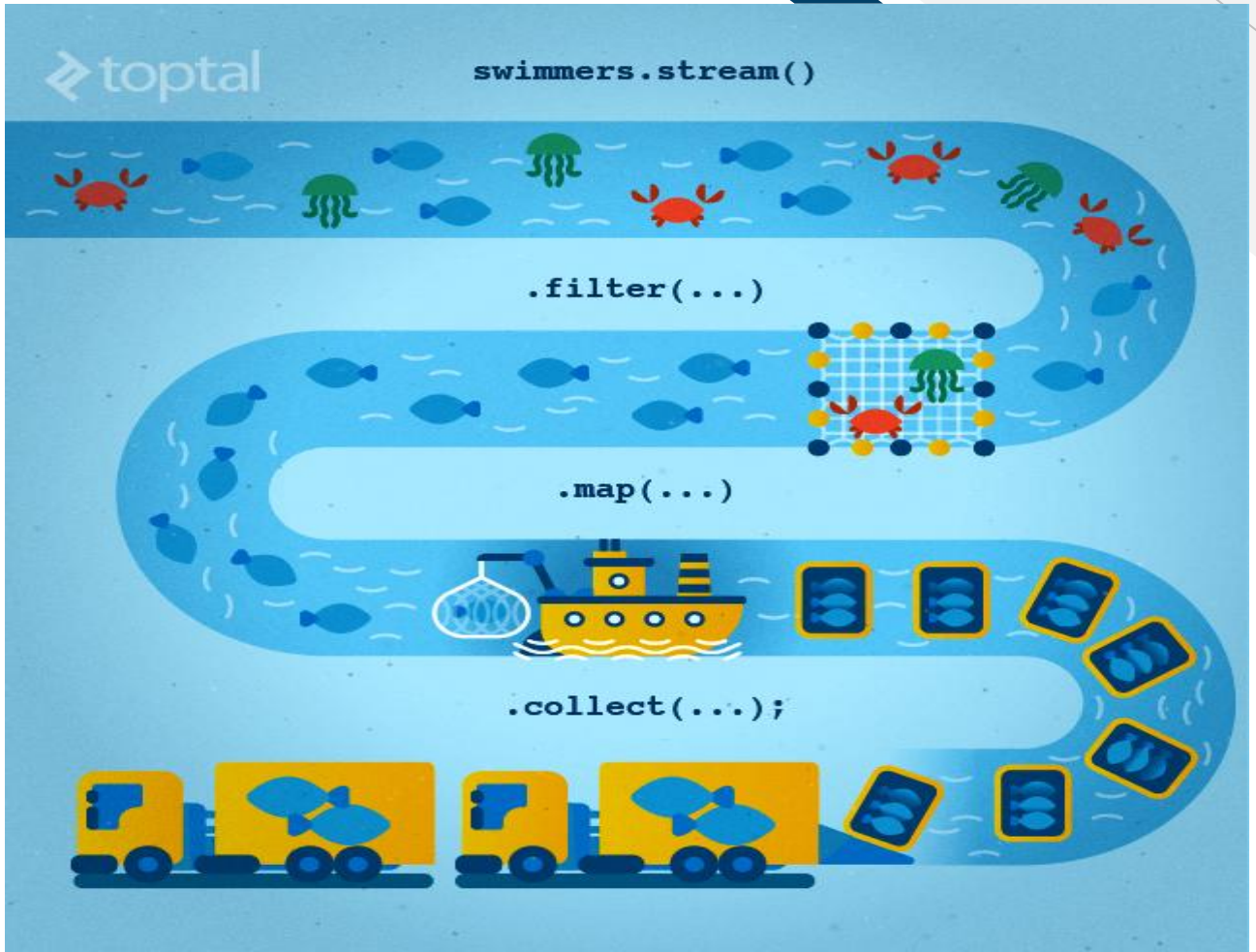
What is stream?

- *"A **sequence of elements** from a **source** that supports **data-processing** operations."*

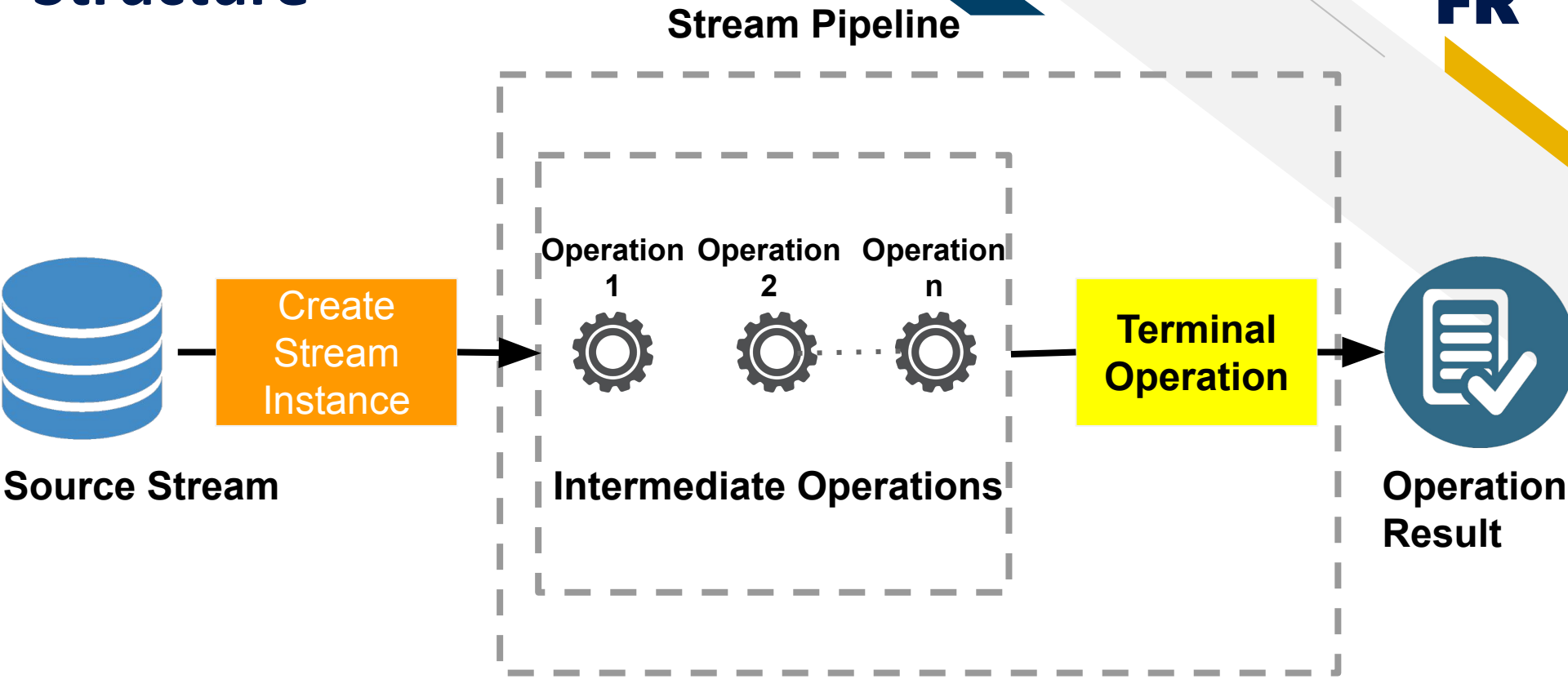
- *Pipelining*
- *Internal Iteration*

What is stream?

FR



Structure

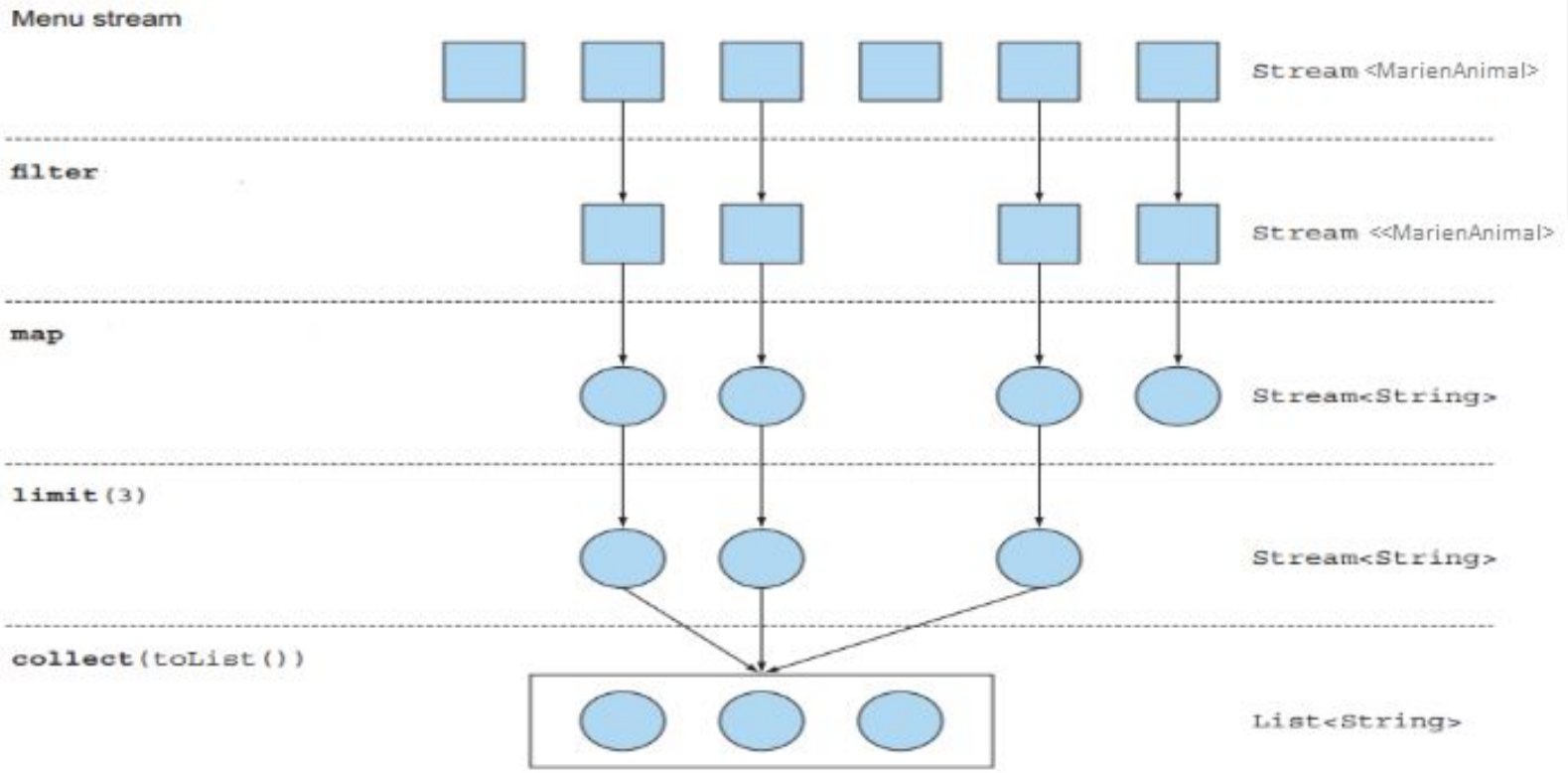

Stream Pipeline
Operation 1
Operation 2
Operation n
Create Stream Instance
Terminal Operation
Source Stream
Intermediate Operations
Operation Result

Example

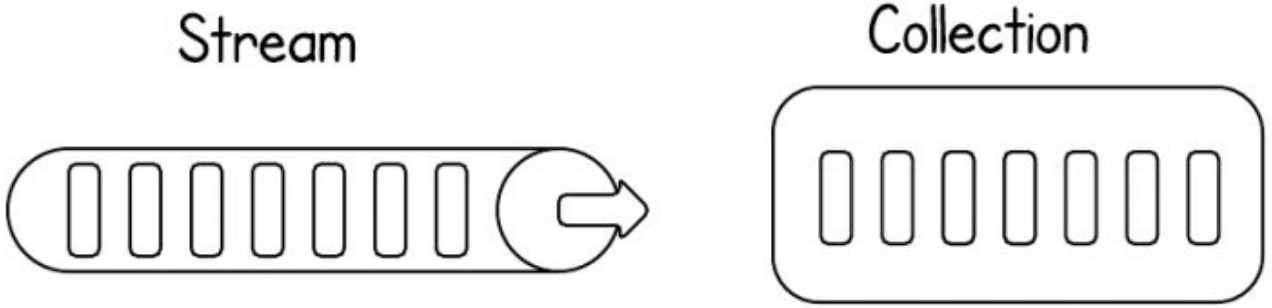
```
List<String> fishNames = marineAnimals
                        .stream ()
                        .filter(seaAnimal -> seaAnimal.getType() == "Fish")
                        .map(MarienAnimal::getName())
                        .limit(3)
                        .collect(toList());
System.out.prinln(fishNames)
```

Output: [clownfish, shark, dolphin]

Example


Menu stream
Stream <MarienAnimal>
filter
Stream <<MarienAnimal>
map
Stream<String>
limit(3)
Stream<String>
collect(toList())
List<String>

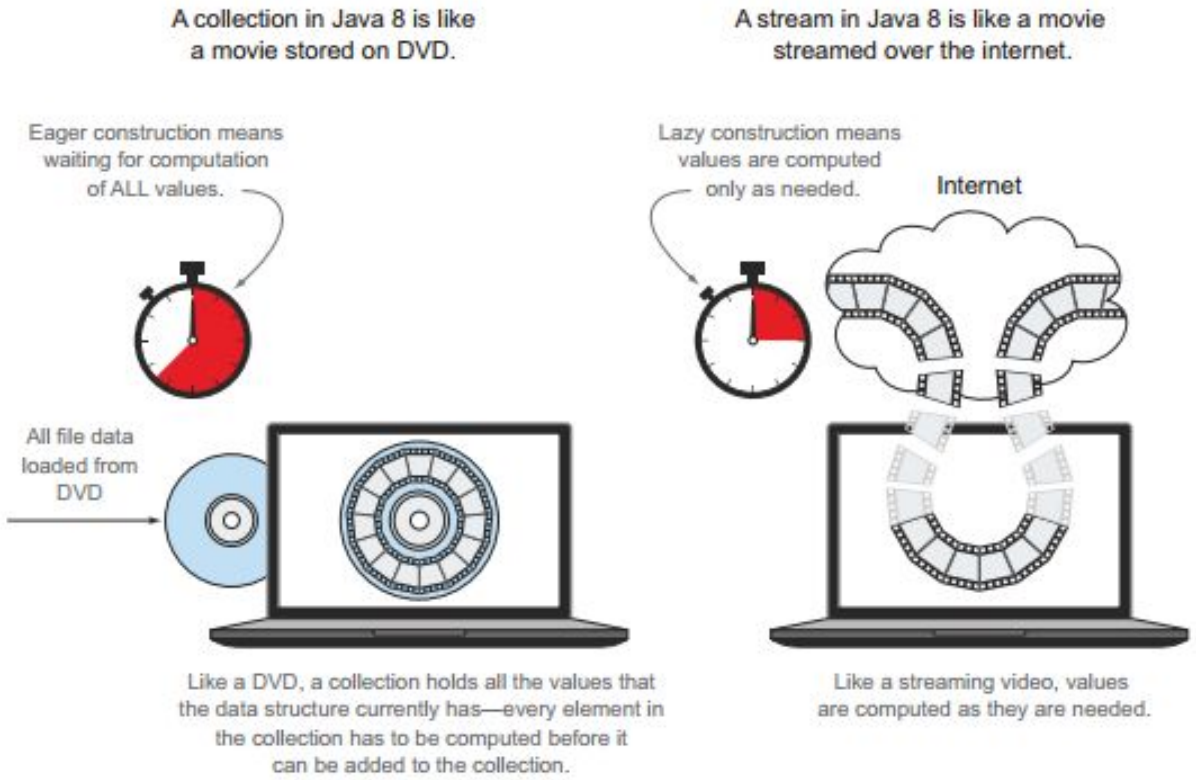
Comparison



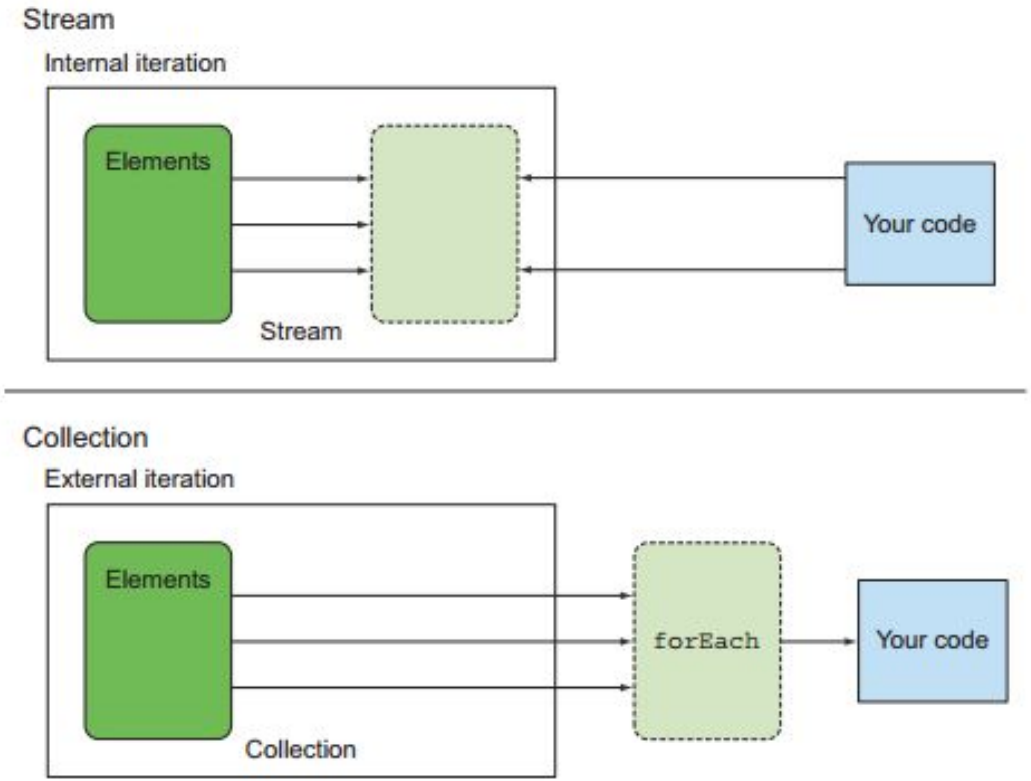
- Computed on demand
- Lazily constructed
- Internal iteration

- Computed before becoming
- Eagerly constructed
- External iteration

Comparison


A collection in Java 8 is like a movie stored on DVD.
Eager construction means waiting for computation of ALL values.
All file data loaded from DVD
Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.
A stream in Java 8 is like a movie streamed over the internet.
Lazy construction means values are computed only as needed.
Internet
Like a streaming video, values are computed as they are needed.

Comparison


Stream
Internal iteration
Elements
Stream
Your code
Collection
External iteration
Elements
Collection
forEach
Your code

Example

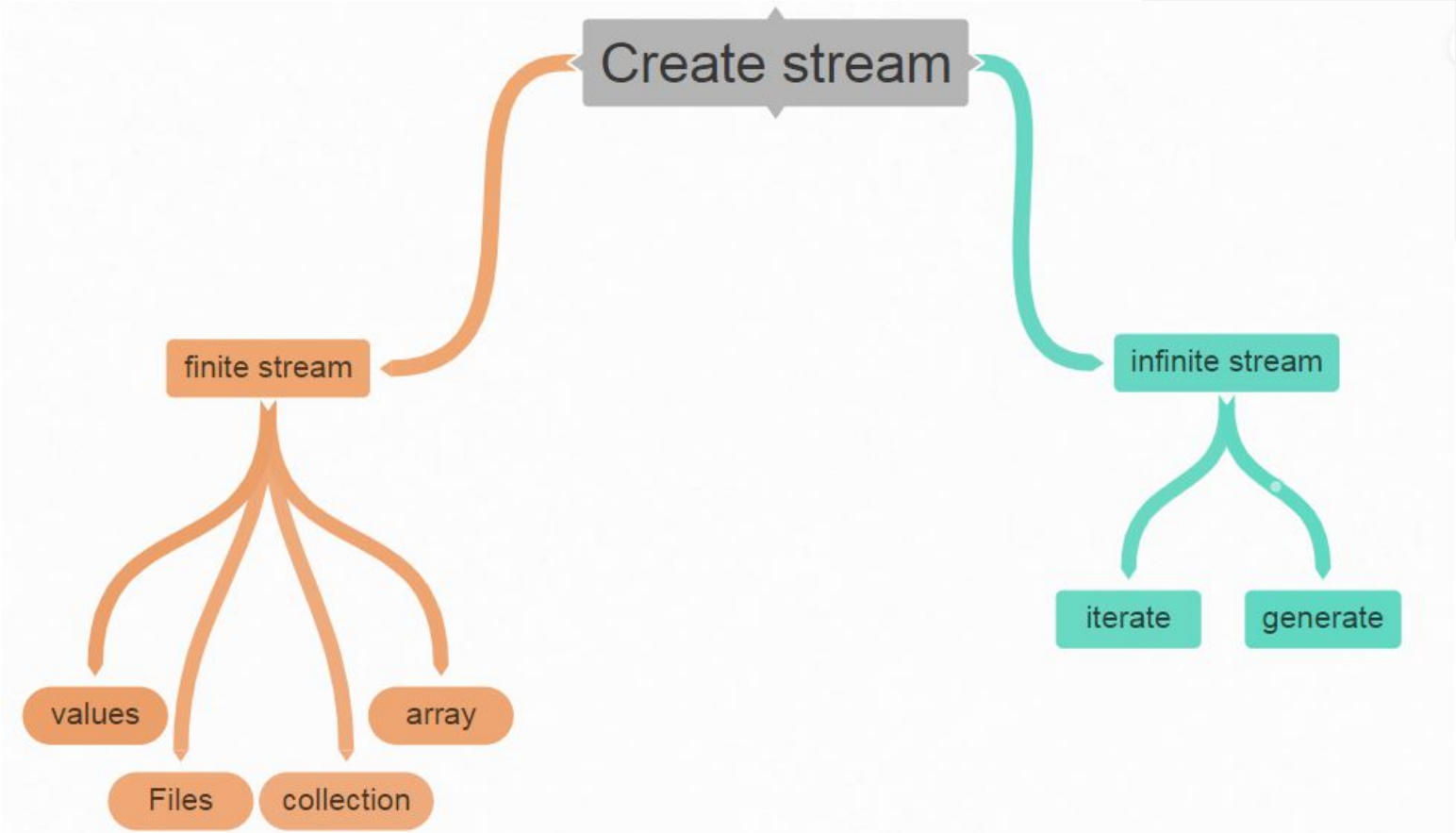
```
List<String> highCaloricDishes = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish dish = iterator.next();
    if(dish.getCalories() > 300) {
        highCaloricDishes.add(d.getName());
    }
}
```

Collection- External iteration

```
List<String> highCaloricDish =
    menu.stream()
        .filter(dish -> dish.getCalories() > 300)
        .collect(toList());
```

Stream - Internal iteration

Working with streams



Working with streams

FR

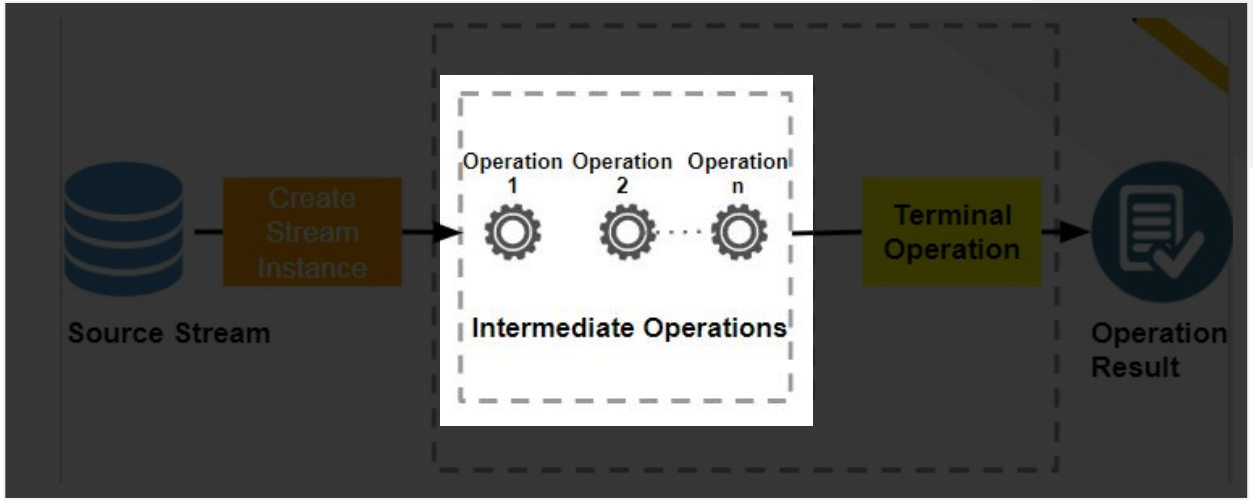
- *You're **dealing** with a stream of **infinite size**, you have to **limit its size***
- ***Can't sort** or **reduce** an **infinite stream***

Stream operations

Intermediate operations

Terminal operations

Intermediate operations



- Can be **connected together** to form a pipeline
- Return another **stream**
- Don’t perform any processing until a terminal operation is invoked on the stream pipeline

Intermediate operations

FR

reduce

limit

distinct

skip

sorted

Stateful

The processing of an element may depend on aspectects of thee other element

Store state to calculate a value

Not easy to handle, specially parallel

Stateless

Each element is processed independently of the others

Don’t store any state

Easy to handle

filter

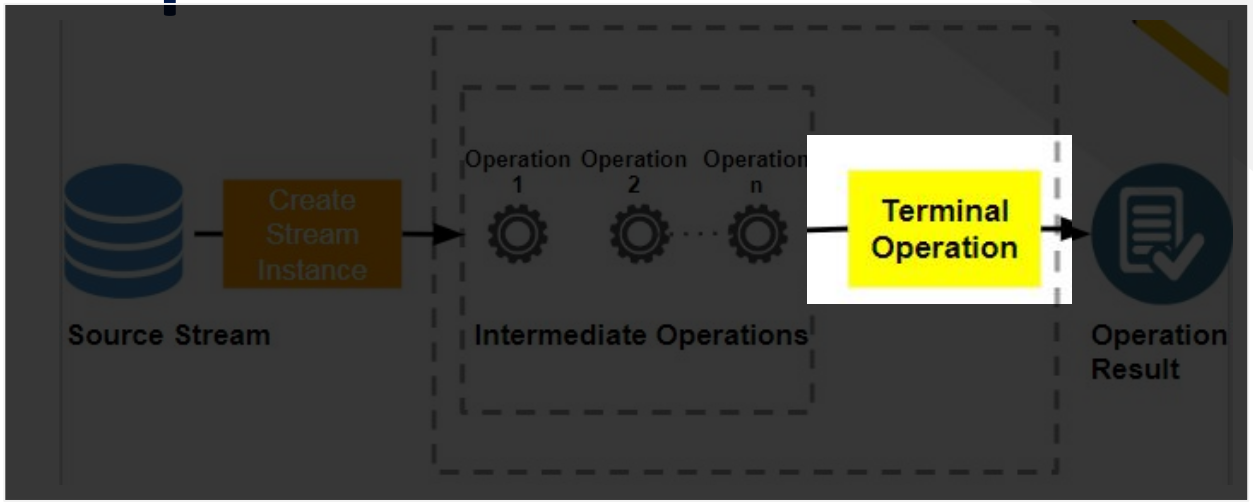
flatMap

dropWhile

takeWhile

map

Terminal operations



- **Executes** the stream pipeline and **produces** a result
- Return a **non-stream value**
- After its execution, the stream **can not** be **revisted**

Terminal operations

anyMatch

noneMatch

allMatch

findAny

findFirst

forEach

collect

reduce

count

anyMatch
noneMatch
allMatch
findAny
findFirst
forEach
collect
reduce
count

Terminal operations

FR

```
Bool isBigFish = marineAnimals
                    .stream ()
                    .anyMatch(seaAnimal -> seaAnimal.getWeight() > 50)
```

Output: true or false

anyMatch

noneMatch

allMatch

findAny

findFirst

forEach

collect

reduce

count

Terminal operations

FR

```
String fishName = marineAnimals
                    .stream ()
                    .filter(seaAnimal -> seaAnimal.getType() == "Fish")
                    .findAny();
```

Output: clownFish

anyMatch
noneMatch
allMatch
findAny
findFirst
forEach
collect
reduce
count

Terminal operations

FR

```
marineAnimals
.stream ()
.filter(seaAnimal -> seaAnimal.getType() == "Fish")
.map(MarienAnimal::getName())
.forEach(System.out::println);
```

Output:
clownFish
shark
dolphin

anyMatch

noneMatch

allMatch

findAny

findFirst

forEach

collect

reduce

count

Terminal operations

FR

```
List<String> fishNames = marineAnimals
                        .stream ()
                        .filter(seaAnimal -> seaAnimal.getType() == "Fish")
                        .map(MarienAnimal::getName())
                        .limit(3)
                        .collect(toList());
```

anyMatch
noneMatch
allMatch
findAny
findFirst
forEach
collect
reduce
count

Terminal operations

FR

```
Int totalWeight = marineAnimals
                .stream ()
                .map(MarienAnimal::getWeight())
                .reduce(0, (total, element) -> total + element);
```

anyMatch

noneMatch

allMatch

findAny

findFirst

forEach

collect

reduce

count

Terminal operations

FR

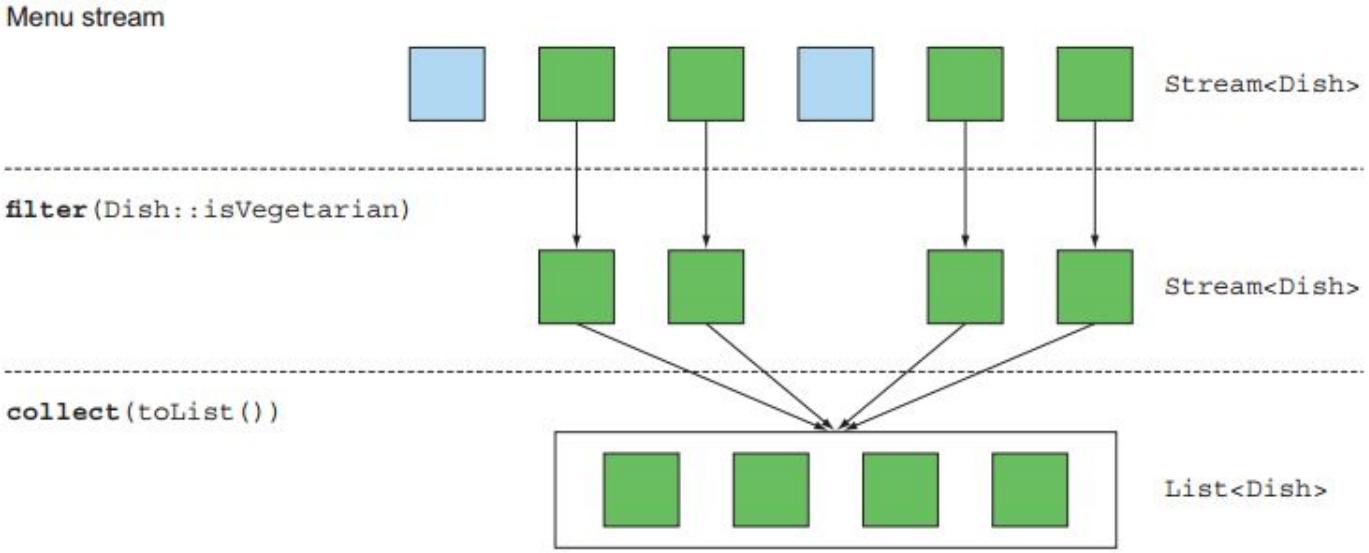
```
Int countFish = marineAnimals
                .stream ()
                .filter(seaAnimal -> seaAnimal.getType() == "Fish")
                .count();
```

Working with Stream

- Filtering
- Slicing a stream
- Mapping
- Reducing
- Numberic streams

Filtering with a predicate

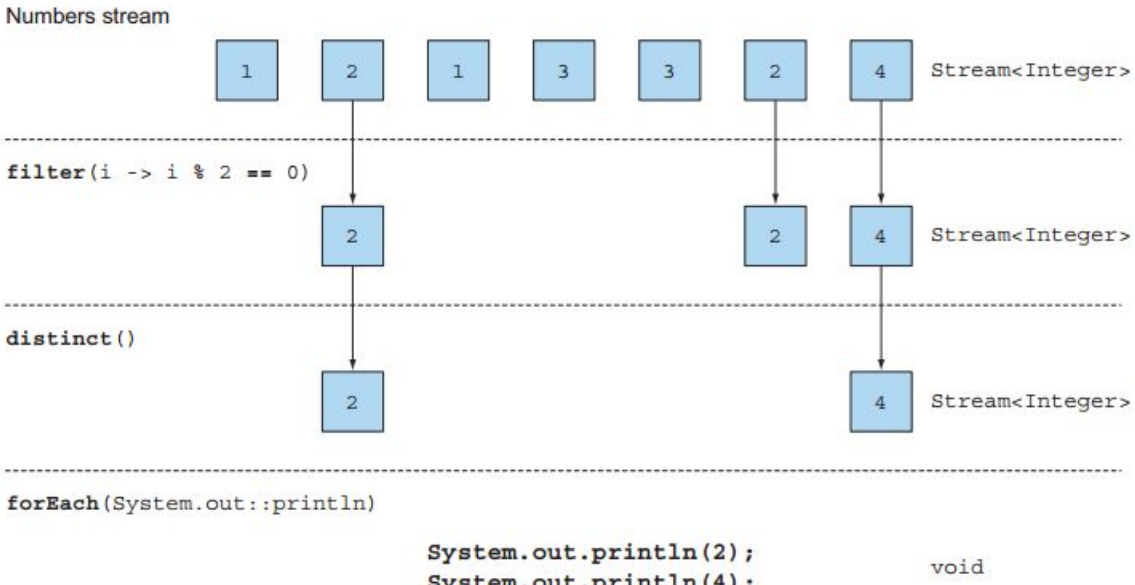
```
List<Dish> vegetarianDishes = menu.stream()
                              .filter(Dish::isVegetarian)
                              .collect(toList());
```



Output: [seasonal fruit, french fries, rice, pizza]

Filtering unique elements

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```



Output: 2 4

Slicing using a predicate

FR

- takeWhile (J9): slice any stream using a predicate. It **stops** once it has found an **element** that **fails to match**
- dropWhile (J9): is the complement of takeWhile. It throws away the **elements** at the **start** where the **predicate is false**

Example

Given list: [seasonal fruit, prawns, rice, chicken, salmon, french fries, pizza, beef, pork]

```
List<Dish> slicedMenu1   = specialMenu.stream()
                                      .takeWhile(dish -> dish.getCalories() < 320)
                                      .collect(toList());
```

takeWhile

Output: [seasonal fruit, prawns]

```
List<Dish> slicedMenu2 = specialMenu.stream()
                                    .dropWhile(dish -> dish.getCalories() < 320)
                                    .collect(toList());
```

dropWhile

Output: [rice, chicken, salmon, french fries, pizza, beef, pork]

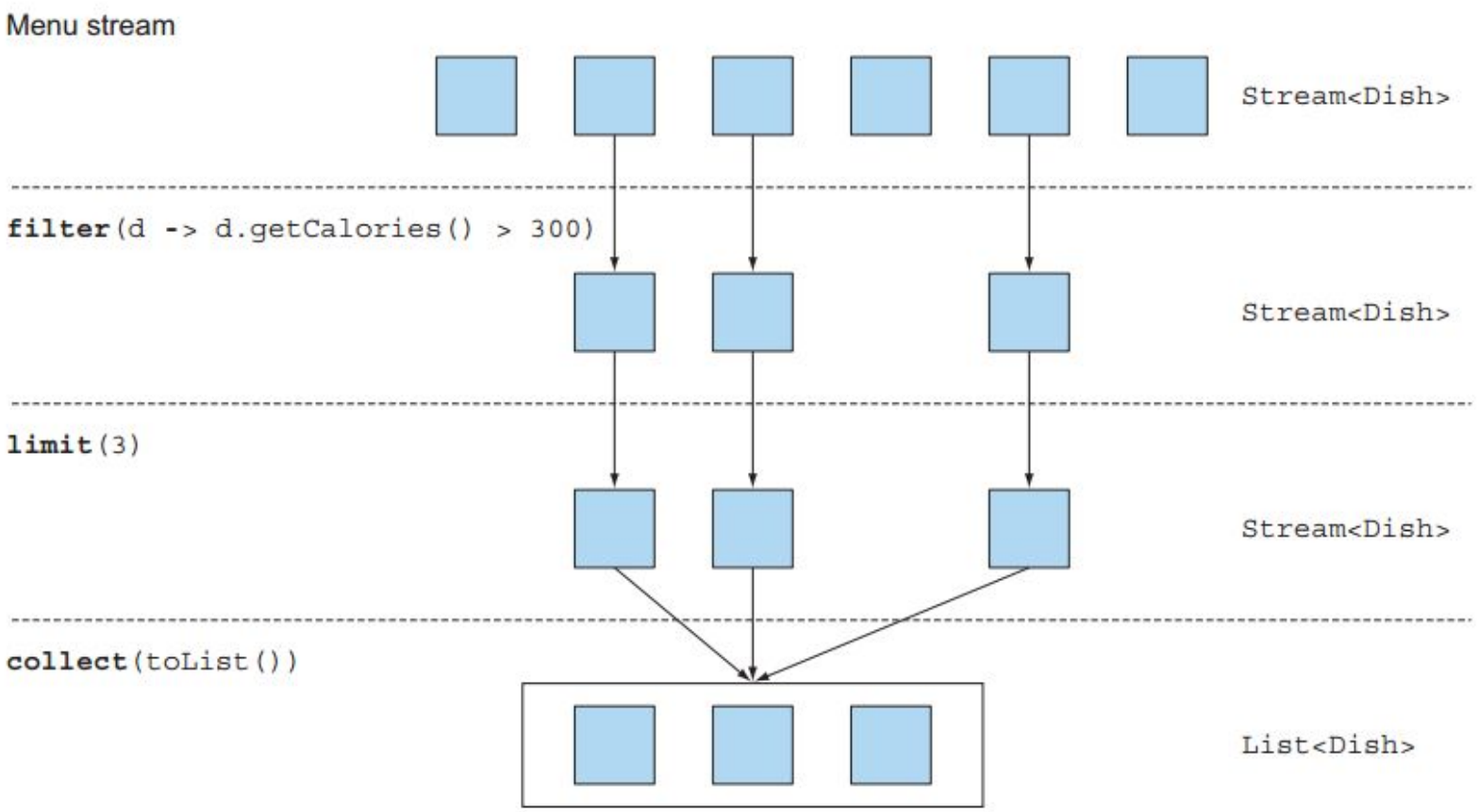
Truncating a stream

- Streams support **the limit(n) method**, which returns **another stream** that's **no longer** than **a given size**.
- **Limit** and **filter** can combine **parallel**

```
List<Dish> dishes = menu.stream()
                        .filter(dish -> dish.getCalories() > 300)
                        .limit(3)
                        .collect(toList());
```

Output: [pork, beef, chicken]

Truncating a stream



Skipping elements

FR

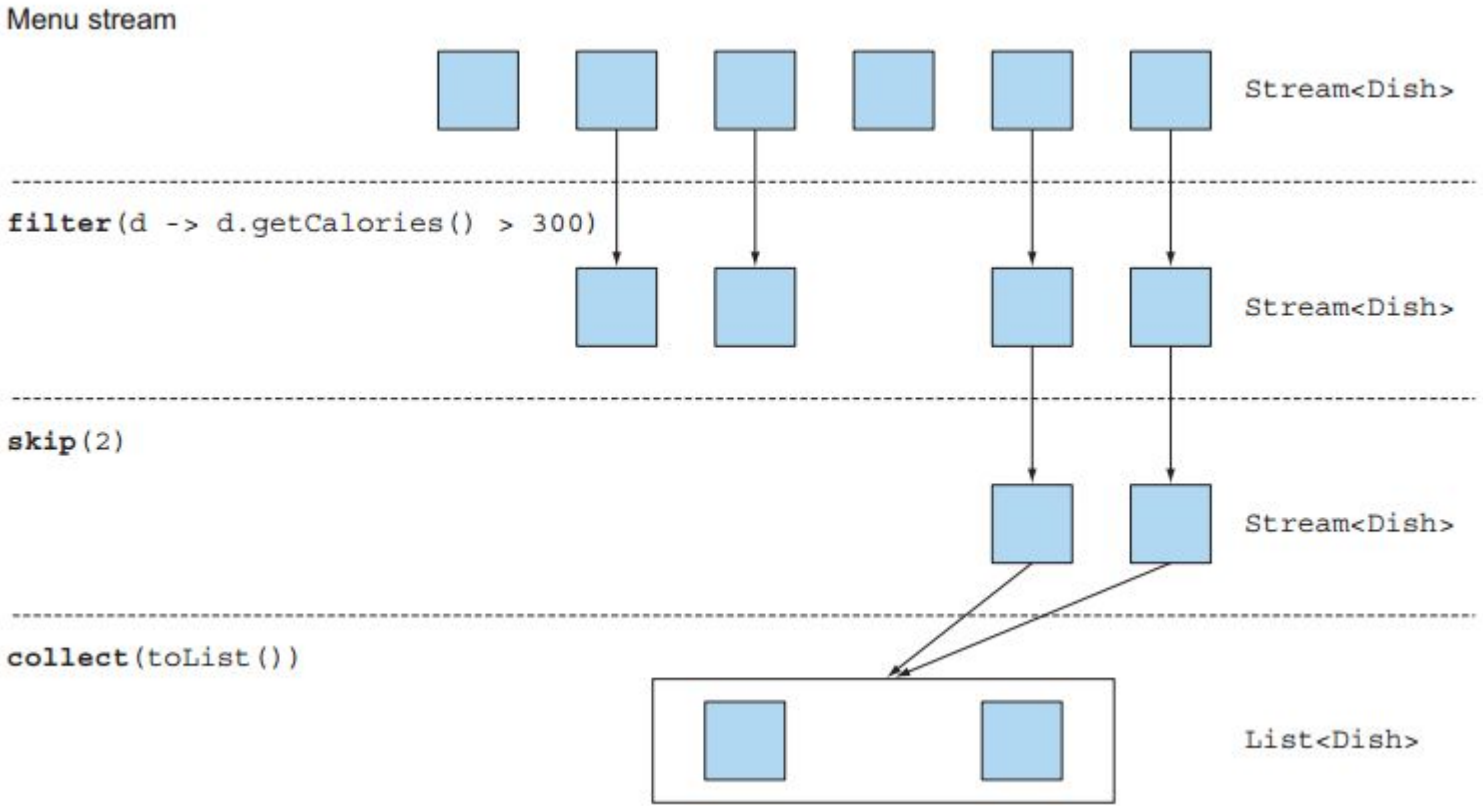
The **skip(n) method** to return a **stream** that **discards** the first n **elements**.

Filtered list without skipping: [pork, beef, chicken, french fries, rice, pizza, salmon]

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```

Output: [chicken, french fries, rice, pizza, salmon]

Skipping elements



Mapping apply a function to each element of a stream

The **map method** takes a function as argument. The function is **applied** to **each element**, **mapping** it **into** a **new element**

```
List<String> names = menu.stream()
                          .map(Dish::getName)
                          .collect(toList());
```

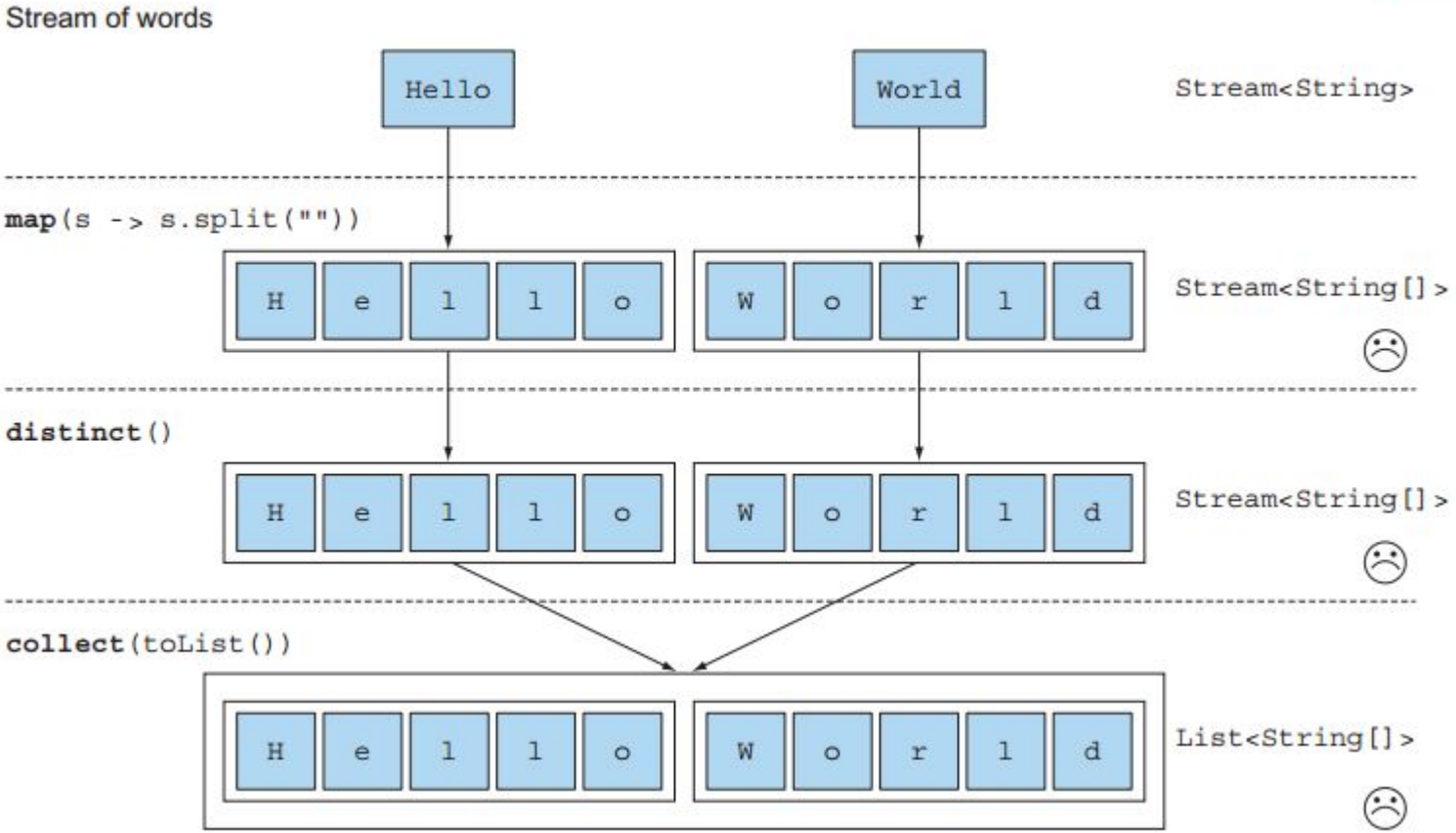
Output: [seasonal fruit, pork, beef, chicken, french fries, rice, pizza, prawns, salmon]

Example

How could you return a list of all the unique characters for a list of words?
For example, given the list of words ["Hello," "World"] you'd like to return the list ["H," "e," "l," "o," "W," "r," "d"]

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

Example

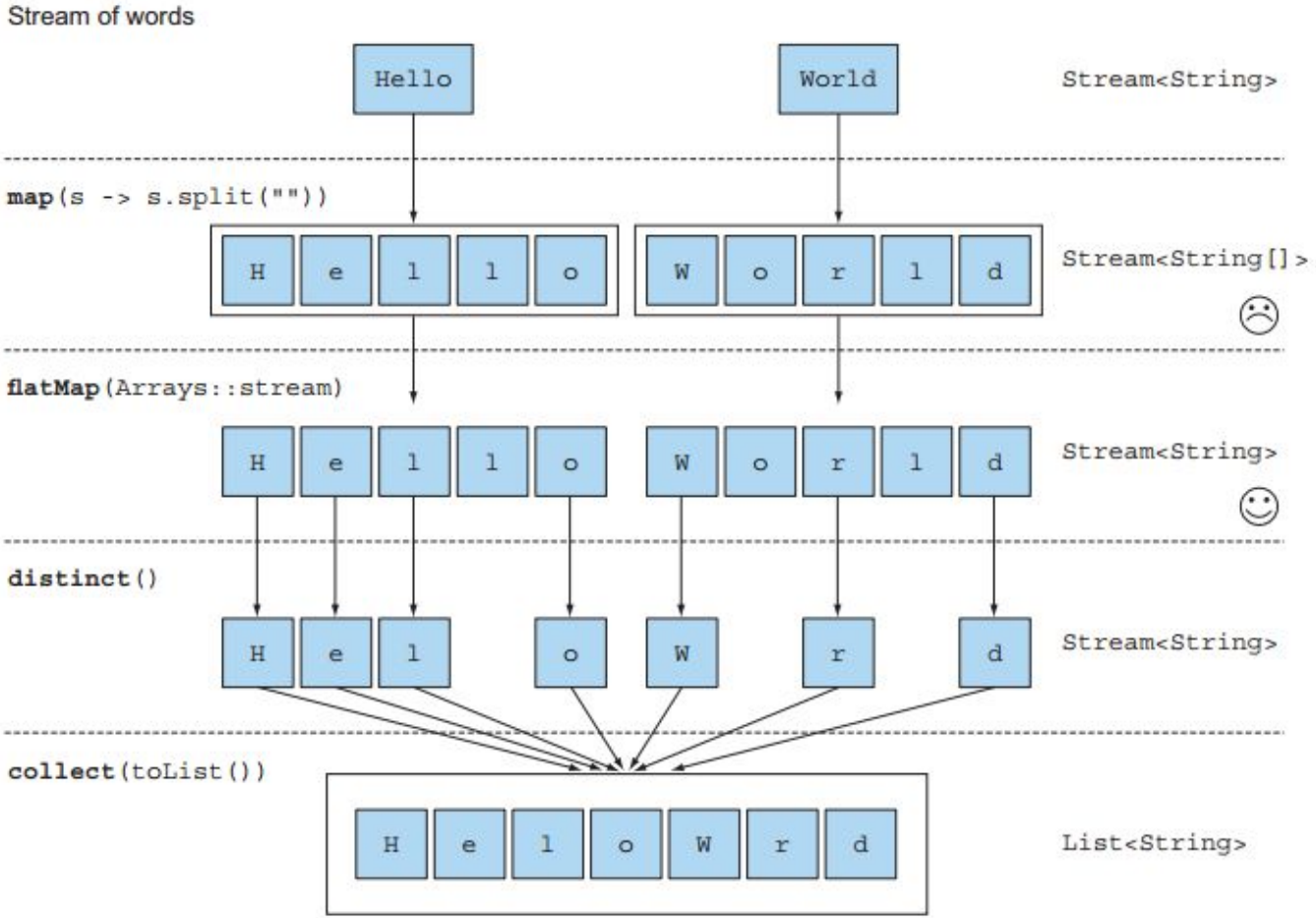

Stream of words
Hello
World
Stream<String>
map(s -> s.split(""))
H e l l o
W o r l d
Stream<String[]>
distinct()
H e l l o
W o r l d
Stream<String[]>
collect(toList())
H e l l o
W o r l d
List<String[]>

Mapping use flatMap

- The **flatMap method** has the **effect** of mapping each array **not** with a **stream** but with the **contents** of that stream.
- All the separate streams that were generated when using map(Arrays::stream) get **amalgamated—flattened** into a **single stream**.

```
words.stream()
     .map(word -> word.split(""))
     .flatMap(Arrays::stream)
     .distinct()
     .collect(toList());
```

Mapping use flatMap



Finding and matching

Method	Description	Return
anyMatch()	check an existing element of the stream match the given predicate	boolean
allMatch()	check no elements in the stream match the given predicate	boolean
noneMatch()	check no elements in the stream match the given predicate	boolean
findAny()	find the any element	an arbitrary element of the current stream
findFirst()	find the first element	the first element of the current stream

Example

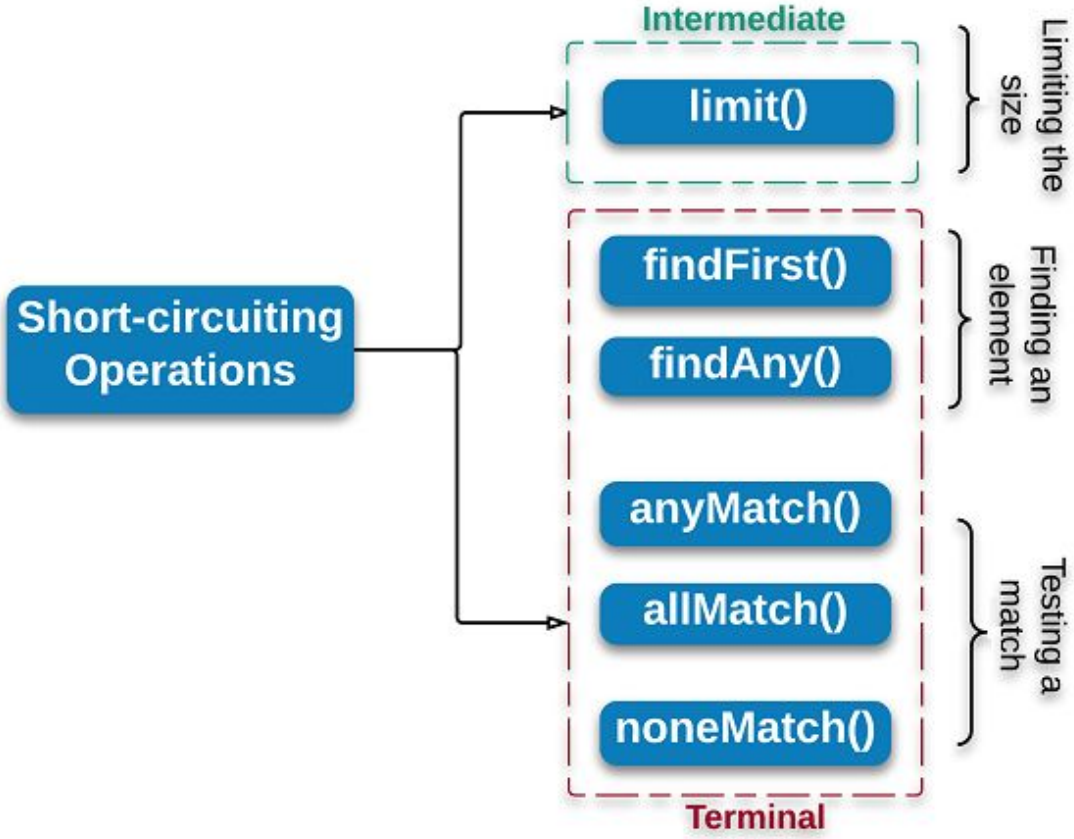
```
menu.stream()
    .anyMatch(dish -> dish.getCalories() < 1000);
```

```
menu.stream()
   .allMatch(dish -> dish.getCalories() < 1000);
```

```
menu.stream()
   .noneMatch(d -> d.getCalories() >= 1000);
```

Short-circuiting evaluation

Don’t need to **process** the **whole stream** to produce a result

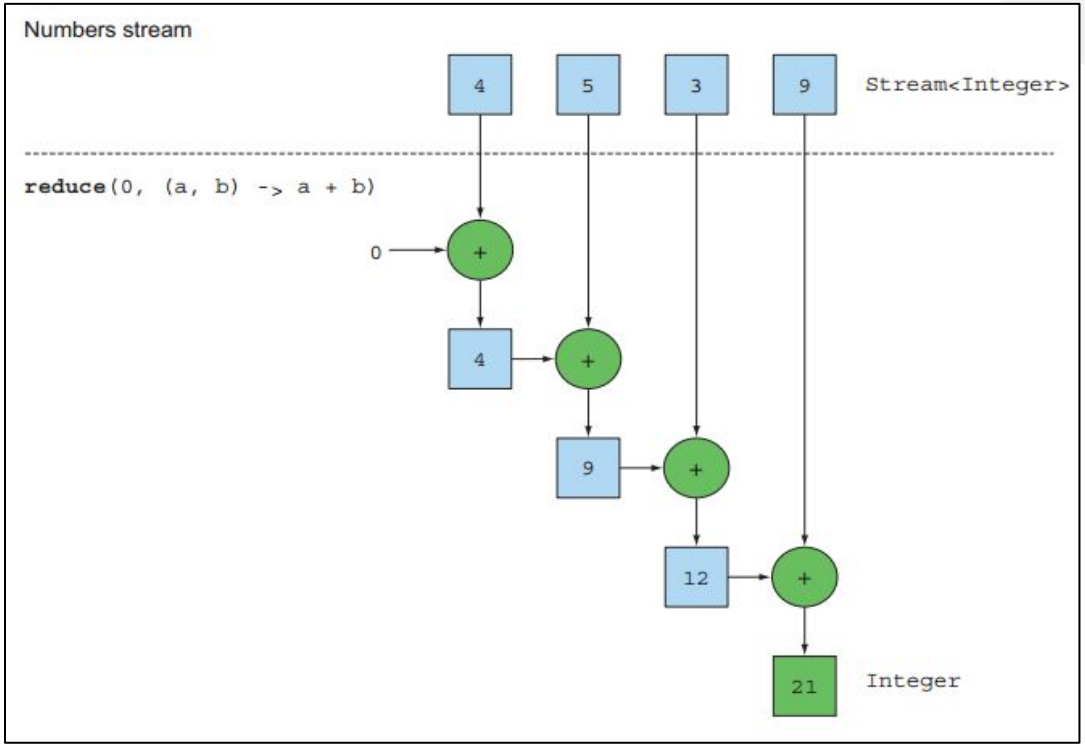


Reducing

- That method allows to **produce one single result** from a sequence of elements
- The benefit of using reduce is that the **iteration is abstracted** using **internal iteration**, which enables the internal implementation to choose to **perform** the reduce operation in **parallel**.

Summing the elements

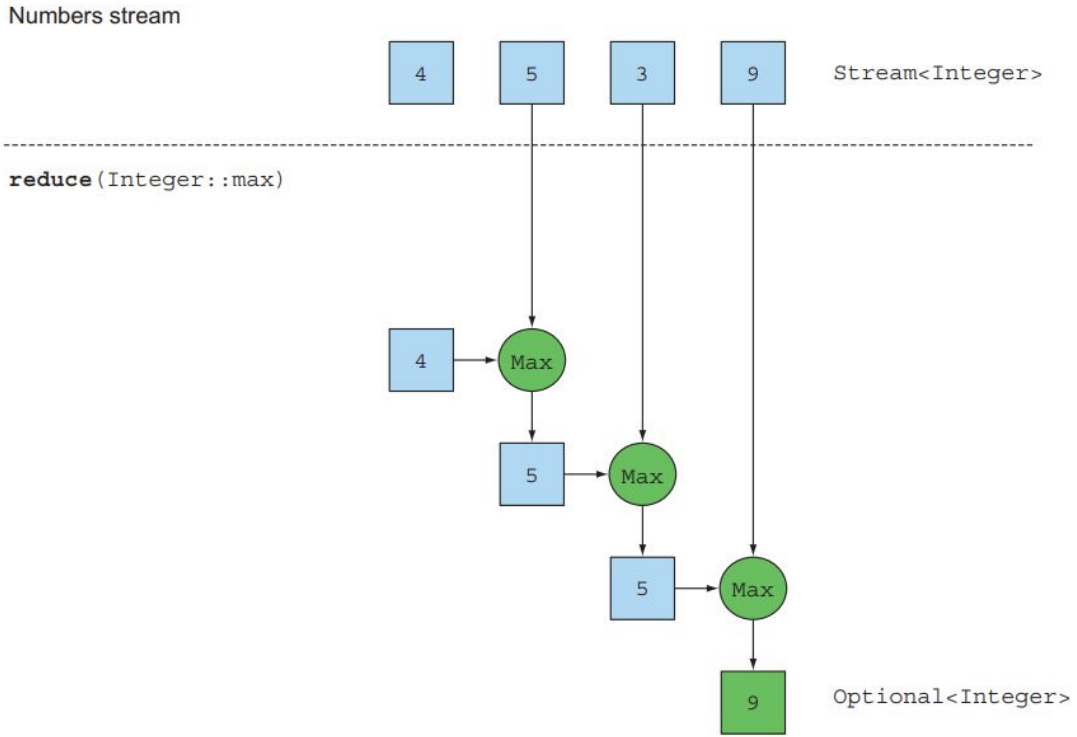
```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```




```
int sum = numbers.stream().reduce(0, Integer::sum);
```

Maximum and minimum

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```



Numeric streams

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

Boxing cost to be unboxed to a primitive type

Primitive stream specializations

IntStream, DoubleStream, and LongStream

- **Respectively specialize** the **elements** of a stream **to** be **int, long, and double**
- **Avoid** hidden **boxing costs**

- Use *mapToInt, mapToDouble, mapToLong* to convert a stream to a specialized version

Primitive stream specializations

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

Numeric ranges

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                 .filter(n -> n % 2 == 0);
System.out.println(evenNumbers.count());
```

Output: [2,4,6,...100]

We can use ***range()*** instead, the result would be 49 even numbers because range is exclusive

```
IntStream evenNumbers = IntStream.range(1, 100)
                                 .filter(n -> n % 2 == 0);
System.out.println(evenNumbers.count());
```

Output: [2,4,6,...98]

Collecting data with stream

Provided Collectors methods

Custom the Collector interface

Overall

Overall

FR

- Collector: a **recipe** for how to **build** a summary of **the elements** in the **stream**
- Offer ***three*** main functionalities:
  - Reducing and summarizing stream elements to a single value
  - Grouping elements
  - Partitioning elements

Reducing and summarizing

To combine all the items in the stream into a single result

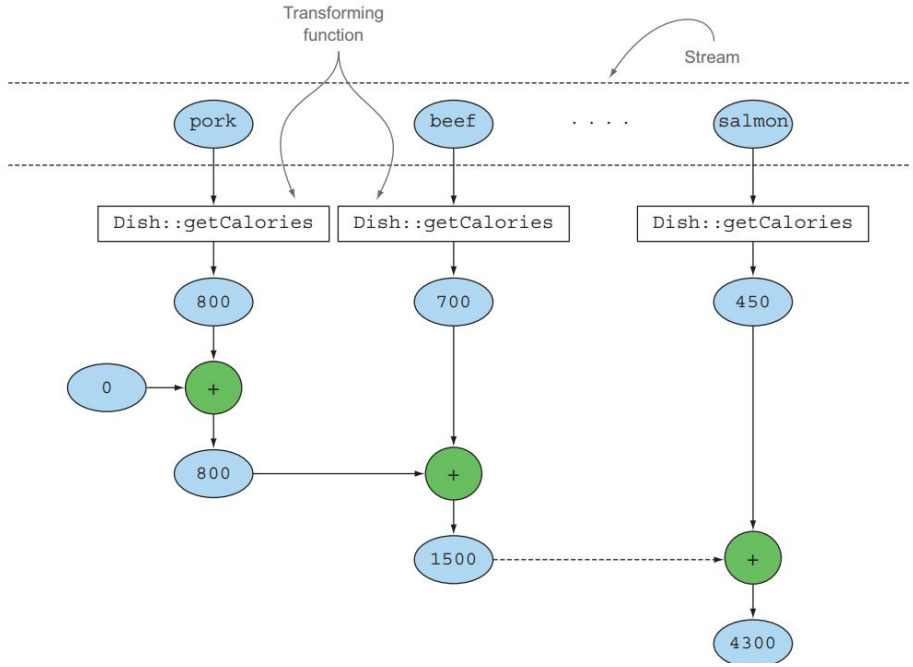
```
Optional<Dish> mostCalorieDish = menu.stream()
                                     .collect(Collectors.maxBy(comparing(Dish::getCalories)));
```

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

```
double avgCalories =
        menu.stream().collect(averagingInt(Dish::getCalories));
```

Example

```
IntSummaryStatistics menuStatistics =
        menu.stream().collect(summarizingInt(Dish::getCalories));
```




```
IntSummaryStatistics{count=9, sum=4300, min=120,
                     average=477.777778, max=800}
```

Provided Collectors Methods

FR

```
IntSummaryStatistics statistics
                    = marineAnimals
                    .streams()
                    .collect(Collectors.summarizingInt(MarienAnimal::getWeight()))
```

<u>Output:</u> count=10, sum= 3200, min = 0.4, average = 320, max = 1200

Provided Collectors Methods

```
Map<MarienAnimal.Type, List<MarienAnimal>> marineAnimalByType =
        marineAnimals.stream()
        .collect(Collectors.groupingBy(MarienAnimal::getType));
```

Output:
{Fish = [clownFish, shark, dolphin], Mollusca = [squid, octopus, jellyfish], Others = [starfish, walrus, seal, turtle]}

Provided Collectors Methods

FR

```
Map<MarienAnimal.Type, List<MarienAnimal>> marineAnimalByType =
        marineAnimals.stream()
        .collect(Collectors.partitioningBy(MarienAnimal::isFish));
```

Output:
{true = [clownFish, shark, dolphin], false= [squid, octopus, jellyfish, starfish, walrus, seal, turtle]}

Provided Collectors Methods

```
IntSummaryStatistics statistics
                     = marineAnimals
                     .streams()
                     .collect(Collectors.summarizingInt(MarienAnimal::getWeight()))
```

Output: count=10, sum= 3200, min = 0.4, average = 320, max = 1200

Joining Strings

FR

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

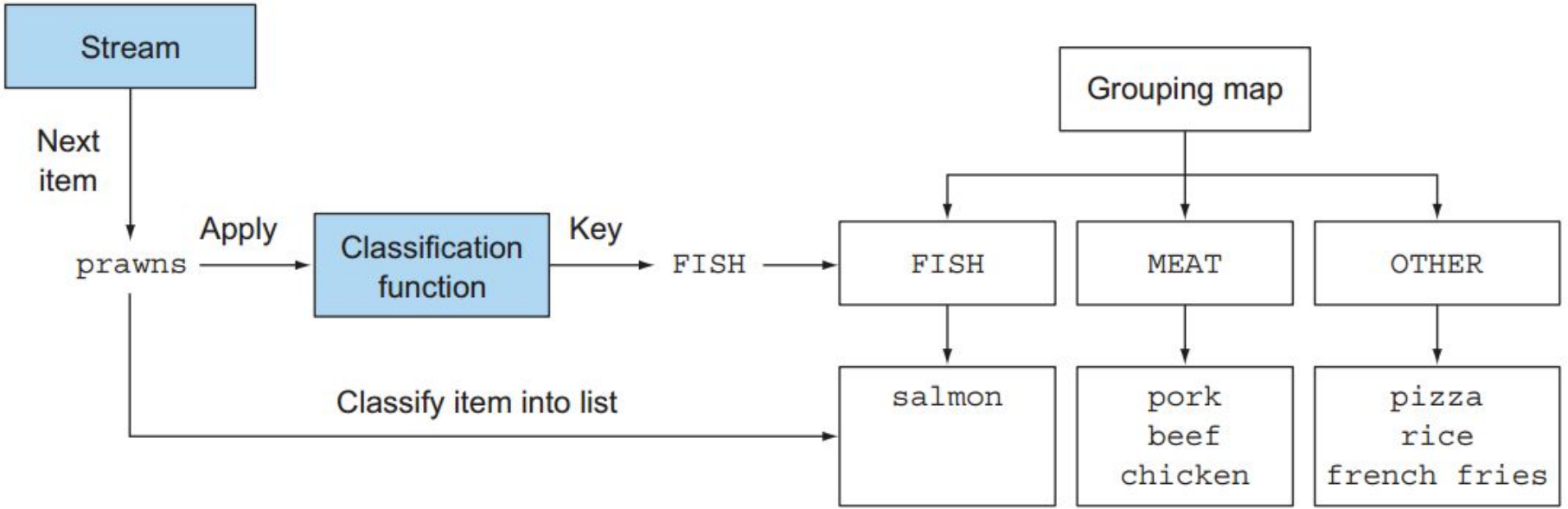
```
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

Grouping

```
Map<Dish.Type, List<Dish>> dishesByType =
                        menu.stream().collect(groupingBy(Dish::getType));
```

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

Grouping


Stream
Next item
prawns
Apply
Classification function
Key
FISH
Classify item into list
Grouping map
FISH
MEAT
OTHER
salmon
pork
beef
chicken
pizza
rice
french fries

Partitioning

```
Map<Boolean, List<Dish>> partitionedMenu =
                menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

```
{false=[pork, beef, chicken, prawns, salmon],
 true=[french fries, rice, season fruit, pizza]}
```

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

Collector Interface

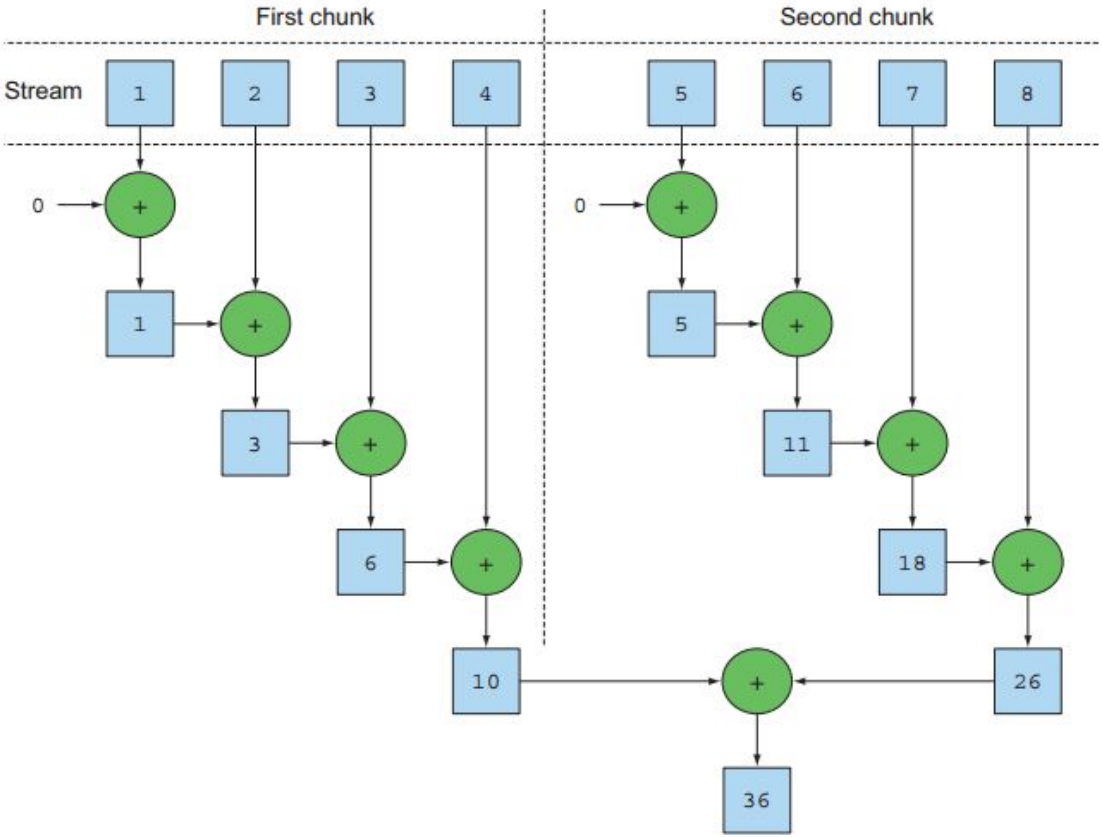
```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

- **T** is the generic type of the items in the stream to be collected.
- **A** is the type of the accumulator, the object on which the partial result will be accumulated during the collection process.
- **R** is the type of the object (typically, but not always, the collection) resulting from the collect operation

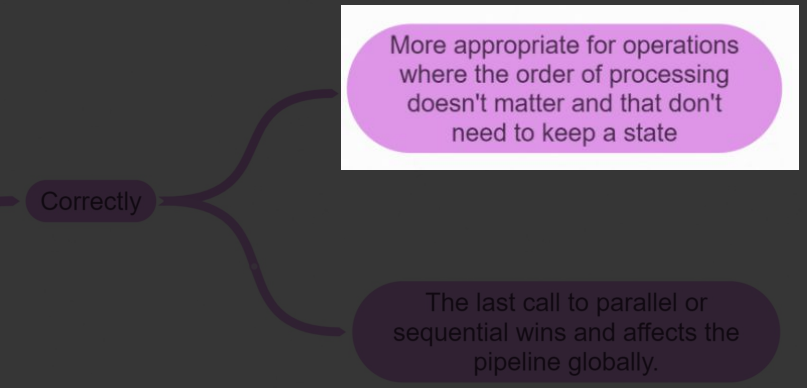
The methods declared by Collector interface

- **supplier**(): returns a Supplier of an empty accumulator
- **accumulator**(): returns the function that performs the reduction operation
- **finisher**(): returns a function that's invoked at the end of the accumulation process to transform the accumulator object into the final result of the whole collection operation
- **combiner**(): returns a function used by the reduction operation
- **characteristics**(): returns an immutable set of Characteristics, defining the behavior of the collector

Parallel data processing and performance


First chunk
Second chunk
Stream
1
2
3
4
5
6
7
8
0
+
1
3
6
10
0
5
11
18
26
36

FR


More appropriate for operations where the order of processing doesn't matter and that don't need to keep a state
Correctly
The last call to parallel or sequential wins and affects the pipeline globally.

FR

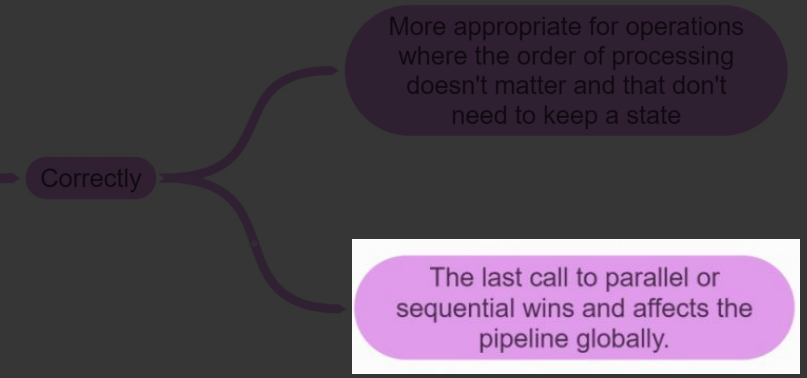
Correctly

More appropriate for operations where the order of processing doesn't matter and that don't need to keep a state

The last call to parallel or sequential wins and affects the pipeline globally.

```
Stream.of("a","b","c","d","e")
.parallel()
.forEach(System.out::print);
```

FR


More appropriate for operations where the order of processing doesn't matter and that don't need to keep a state
Correctly
The last call to parallel or sequential wins and affects the pipeline globally.

Sequential stream is better for small set of data

Avoid stateful(sorted) and order-based (findFirst) operation

Reducing of collection better reduce method

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

When in doubt, check performance with an appropriate with BenchMark

FR

Sequential stream is better for small set of data

Avoid stateful(sorted) and order-based (findFirst) operation

When in doubt, check performance with an appropriate with BenchMark

Reducing of collection better reduce method

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

```
double start = System.nanoTime();
long c = IntStream.rangeClosed(0, 1_000_000_000)
            .parallel()
            .filter(i -> i % 2 == 0)
            .count();
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Got " + c + " in " + duration + " milliseconds");
```

```
Got 500000001 in 738.678448 milliseconds
```

```
Got 500000001 in 1275.271882 milliseconds
```

FR

Sequential stream is better for small set of data

Avoid stateful(sorted) and order-based (findFirst) operation

When in doubt, check performance with an appropriate with BenchMark

Reducing of collection better reduce method

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

```
Stream.of("a","b","c","d","e")
.forEach(System.out::print);
```

abcde

```
Stream.of("a","b","c","d","e")
.parallel()
.forEach(System.out::print);
```

cabde

cedab

caedb

FR

Sequential stream is better for small set of data

Avoid stateful(sorted) and order-based (findFirst) operation

Reducing of collection better reduce method

Reducing of collection better reduce method

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
             .reduce( 4, (sum, n) -> sum + n );
```

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
             .parallel()
             .reduce( 4, (sum, n) -> sum + n );
```

Use reducing() method of class Collectors

Reducing and reduce

FR

Reducing method in Collector

- **Mutable** container to accumulate result
- Useful for **expressing** but **crucially** in **parallel-friendly**

Reduce method

- **Immutable** reduction

```
int totalCalories = menu.stream().collect(reducing(0,
                                  Dish::getCalories,
                                  Integer::sum));
```

Initial value

Aggregating function

Transformation function

FR

Sequential stream is better for small set of data

Avoid stateful(sorted) and order-based (findFirst) operation

Reducing of collection better reduce method

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

When in doubt, check performance with an appropriate with BenchMark

Stream

```
long start = System.nanoTime();
int sum = Stream.of(1,2,3,4,5,6,7,8,9,10).reduce(0, Integer::sum);
long duration = (System.nanoTime() - start) / 1000000;
System.out.println("Sum =" + sum + " found in " + duration + " milliseconds");
```

Output: Sum =55 found in 47 milliseconds

IntStream

```
long start1 = System.nanoTime();
IntStream intStream = Stream.of(1,2,3,4,5,6,7,8,9,10).mapToInt(Integer::intValue)
int sum1 = intStream.sum();
long duration1 = (System.nanoTime() - start1) / 1000000;
System.out.println("Sum =" + sum1 + " found in " + duration1 + " milliseconds");
```

Output: Sum =55 found in 3 milliseconds

FR

Sequential stream is better for small set of data

Avoid stateful(sorted) a
based (findFirst) ope

Reducing of collection be
method

```
new Random().ints(100).boxed()
        .parallel()
        .map(this::slowOperation)
        .collect(Collectors.toList())
        // Start new stream here
        .stream()
        .map(Function.identity())//some fast operation, but must be in single thread
        .collect(Collectors.toSet());
```

Boxing cost

The last call to parallel or sequential wins and affects the pipeline globally.

When in doubt, check performance with an appropriate with BenchMark

FR

Sequential stream is better for small set of data

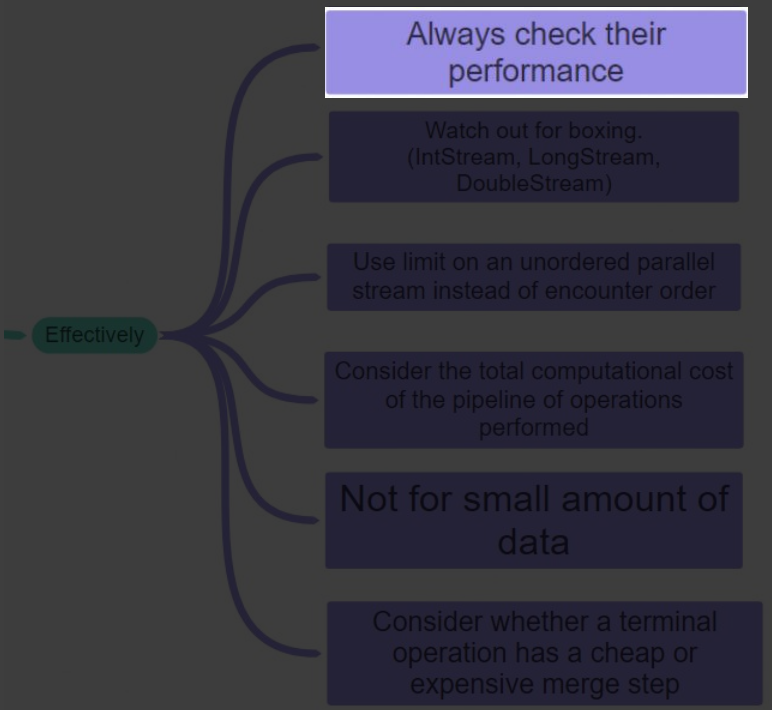
Avoid stateful(sorted) and order-based (findFirst) operation

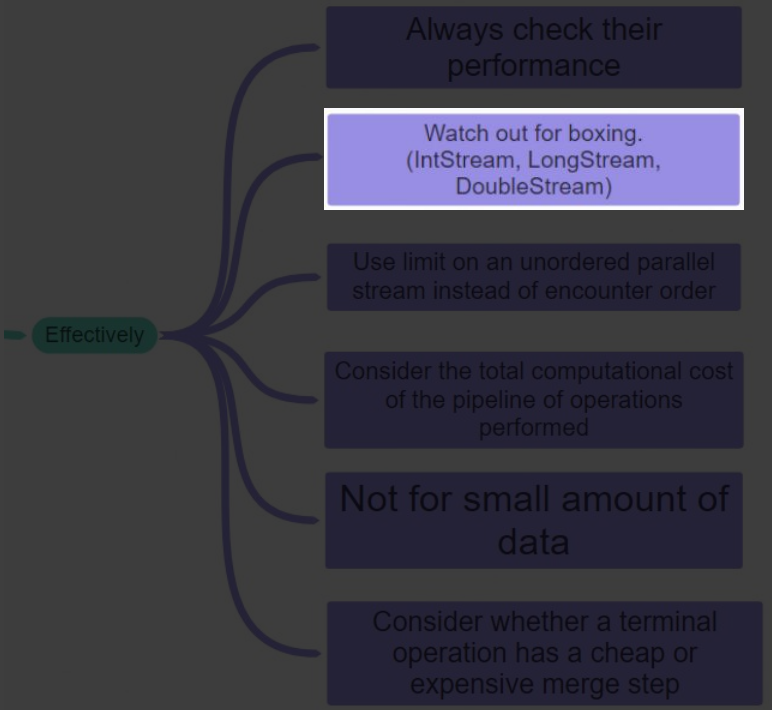
When in doubt, check performance with an appropriate with BenchMark

Reducing of collection better reduce method

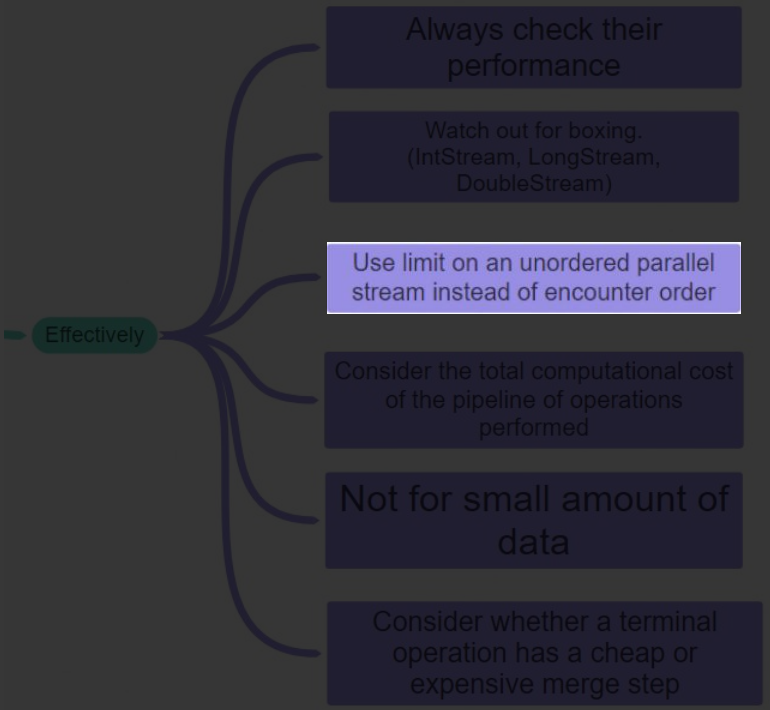
Boxing cost

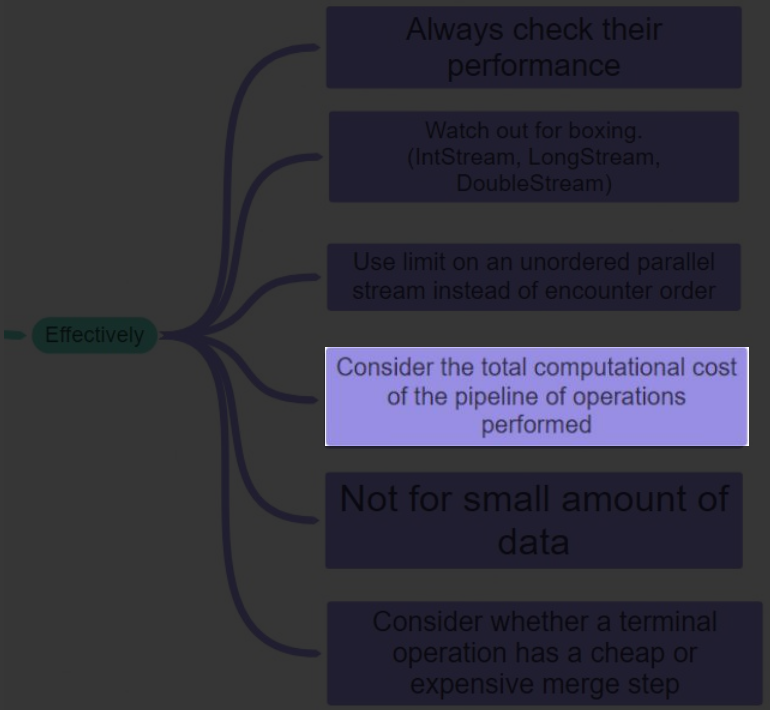
When in doubt, check performance with an appropriate with BenchMark

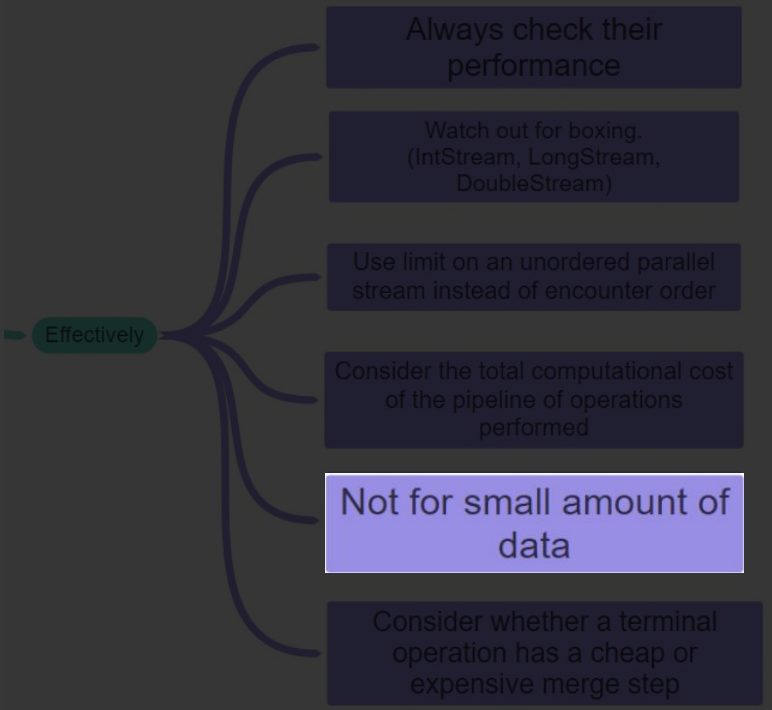

Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step


Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step

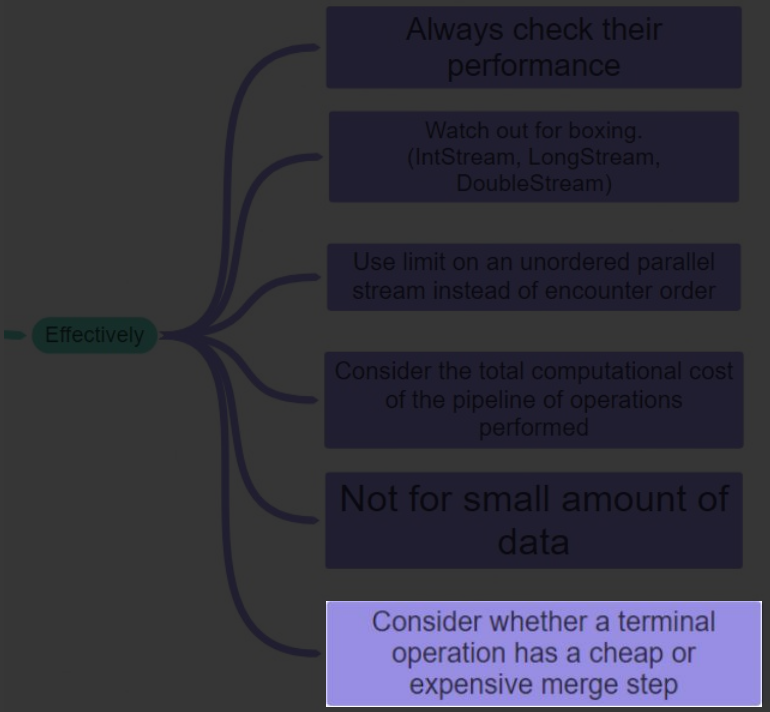
FR


Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step


Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step

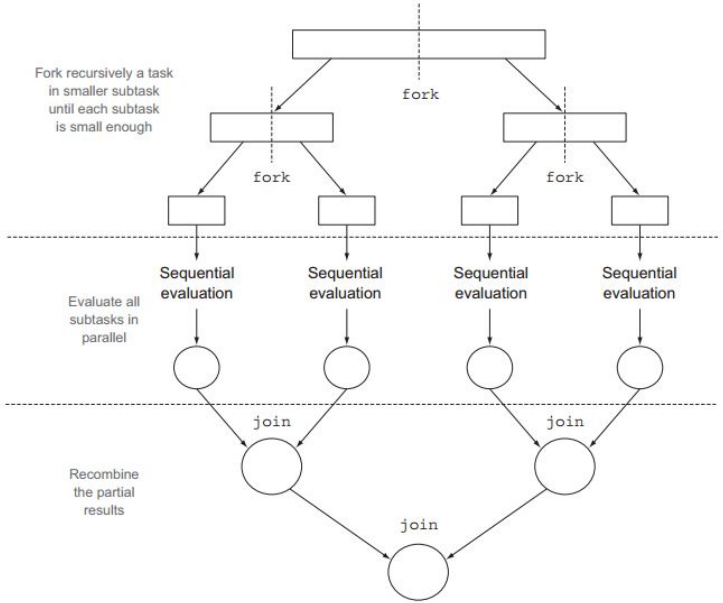

Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step

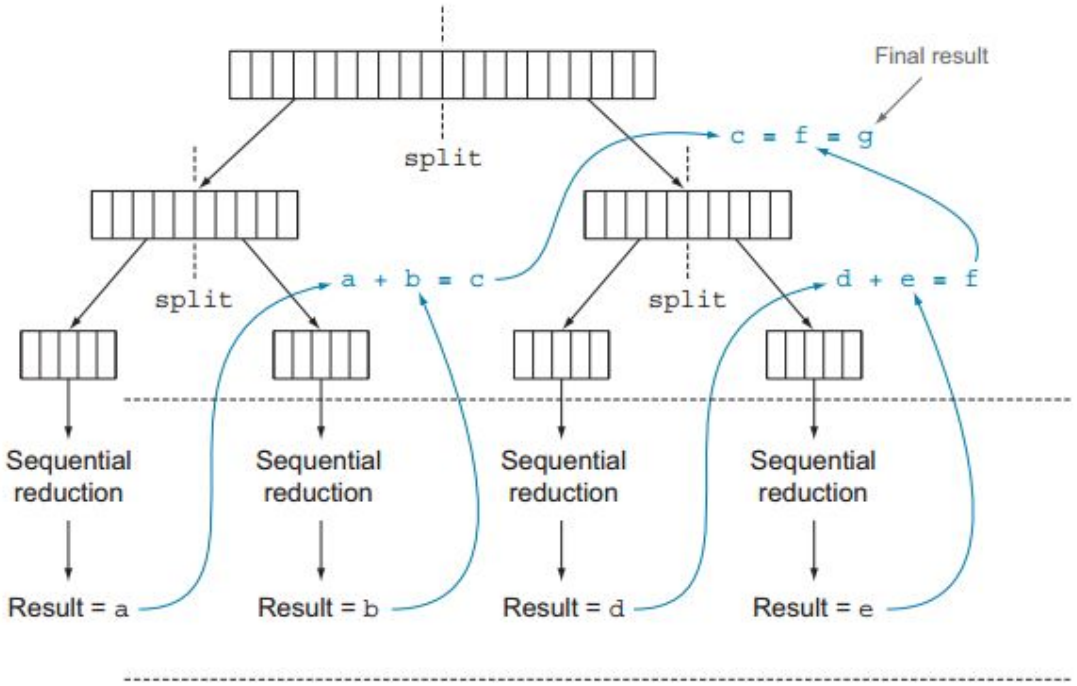
FR


Effectively
Always check their performance
Watch out for boxing. (IntStream, LongStream, DoubleStream)
Use limit on an unordered parallel stream instead of encounter order
Consider the total computational cost of the pipeline of operations performed
Not for small amount of data
Consider whether a terminal operation has a cheap or expensive merge step

Fork/Join FrameWork

The fork/join framework was designed to recursively split a parallelizable task into smaller tasks and then combine the results of each subtask to produce the overall result.




Final result
split
c = f = g
split
a + b = c
split
d + e = f
Sequential reduction
Sequential reduction
Sequential reduction
Sequential reduction
Result = a
Result = b
Result = d
Result = e

Best practices for using the fork/join framework

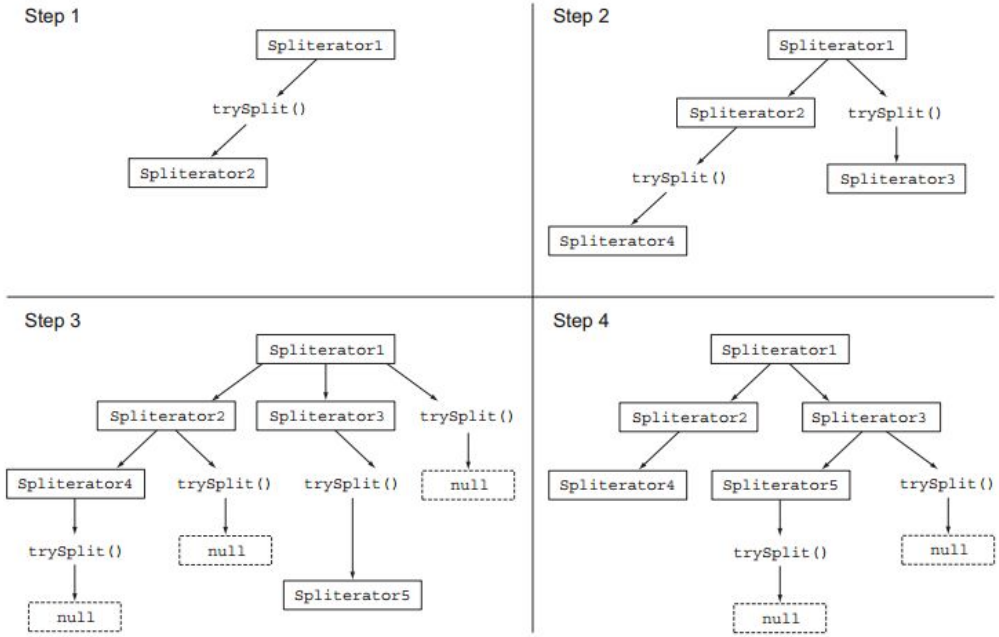
- Invoking the join method on a task blocks the caller until the result produced by that task is ready.
- Should always call the methods compute or fork directly; only sequential code should use invoke to begin parallel computation.
- Calling the fork method on a subtask is the way to schedule it on the ForkJoinPool. Doing this allows you to reuse the same thread for one of the two subtasks and avoid the overhead caused by the unnecessary allocation of a further task on the pool.
- Debugging a parallel computation using the fork/join framework can be tricky.
- It’s always important to run the program multiple times before to measure its performance.

Spliterator

Splitable + Iterator

traverse the elements of a source, but they’re also designed to do this in parallel.

- tryAdvance: sequentially consume the elements of the Spliterator one by one, returning true if there are still other elements to be traverse.
- trySplit: partition off some of its elements to a second Spliterator (the one returned by the method), allowing the two to be processed in parallel.
- estimateSize: number of the elements remaining to be traverse


Step 1
Spliterator1
trySplit()
Spliterator2
Step 2
Spliterator1
Spliterator2
trySplit()
trySplit()
Spliterator3
Spliterator4
Step 3
Spliterator1
Spliterator2
Spliterator3
trySplit()
Spliterator4
trySplit()
trySplit()
null
trySplit()
null
null
Spliterator5
Step 4
Spliterator1
Spliterator2
Spliterator3
Spliterator4
Spliterator5
trySplit()
trySplit()
null
null

Characteristic	Meaning
ORDERED	Elements have a defined order (for example, a `List`), so the `Spliterator` enforces this order when traversing and partitioning them.
DISTINCT	For each pair of traversed elements `x` and `y`, `x.equals(y)` returns `false`.
SORTED	The traversed elements follow a predefined sort order.
SIZED	This `Spliterator` has been created from a source with a known size (for example, a `Set`), so the value returned by `estimatedSize()` is precise.
NON-NULL	It's guaranteed that the traversed elements won't be `null`.
IMMUTABLE	The source of this `Spliterator` can't be modified. This implies that no elements can be added, removed, or modified during their traversal.
CONCURRENT	The source of this `Spliterator` may be safely, concurrently modified by other threads without any synchronization.
SUBSIZED	Both this `Spliterator` and all further `Spliterator`s resulting from its split are `SIZED`.

Advantages:

- Declarative - More concise and readable
- Composable - Greater flexibility
- Parallelizable - Better performance

Disadvantages:

- Cannot be reused

References

- The book “Modern Java In Action”

  Raoul-Gabriel Urma

  Mario Fusco

  Alan Mycroft

- http://ocpj8.javastudyguide.com

THANK YOU FOR YOUR LISTENING!

Reference

- http://ocpj8.javastudyguide.com/ch18.html
- Manning - Modern Java in Action , 2018/9