Java Stream

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

Version 1.4.1 - May 2018

© Marco Torchiano, 2018

Licensing Note

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

You are free: to copy, distribute, display, and perform the work

Under the following conditions:

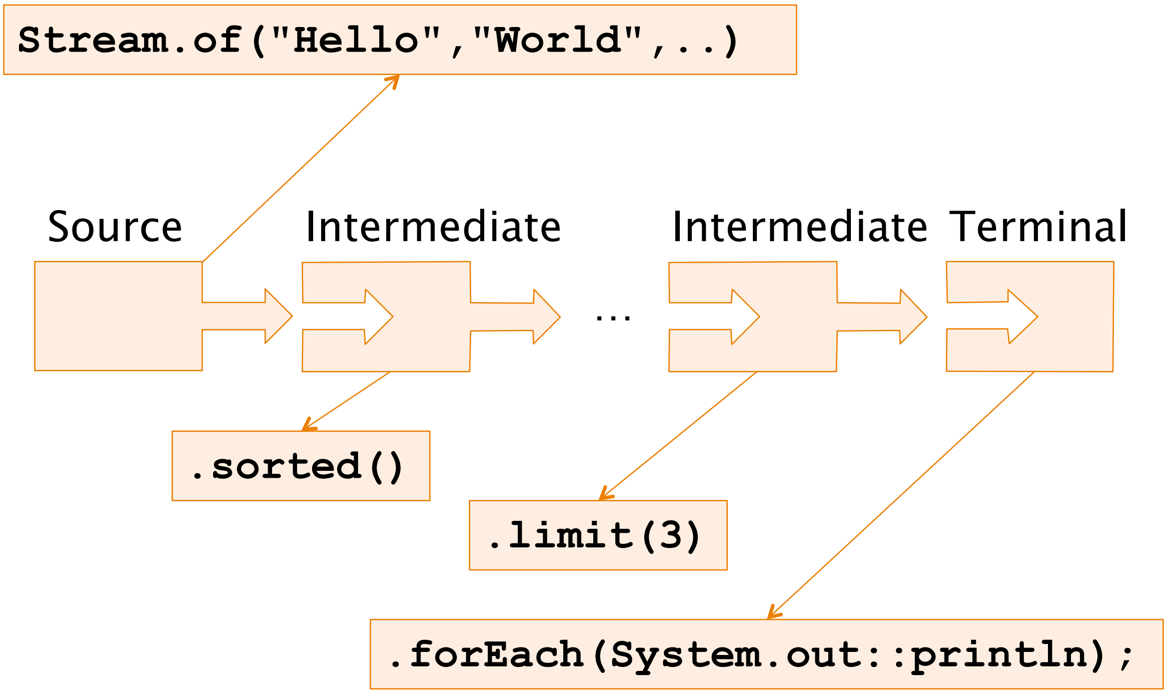
- **Attribution.** You must attribute the work in the manner specified by the author or licensor.
- **Non-commercial.** You may not use this work for commercial purposes.
- **No Derivative Works.** You may not alter, transform, or build upon this work.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Stream

- A sequence of elements from a source that supports data processing operations.
  - Operations are defined by means of behavioral parameterization
- Basic features:
  - Pipelining
  - Internal iteration:
    - no need to write explicit loops statements
  - Lazy evaluation (*pull*):
    - no work until a terminal operation is invoked

Pipelining



`Stream.of("Hello","World",..)`

Source

Intermediate

Intermediate

Terminal

`.sorted()`

`.limit(3)`

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

Source operations

Operation	Args	Purpose
`static Arrays.stream`	`T[]`	Returns a stream from an existing array
`default Collection.stream`	-	Returns a stream from a collection
`static Stream.of`	`T...`	Creates stream from the variable list of arguments/array

Stream source

- Arrays
  - `Stream<T> stream()`

```
String[] s={"Red", "Green", "Blue”}.
Arrays.stream(s)
           .forEach(System.out::println)
```

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

```
Stream.of("Red", "Green", "Blue”).
           forEach(System.out::println);
```

Stream source

- Collection
  - Stream<T> stream()

```
Collection<Student> oopClass =
                    new LinkedList<>();
oopClass.add(new
Student(100,"John","Smith"));
…
oopClass.stream().
          forEach(System.out::println);
```

Source generation

Operation	Args	Purpose
generate()	Supplier<T> s	Elements are generated by calling **get()** method of the supplier
iterate()	T seed, UnaryOperator<T> f	Starts with the seed and computes next element by applying operator to previous element

Stream source generation

- Generate elements using a supplier

```
   Stream.generate(
      () -> Math.random()*10 )
```

- Build from a seed

```
   Stream.iterate( 0,
      (prev) -> prev + 2 )
```

  - Warning: they generate infinite streams

Sample Classes

```
class Student {
  Student(int id, String n, String s) { }
  String getFirst() { }
  boolean isFemale() { }
  Collection<Course> enrolledIn() { }
}
```

```
class Course {
  String getTitle() {}
}
```

Intermediate operations

Return type	Operation	Argument type	Ex. argument
Stream<T>	filter	Predicate<T>	T -> boolean
Stream<T>	limit	int	
Stream<T>	skip	int	
Stream<T>	sorted	*optional* Comparator<T>	(T, T) -> int
Stream<T>	distinct	-	
Stream<R>	map	Function<T, R>	T -> R

Filter

- **default Stream<T> filter(Predicate<T>)**
  - Accepts a predicate
    - a boolean method reference

```
oopClass.stream().
          filter(Student::isFemale).
          forEach(System.out::println);
```

    - a lambda

```
oopClass.stream().
   filter(s->s.getFirst().equals("John")).
   forEach(System.out::println);
```

Intermediate filtering

- **default Stream<T> distinct()**
  - Discards duplicates
- **default Stream<T> limit(int n)**
  - Retains only first n elements
- **default Stream<T> skip(int n)**
  - Discards the first n elements
- **default Stream<T> sorted()**
  - Sorts the elements of the stream
  - Either in natural order or with comparator

Mapping

- **default Stream<R> map(Function<T,R> mapper)**
  - Transforms each element of the stream using the mapper function

```
oopClass.stream().
      map(Student::getFirst).
      map(String::length).
      forEach(System.out::println);
```

Auto-boxing

Mapping primitive variants

- Defined for the main primitive types:

```
IntStream mapToInt(ToIntFunction<T> mapper)
LongStream mapToLong(ToLongFunction<T> m)
DoubleStream mapToDouble(ToDoubleFunction<T>m)
```

  - Improve efficiency

```
oopClass.stream().
      map(Student::getFirst).
      mapToInt(String::length).
      forEach(System.out::println);
```

Flat mapping

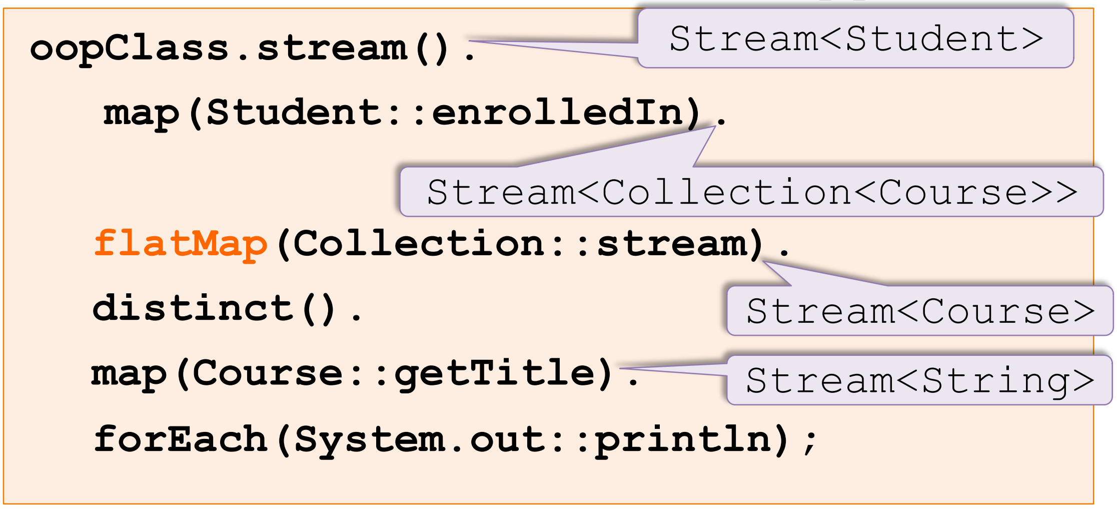
```
<R> Stream<R>
flatMap(Function<T, Stream<R>> mapper)
```

  - Extracts a stream from each incoming stream element
  - Concatenate together the resulting stream

- Typically
  - **T** is a **Collection** (or a derived type)
  - **mapper** can be **Collection::stream**

Flat mapping

- **<R> Stream<R>** flatMap**(**
  Function<T,Stream<R>> mapper)



```
oopClass.stream().
    map(Student::enrolledIn).

   flatMap(Collection::stream).
   distinct().
   map(Course::getTitle).
   forEach(System.out::println);
```

Stream<Student>

Stream<Collection<Course>>

Stream<Course>

Stream<String>

Terminal – Predicate Matching

Operation	Return	Purpose
`anyMatch()`	`boolean`	Checks if any element in the stream matches the predicate
`allMatch()`	`boolean`	Checks if all the elements in the stream match the predicate
`noneMatch()`	`boolean`	Checks if none element in the stream match the predicate
`findFirst()`	`Optional<T>`	Returns the first element
`min() / max()`	`Optional<T>`	Finds the min/max element base on the comparator argument
`count()`	`long`	Returns the number of elements in a stream
`forEach()`	`void`	Consumes each element and applies a lambda to each of them

Optional

- `Optional` represents a potential value
- Methods returning `Optional<T>` make explicit that return value may be missing
  - For methods returning a reference we cannot know whether a null could be returned
  - Force the client to deal with potentially empty optional

Optional

- Access to embedded value through
  - `boolean isPresent()`
    - checks if Optional contains a value
  - `ifPresent(Consumer<T> block)`
    - executes the given block if a value is present.
  - `T get()`
    - returns the value if present; otherwise it throws a `NoSuchElementException`.
  - `T orElse(T default)`
    - returns the value if present; otherwise it returns a `default` value.
  - `T orElse(Supplier<T> s)`
    - when empty return the value supplied value by `s`

Optional

- Provides additional stream-like methods
  - map, filter, etc.
  - Behaves like a stream with 1 or no elements
- Creation uses static factory methods:
  - `of(T v)`:
    - throw exception if `v` is `null`
  - `ofNullable(T v)`:
    - returns an empty Optional when `v` is `null`
  - `empty()`
    - returns an empty Optional

Numeric streams

- More efficient: no boxing and unboxing
- Provided for numeric types
  - `DoubleStream`
  - `IntStream`
  - `LongStream`
- Conversion methods from `Stream<T>`
  - `mapToX()`
- Generator method: `range(start,end)`
- New terminal operations e.g. `average()`

Numeric streams

24 ns per element

```
IntStream seq = IntStream.generate(
                ()-> (int)(Math.random()*100));
int max = seq.limit(10).max().getAsInt();
```

30 ns per element

```
Stream<Integer> seq = IntStream.generate(
                ()-> (int)(Math.random()*100))
                .mapToObj(x -> x);
int max = seq.limit(10)
             .max(naturalOrder()).get();
```

~ 6ns for boxing + unboxing

Kinds of Operations

- Stateless operations
  - No internal storage is required
    - E.g. map, filter
- Stateful operations
  - Require internal storage, can be
    - Bounded: require a fixed amount of memory
      - E.g. reduce, limit
    - Unbounded: require unlimited memory
      - E.g. sorted, collect

Terminal operations

Operation	Arguments	Purpose

`reduce()`	`T,`
	`BinaryOperator<T>`
		Reduces the elements using an identity value and an associative merge operator

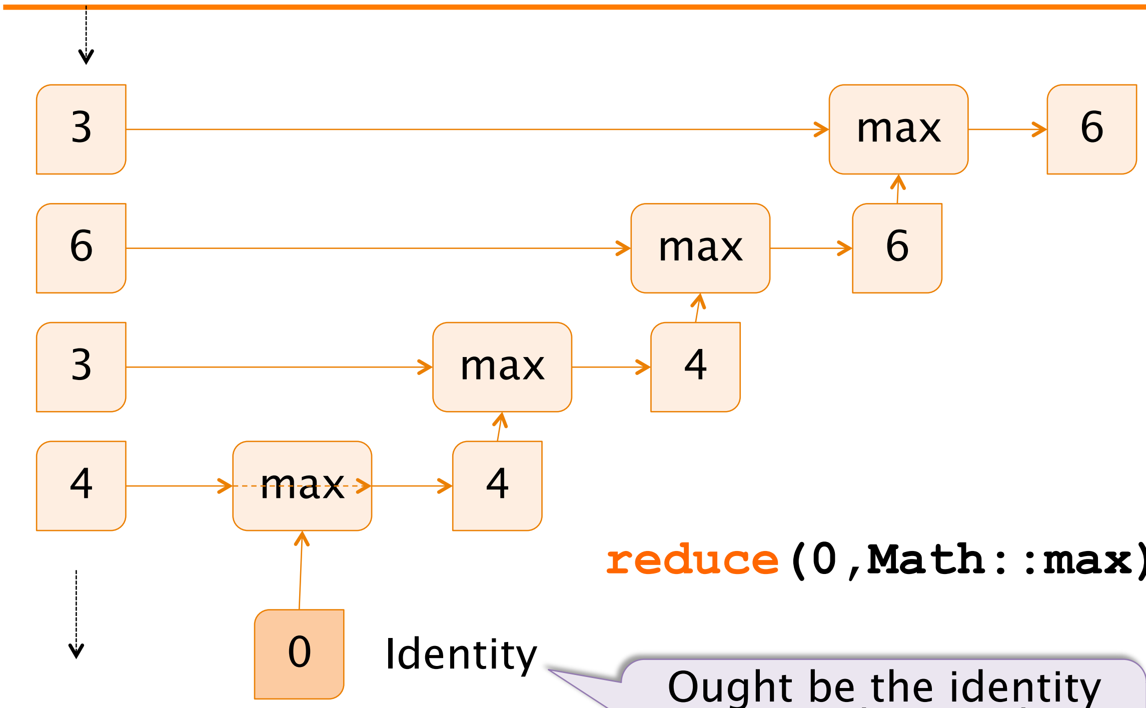
`collect()`	`Collector<T,A,R>`	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

Reducing

- **`T reduce(T identity, BinaryOperator<T> merge)`**
  - Reduces the elements of this stream, using the provided identity value and an associative merge function

```
int m=oopClass.stream().
        map(Student::getFirst).
        map(String::length).
        reduce(0,Math::max);
```

Reducing

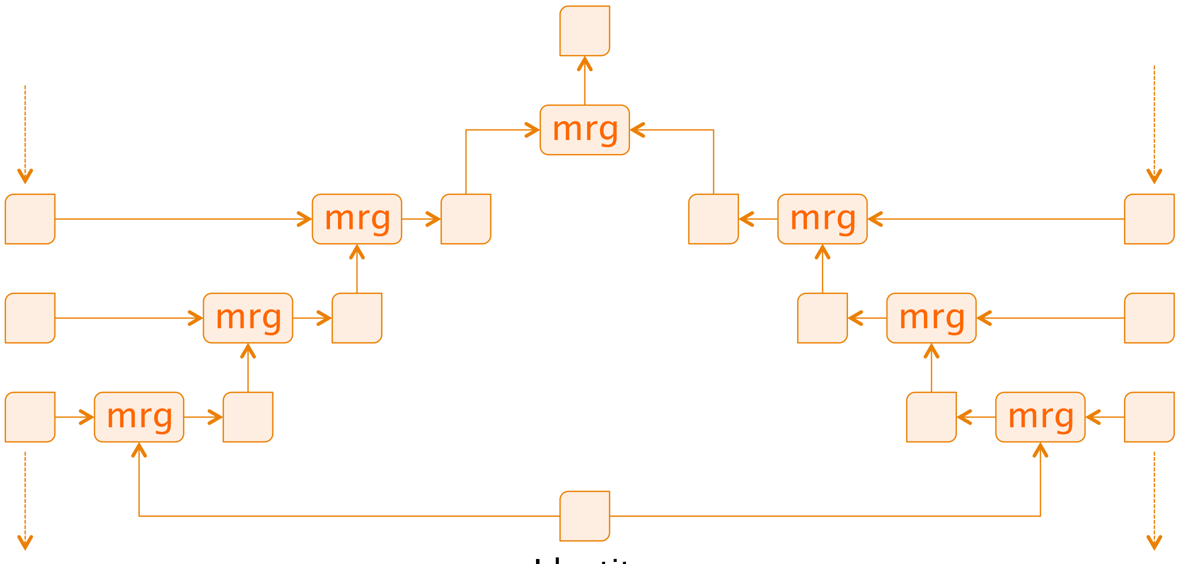


`reduce(0,Math::max)`

Identity

Ought be the identity operand w.r.t. the merge operator

Parallelized reduce



Identity

Collecting

- `Stream.collect()` takes as argument a recipe for accumulating the elements of a stream into a summary result.
  - It is a stateful operation
- Typical recipes available to
  - Summarize (reduce)
  - Accumulate
  - Group or partition

Collector

T : element

A : accumulator

R : result

```
interface Collector<T,A,R>{
   Supplier<A> supplier()
      – Creates the accumulator container
   BiConsumer<A,T> accumulator();
      – Adds a new element into the container
   BinaryOperator<A> combiner();
      – Combines two containers (used for
      – izing)
   Function<A,R> finisher();
      – Performs a final transformation step
}
```

Collector example

```
class addToList<T> implements
Collector<T,List<T>,List<T>>{
public Supplier<List<T>> supplier(){
      return ArrayList<T>::new; }
public BiConsumer<List<T>,T> accumulator(){
   return List<T>::add; }
public BinaryOperator<List<T>> combiner() {
    return(a,b)->{a.addAll(b); return a;}; }
public Function<List<T>,List<T>> finisher()
   { return Function.identity();  }
…
}
```

Collector example

- More compact form:

```
Collector<Student, List<Student>,
             List<Student>> ctl =
 Collector.of(ArrayList::new,
             List::add,
             (a,b)->{a.addAll(b);return a;});
```

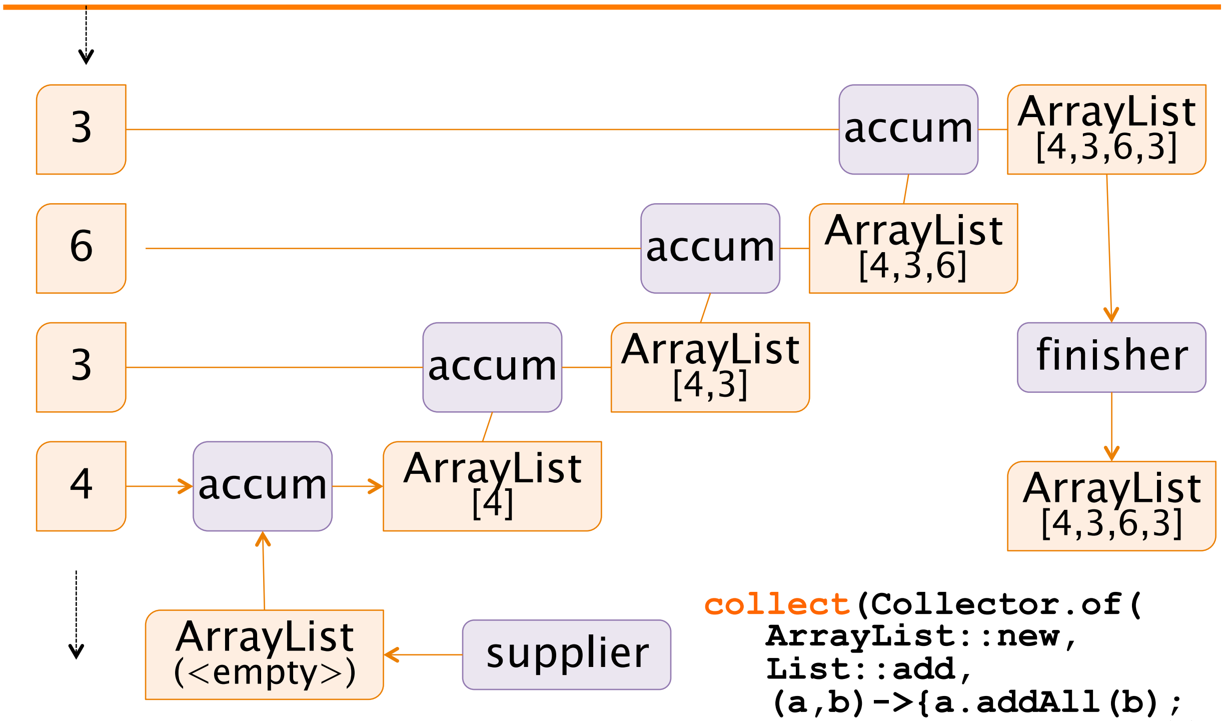
supplier

accumulator

combiner

Implicit finisher => identity transformation

Collector



```
collect(Collector.of(
   ArrayList::new,
   List::add,
   (a,b)->{a.addAll(b);
           return a;}))
```

Characteristics

- Collectors exhibit characteristics that can be used to optimize execution
- Returned by method

  `Set<Characteristics> characteristics()`
- Set of values:
  - `CONCURRENT`
  - `IDENTITY_FINISH`
  - `UNORDERED`

Collector example

- More compact form:

```
String listOfWords = Stream.of(txta)
.map(String::toLowerCase)
.distinct()
.sorted(comparing(String::length).reversed())
.collect(Collector.of(
      ArrayList::new,
      List::add,
      (a,b) -> { a.addAll(b); return a; },
      (Function<List,String>)List::toString));
```

supplier

accumulator

combiner

finisher

Collector and accumulator

- Collector used to compute the average length of a stream of String
  - Uses the **AverageAcc** accumulator object

```
Collector<Integer,AverageAcc,Double>
avgCollector = Collector.of(
        AverageAcc::new,     // supplier
        AverageAcc::addWord,// accumulator
        AverageAcc::merge ,  // combiner
        AverageAcc::average // finisher
);
```

Average Accumulator

```
class AverageAcc {
   private long length;
   private long count;
   public void addWord(String w){
   this.length += w.length(); // accumulator
      count++;   }
   public double average(){    // finisher
      return length*1.0/count; }
   public AverageAcc merge(AverageAcc o){
      this.length+=other.length;
      this.count+=other.count;  // combiner
      return this;}
}
```

Collect vs. Reduce

- Reduce
  - Is bounded
  - The merge operation can be used to combine results from parallel computation threads
- Collect
  - Is unbounded
  - Combining results form parallel computation threads can be performed with the combiner
    - What about the order?

Predefined collectors

- Predefined recipes are returned by static methods of **`Collectors`** class
  - Typically useful to declare:

```
import static java.util.stream.Collectors.*;
```

```
double averageWord = Stream.of(txta)
     .collect(averagingInt(String::length));
```

Summarizing Collectors

Collector	Return	Purpose
`counting()`	`long`	Count number of elements in stream
`maxBy() / minBy()`	`T` (elements type)	Find the min/max according to given Comparator
`summing`*`Type`*`()`	*`Type`*	Sum the elements
`averaging`*`Type`*`()`	*`Type`*	Compute arithmetic mean
`summarizing`*`Type`*`()`	*`Type`*`Summary-Statistics`	Compute several summary statistics from elements

Type can be **`Int`**, **`Long`**, or **`Double`**

Accumulating Collectors

Collector	Return	Purpose
`toList()`	`List<T>`	Accumulates into a new List
`toSet()`	`Set<T>`	Accumulates into a new Set (i.e. discarding duplicates)
`toCollection (Supplier<> cs)`	`Collection<T>`	Accumulate into the collection provided by given Supplier
`joining()`	`String`	Concatenates elements into a new String Optional arguments: separator, prefix, and postfix

Group container collectors

- Returns the three longest words in text:

```
List<String> longestWords = Stream.of(txta)
.filter( w -> w.length()>10)
.distinct()
.sorted(comparing(String::length).reversed())
.limit(3)
.collect(toList());
```

What if two words share the 3rd position?

Grouping Collectors

Collector	Return	Purpose
`groupingBy (Function<T,K> classifier)`	`Map<K, List<T>>`	Map according to the key extracted (by **`classifier`**) and add to list. Optional arguments: – Downstream Collector (nested) – Map factory supplier
`partitioningBy (Function<T, Boolean> p)`	`Map<Boolean, List<T>>`	Split according to partition function (**`p`**) and add to list. Optional arguments: – Downstream Collector (nested) – Map supplier

Example: grouping collectors

- Grouping by feature

```
Map<Integer,List<String>> byLength =
   Stream.of(txta).distinct()
   .collect(groupingBy(String::length));
```

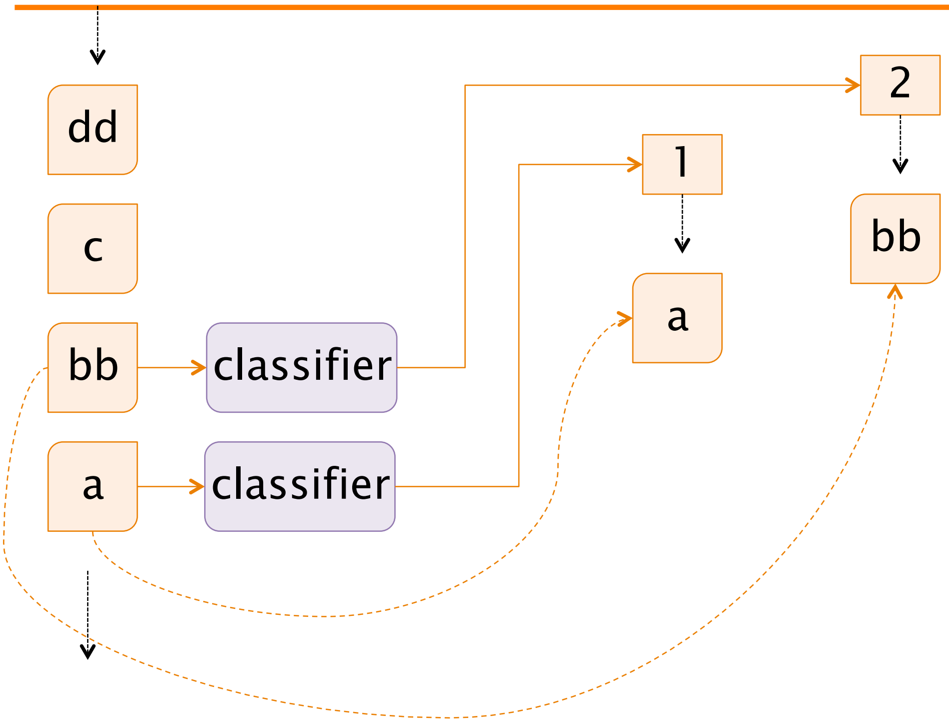
Example: grouping collectors

- Sorted grouping by feature

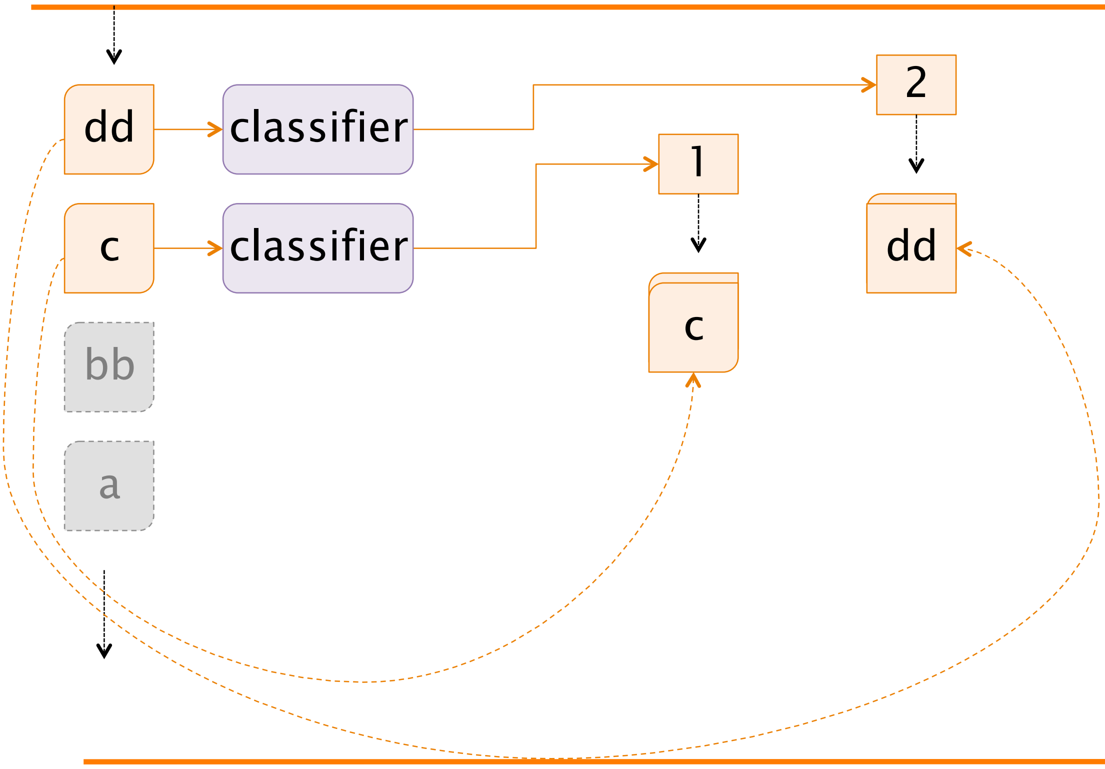
```
Map<Integer,List<String>> byLength =
Stream.of(txta).distinct()
.collect(groupingBy(String::length,
           ()-> new TreeMap<>(reverseOrder()),
           toList()))
```

Map sorted by descending length

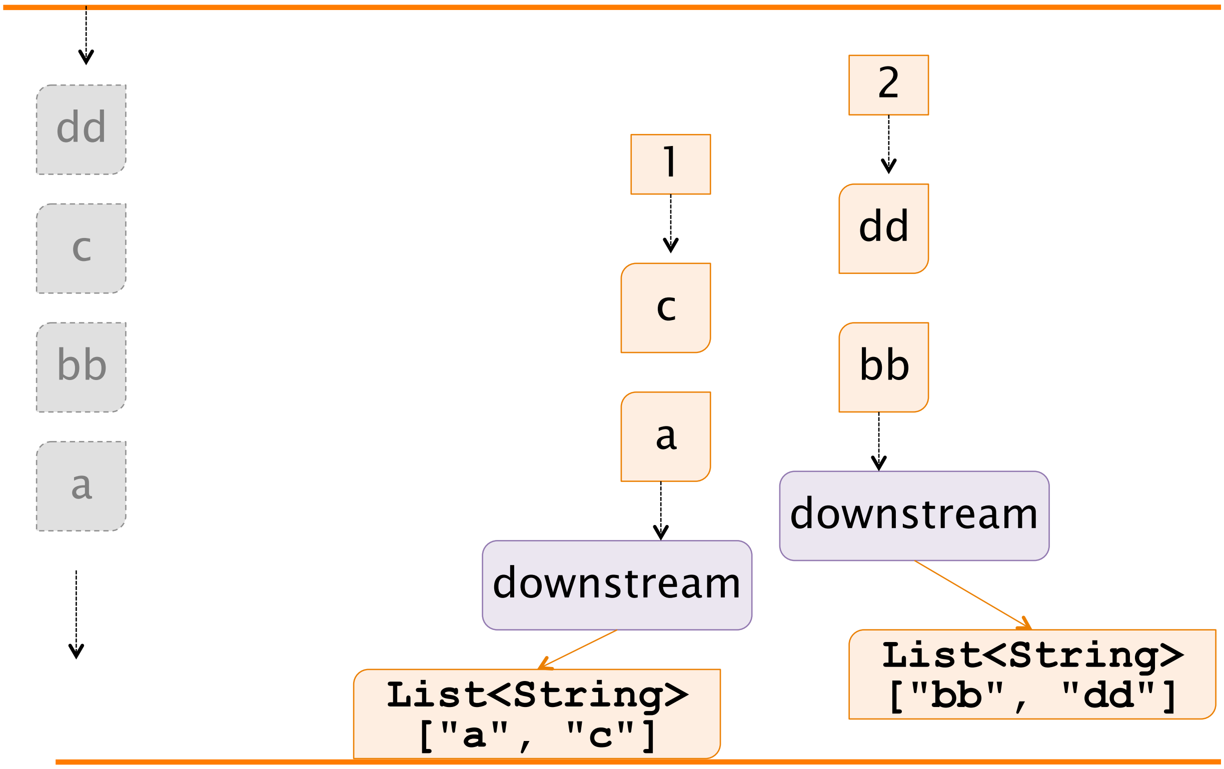
Grouping Collector



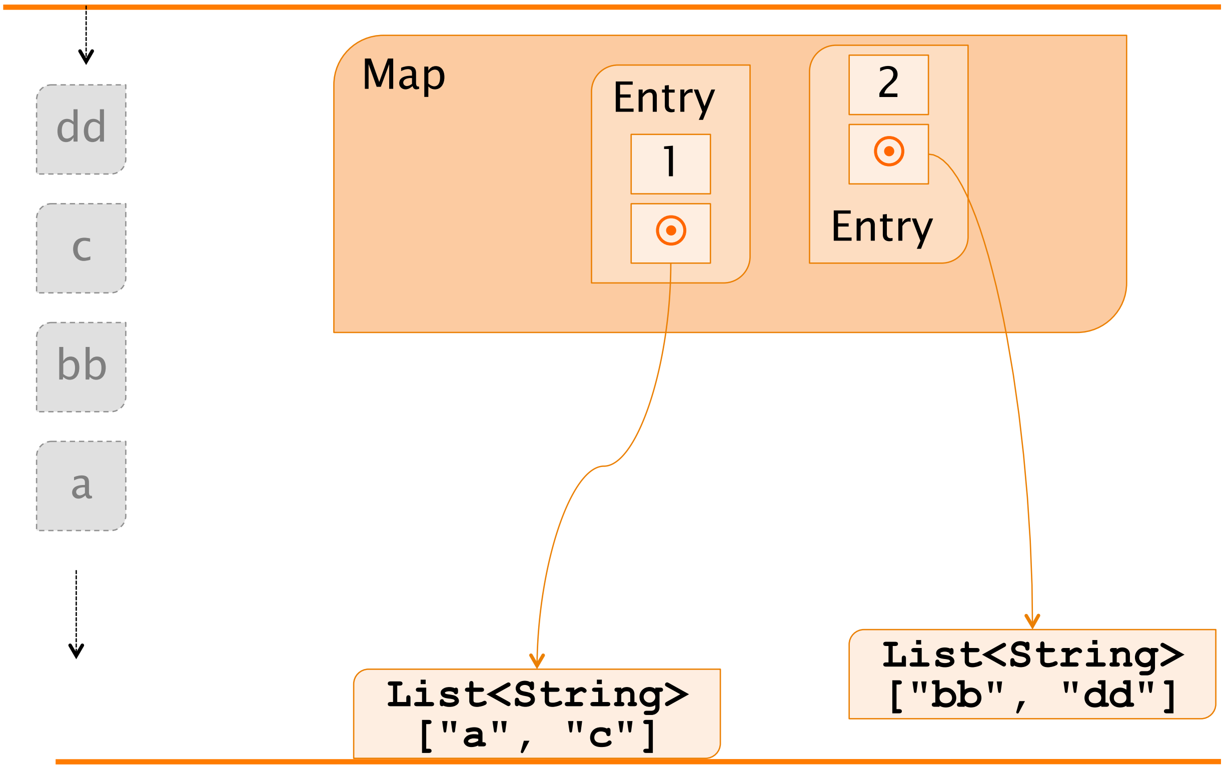
Grouping Collector



Grouping Collector



Grouping Collector



dd

c

bb

a

Map

Entry

1

2

Entry

List<String>
["a", "c"]

List<String>
["bb", "dd"]

Example: grouping collectors

- Re-open the map entry set:

```
List<String> longestWords =
Stream.of(txta).distinct()
.collect(groupingBy(String::length,
          ()->new TreeMap<>(reverseOrder()),
          toList()))
.entrySet().stream()
.limit(3)
.flatMap(e->e.getValue().stream())
.collect(toList());
```

Collector Composition

Collector	Purpose

`collectingAndThen`
`(Collector<T,?,R> cltr,`
`Function<R,RR> mapper)`
	Performs a collection (`cltr`) then transform the result (`mapper`)

`mapping`
`(Function<T,U> mapper,`
`Collector<U,?,R> cltr)`
	Performs a transformation (`mapper`) before applying the collector (`cltr`)

Example: grouping collectors

- Re-open the map entry set:

```
List<String> longestWords =
Stream.of(txta).distinct()
.collect(collectingAndThen(
     groupingBy(String::length,
        ()->new TreeMap<>(reverseOrder()),
        toList())
     ,
     m -> m.entrySet().stream()
        .limit(3)
        .flatMap(e->e.getValue().stream())
        .collect(toList()) );
```

collecting

and then

Summary

- Streams provide a powerful mechanism to express computations of sequences of elements
- The operations are optimized and can be parallelized
- Operations are expressed using a functional notation
  - More compact and readable w.r.t. imperative notation