Functional-style data processing with Java **Streams**

Marco Faella
Corso di Laurea in Informatica
Corso di Linguaggi di Programmazione II
Università di Napoli Federico II

A stream is...

...a sequence of objects...

...supporting **internal iteration**

I.e., the stream library takes *responsibility* for the iteration

Sequences of objects

List

- add
- remove
- search
- scan/iterate

Iterator

- scan/iterate
- (remove)

Stream

- internal iteration

Printing a sequence of objects

```
List<Integer> l = ...;
for (Integer n: l)
    System.out.println(n);
```

Printing a sequence of objects

```
List<Integer> l = ...;
for (Integer n: l)
    System.out.println(n);
```

```
Iterator<Integer> i = ...;
while (i.hasNext())
  System.out.println(i.next());
```

Printing a sequence of objects

```
List<Integer> l = ...;
for (Integer n: l)
    System.out.println(n);
```

```
Iterator<Integer> i = ...;
while (i.hasNext())
  System.out.println(i.next());
```

```
Stream<Integer> s = ...;
s.forEach(i -> System.out.println(i));
```

Internal iteration: The stream takes care of the iteration

Stream Interface

Interface java.util.**Stream<T>**

- 30+ instance methods
- 7 static methods

Internal Iteration

Repeatedly apply an operation to all elements

Responsibility shifts from client to stream library

Types of Operations

Build ops:

create a stream from a data source

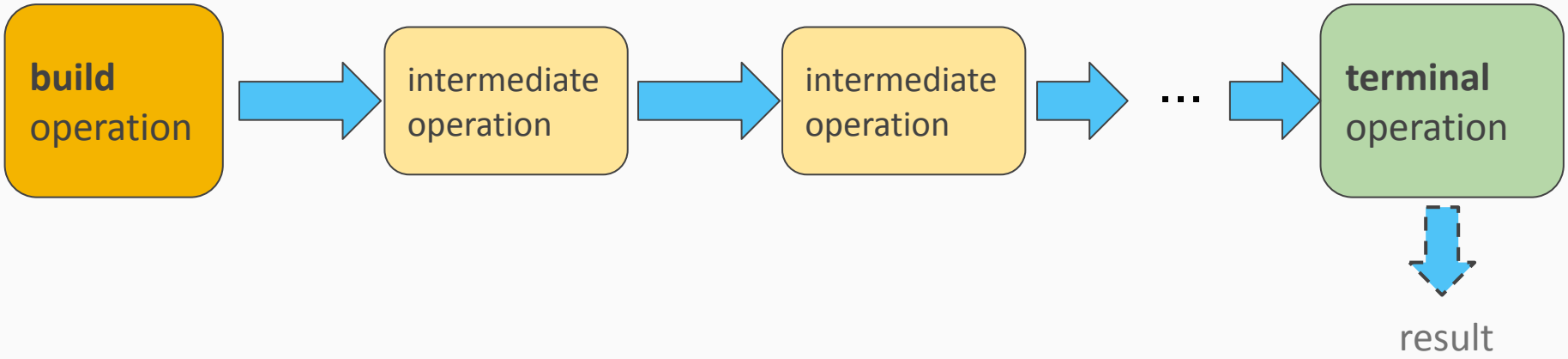
Intermediate ops:

convert one stream into another

Terminal ops:

convert stream into something else (or nothing)

A *Pipeline* of Stream Operations



Example

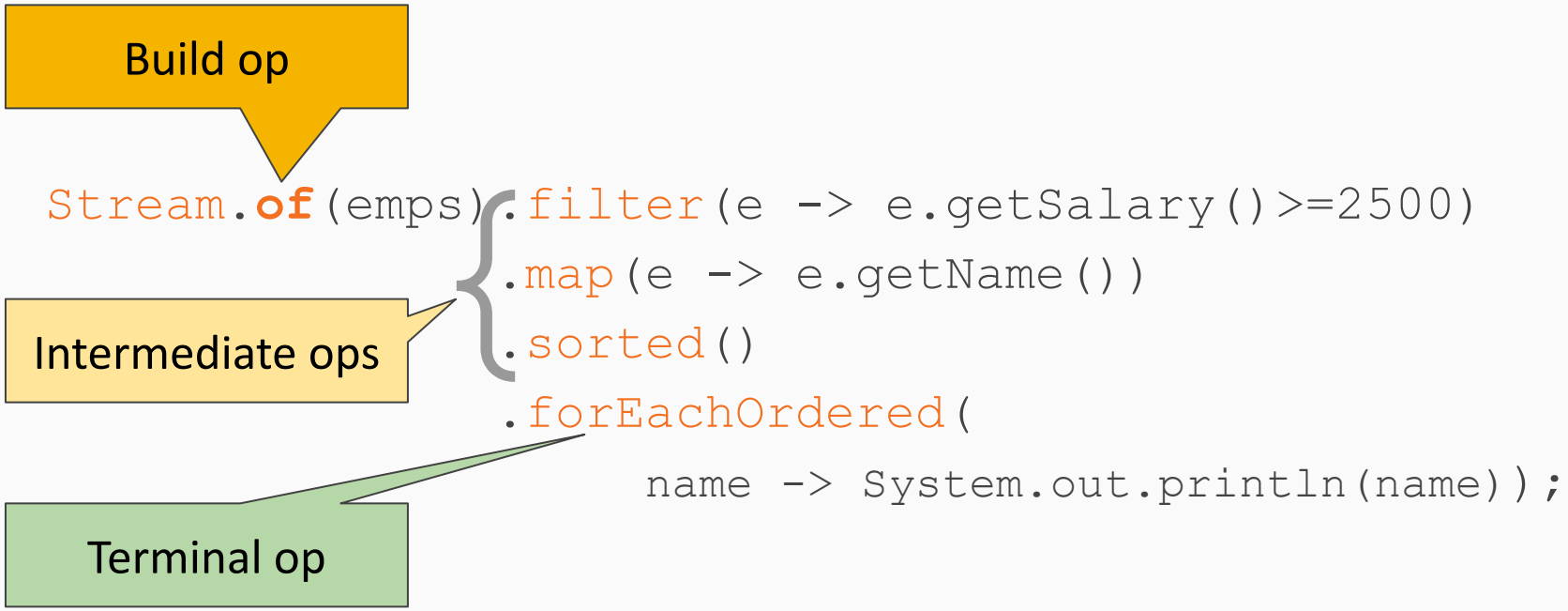
Print the names of the employees with salary at least 2500$, alphabetically sorted

Example

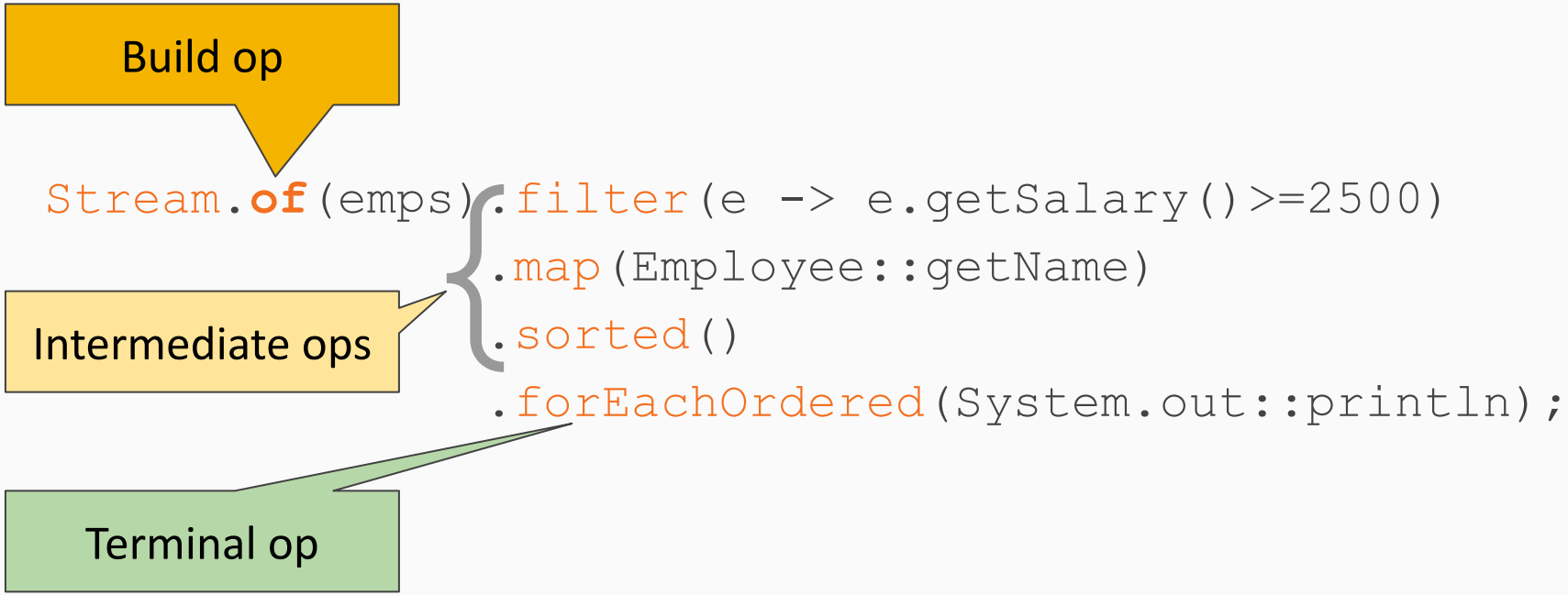
Print the names of the employees with salary at least 2500$, alphabetically sorted

```
Employee[] emps = { /* put employees here */ };

Stream.of(emps).filter(e -> e.getSalary()>=2500)
               .map(e -> e.getName())
               .sorted()
               .forEachOrdered(
                  name -> System.out.println(name));
```

Example


Build op
Intermediate ops
Terminal op
Stream.of(emps).filter(e -> e.getSalary()>=2500)
.map(e -> e.getName())
.sorted()
.forEachOrdered(
name -> System.out.println(name));

Example, with Method References



Types of streams

Streams can be...

...*ordered* or *unordered*

...*sequential* or *parallel*

Ordered vs Unordered Streams

Objects in a stream may have a fixed order, or not

the *encounter order*

Ordered stream:

operations are performed in that order

Unordered stream:

operations may be performed in any order

Sequential vs Parallel Streams

Sequential stream:

operations are performed on one object at a time

Parallel stream:

operations may be performed on several objects in parallel

Creating Streams

Creating streams from...

...a static sequence of objects,

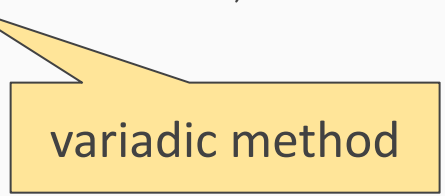
...a collection,

...a computation

Stream from a static sequence of objects

Static method “**Stream.of**”

```
public static <T> Stream<T> of(T ... values)
```



Example:

```
Stream<Integer> fib = Stream.of(1, 1, 2, 3);
```

Stream from a static sequence of objects

Static method “**Stream.of**”

```
public static <T> Stream<T> of(T ... values)
```

Example:

```
Stream<String> italianNumbers =
                    Stream.of(“uno”, “due”, “tre”);
```

Stream from an **array** of objects

Static method "**Stream.of**"

```
public static <T> Stream<T> of(T ... values)
```

Example:

```
Employee[] emps = …
Stream<Employee> empStream = Stream.of(emps);
```

Streams from *.of* are *ordered* and *sequential*

Creating Streams from **Collections**

In interface **Collection**<T>:

```
Stream<T> stream()
Stream<T> parallelStream()
```

Example:

```
Collection<Employee> emps = …
Stream<Employee> empStream = emps.stream();
```

Streams from lists are *ordered*

Streams from sets are *unordered*

Non-interference

Streams must **not** *modify their source collection*

Similar rule for iterators

Terminal Operations

Standard Terminations for Stream<T>

To **void**: forEach, forEachOrdered

To **long**: count

To **Collection<T>**: collect

Terminations to void

Execute a Consumer for each element

```
void        forEach           (Consumer<? super T> c)
```
ignores the encounter order

```
void        forEachOrdered    (Consumer<? super T> c)
```
respects the encounter order

Termination to long

Counts the number of elements

```
long count()
```

Termination into a Collection

Summarize using a **Collector**

On a Stream<T> s:

```
s.collect(Collectors.toList())
```
returns List<T>

```
s.collect(Collectors.toSet())
```
returns Set<T>

No guarantee on the actual type of List/Set

Note: The collect method is much more general…

Stream Operations

Stream Operations

Instance methods of Stream<T>

Intermediate: return Stream<T> or Stream<R>

Terminal: return anything else, or *void*

Filtering Stream Elements

Stream Operations

Filtering Operations

Based on:

- content: `filter`
- amount: `limit`
- uniqueness: `distinct`

Filter Operation

Discards all elements which violate a Predicate

```
Stream<T> filter(Predicate<? super T> property)
```

```
interface Predicate<E> {
    bool test(E elem);
}
```

For a Stream<Employee>, it accepts a Predicate<Person>

Limit Operation

Picks the first `n` elements

```
Stream<T> limit(long n)
```

Distinct Operation

Discards duplicates (according to *equals*)

```
Stream<T> distinct()
```

Example

Objective: Select 10 random positive distinct integers

Example

Objective: Select 10 random positive distinct integers

```
Random rand = new Random();

rand.ints.filter(n -> n>0)
         .distinct()
         .limit(10)
         .forEach(n -> System.out.println(n));
```

Note: this is an IntStream, not a Stream<Integer>

Example

Objective: Select 10 random positive distinct integers

```
Random rand = new Random();


rand.ints.filter(n -> n>0)
         .limit(10)
         .distinct()
         .forEach(System.out::println);
```

We may end up with fewer than 10 numbers!

Example

Objective: Select 10 random positive distinct integers

```
Random rand = new Random();

rand.ints.distinct()
        .filter(n -> n>0)
        .limit(10)
        .forEach(System.out::println);
```

It works,
but why spend time to remove **duplicate negative numbers**?

About the Distinct Operation

One of the few *stateful* intermediate ops

- It does *not* operate independently on each element

Harder to parallelize (see later…)

Transforming and Rearranging

Map Operation

Applies function to every element

In Stream<T>:

```
Stream<R> map(Function<? super T, ? extends R> fun)
```

Can accept a superclass of T and return a subclass of R

Example

Assume Employee contains an Address, which contains a City

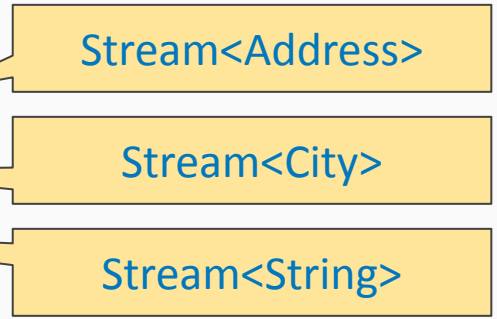
Objective: Print names of cities where at least one employee lives

Example

Assume Employee contains an Address, which contains a City

Objective: Print names of cities where at least one employee lives

```
Stream<Employee> emps = ...;

emps.map(Employee::getAddress)
    .map(Address::getCity)
    .map(City::getName)
    .distinct()
    .forEach(System.out::println);
```



Rearranging Operations

Sorting:

- based on natural order (*Comparable*)
- based on a *Comparator*

Unordering:

- convert stream to unordered
- release ordering guarantees

Sort Operations

According to a custom ordering

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comp)
```

Stateful operations

Example

Objective: Print names of the 10 employees with the highest salary

```
Stream<Employee> emps = ...;

emps.sorted(
            Comparator.comparingInt(Employee::getSalary)
                      .reversed()
    ).limit(10)
     .map(Employee::getName)
     .forEachOrdered(System.out::println);
```

Unordered Operation

Convert stream to unordered

```
Stream<T> unordered()
```

- It does nothing to the data!
- Releases ordering guarantees (just a stream status flag)
- May improve parallel performance (Section 6)

Example

Objective: Efficiently count the number of distinct integers in a list

```
List<Integer> list = ...;

long n = list.parallelStream()
              .unordered()
              .distinct()
              .count();
```

this stream is naturally ordered (from List)

relaxing the order gives **significant speedup**

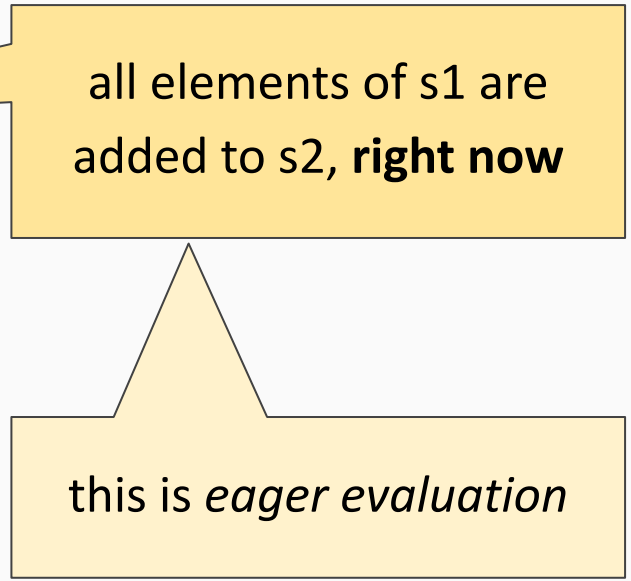
Lazy Evaluation

Lazy = As Late As Possible (on demand)

Eager = As Soon As Possible

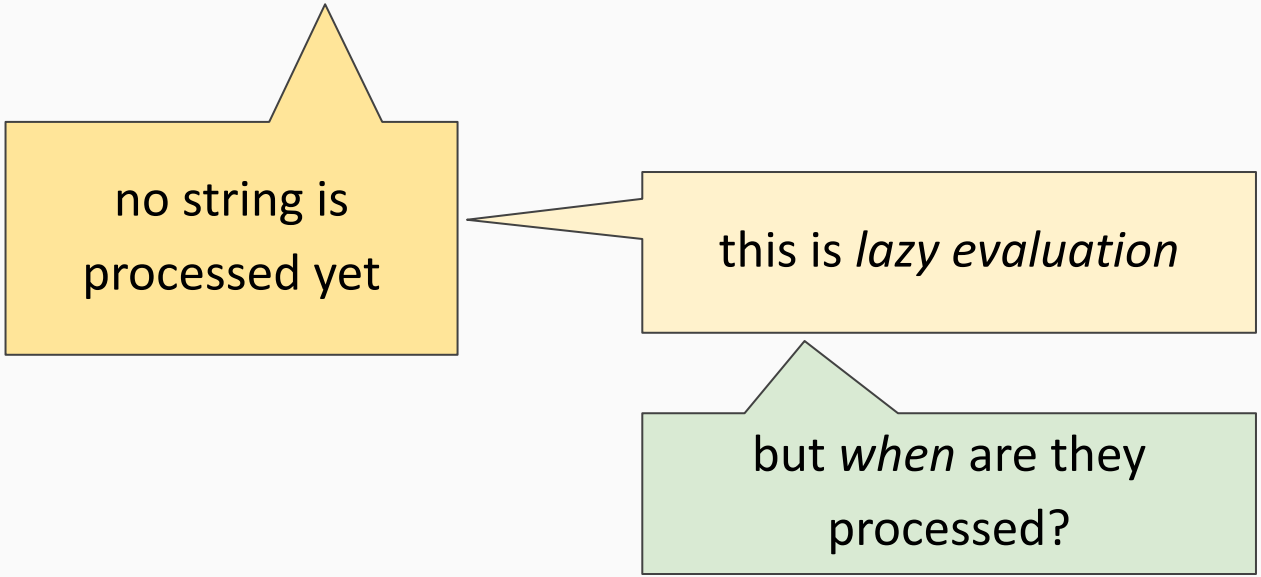
An Eager Example

```
Set<Employee> s1 = …

Set<Employee> s2 = …

s2.addAll(s1);
```



A Lazy Example

```
List<String> content = …

content.stream().distinct();
```



Pushing and Pulling

Elements are iterated and processed *when required by the terminal operation*

I.e.:

elements are *pulled from the end*

not *pushed from the start*

Example `LazyTests.java`

Streams can only be traversed once

Traversing Twice

```
Stream<Integer> fib = Stream.of(1, 1, 2, 3);
fib.forEach(System.out::println);
fib.forEach(System.out::println);
```

→ 1 1 2 3

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

Wrong Way to Split a Stream Pipeline

```
Stream<Integer> fib = Stream.of(1, 1, 2, 3);
fib.limit(2);
fib.forEach(System.out::println);
```

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

Hint: intermediate ops return a *new stream*

Correct Way to Split a Stream Pipeline

```
Stream<Integer> fib = Stream.of(1, 1, 2, 3);
Stream<Integer> shortFib = fib.limit(2);
shortFib.forEach(System.out::println);
```

→ 1 1

Custom Reductions

Stream Operations

Custom Terminal Operations

Summarize a stream into a single object

Functional (i.e., *stateless*): `reduce`

Mutable (i.e., *stateful*): `collect`

A Common Summarization Pattern

Repeatedly apply a binary operation, starting from a *seed*

```
T summary = seed;
for (T t: collection) {
    summary = operation(summary, t);
}
```

Examples:

- sum the elements
- compute the minimum/maximum

A Common Summarization Pattern

Repeatedly apply a binary operation, starting from a *seed*

```
T summary = seed;
for (T t: collection) {
    summary = operation(summary, t);
}
```

With streams:

```
T summary = collection.stream().reduce(seed, operation);
```

Reduce Terminal Operation

Summarize using a BinaryOperator

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

Example: string concatenation

```
Collection<String> words = …
String allWords = words.stream()
                       .reduce("", (a,b) -> a + " " + b);
```

Warning: Inefficient! Quadratic complexity! Use the builtin *joining* collector...

Parallel Reductions

`reduce` can be efficiently parallelized, provided:

1. the binary operation is stateless and associative
2. the seed is an identity for the binary operation

When one of the above is false, `reduce` **will not work correctly** on parallel streams

The Binary Operation Must Be Stateless

State*less*:

```
(String a, String b) -> a + " " + b
(double a, double b) -> a * b
```

State*ful*:

```
(int a, int b) -> {
                    sum += a + b;
                    if (sum>0) return a;
                    else return b;
                  }
```

where `sum` is an accessible field

The Binary Operation Must Be Associative

Order of application is irrelevant:

$$a \; op \; (b \; op \; c) = (a \; op \; b) \; op \; c$$

Associative: addition, multiplication, string concatenation

Not associative: subtraction

$$a - (b - c) = a - b + c \neq (a - b) - c = a - b - c$$

The Seed Must be an Identity

It does not affect the result of the operation:

seed *op* a = a

Examples:

- if *op* is +, seed must be 0
- if *op* is *, seed must be 1
- if *op* is String::concat, seed must be “”

Standard Collectors

Static factory methods of **Collectors**, returning a **Collector**

To strings:	`joining`
To standard collections:	**`toList`**, **`toSet`**, `toCollection`
To maps:	`toMap`, `groupingBy`, `partitioningBy`