Java Stream

Object-Oriented Programming

http://softeng.polito.it

Version 2.2.2

Licensing Note

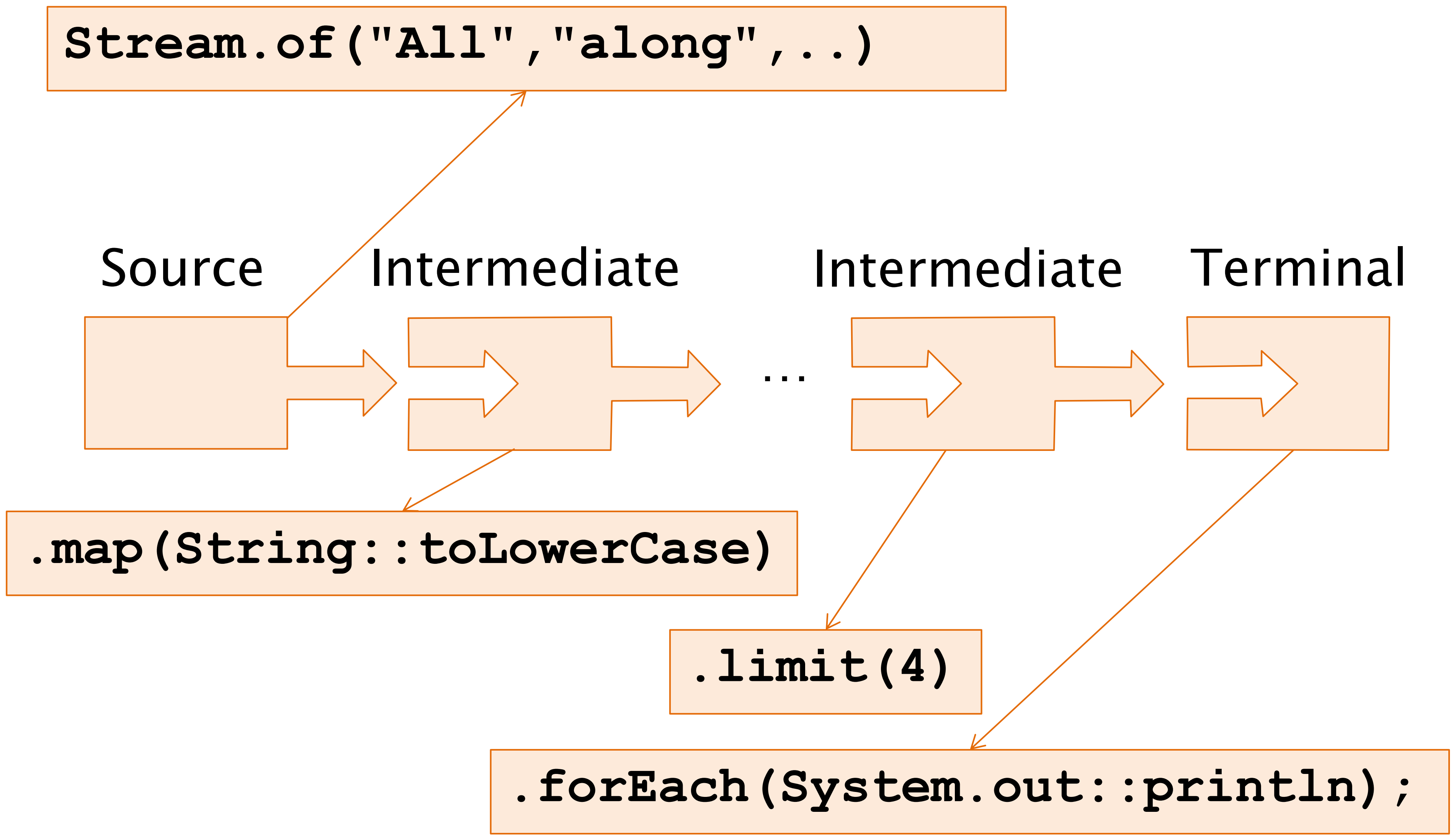
Stream

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 explicit loops statements
  - Lazy evaluation (*pull*):
    - no work until a terminal operation is invoked

Pipelining



```
Stream.of("All","along",..)
.map(String::toLowerCase)
.limit(4)
.forEach(System.out::println);
```

External vs. Internal iteration

```
for( word : lyrics){
   out.println(word);
}
```

- External
  - Mixes how and what
  - Fine grained control
  - Multiple iterations

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

- Internal
  - Focuses on what
  - More concise code
  - Less error prone
  - Transparent optimization

Reminder: functional interfaces

- An interface with exactly one method
- The semantics is purely functional
  - The result of the method depends solely on the arguments
  - There are no side-effects on attributes
- Can be implemented as lambda expressions
- Predefined interfaces are defined in
  - **`java.util.function`**

Lazy evaluation

- Stream pipelines are built first
  - without performing any processing
- Then executed
  - In response to a terminal operation
- **`Supplier<T>`** is used to delay creation of objects until when required, e.g.:
  - Supplier argument in **`collect`** is a factory object as opposed to passing an already created accumulating object

Common Functional Interfaces

Interface	Method
`Function <T,R>`	`R apply(T t)`
`BiFunction <T,U,R>`	`R apply(T t, U u)`
`BinaryOperator <T>`	`T apply(T t, T u)`
`UnaryOperator <T>`	`T apply(T t)`
`Predicate <T>`	`boolean test(T t)`
`Consumer <T>`	`void accept(T t)`
`BiConsumer <T,U>`	`void accept(T t, U u)`
`Supplier <T>`	`T get()`

Primitive specializations

- Functional interfaces handle references
- Specialized versions are defined for primitive types (**`int`**, **`long`**, **`double`**, **`boolean`**)
- Functions: **To*Type*Function**
  Type1ToType2Function
- Suppliers: ***Type*Supplier**
- Predicate: ***Type*Predicate**
- Consumer: ***Type*Consumer**

Source operations

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

Stream source

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

```
String[] s={"One", "Two", "Three"};
Arrays.stream(s)
          .forEach(System.out::println);
```

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

```
Stream.of("One", "Two", "Three").
          forEach(System.out::println);
```

Stream source

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

```
Collection<String> coll =
   Arrays.asList("One", "Two", "Three");
coll.stream().
          forEach(System.out::println);
```

Source generation in `Stream`

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
`empty()`		Returns an empty stream

Stream source generation

- Generate elements using a **`Supplier`**

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

- Generate elements from a seed

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

  - Warning: they generate infinite streams

Numeric streams

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

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

~ 6ns for boxing + unboxing

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	Arg. 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`

Basic 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

Advanced Filtering

- `default Stream<T> filter(Predicate<T>)`
  - Accepts as predicate
    - boolean method reference

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

    - lambda

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

Sorting

- `default Stream<T> sorted()`
  - Sorts the elements of the stream
  - Either in natural order

```
oopClass.stream().
      sorted().
      forEach(System.out::println);
```

  - or with comparator

```
oopClass.stream().
     sorted(comparingInt(Student::getId).
     forEach(System.out::println);
```

Mapping

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

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

Mapping to primitive streams

- 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

- Context:
  - Stream elements are containers (e.g. List)
    - Or elements are mapped to containers
- Problem:
  - Processing should be applied to all elements inside those containers
- Solution:
  - Use the **`flatMap()`** method

Flat mapping

`<R> Stream<R>`

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

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

- Typically
  - **`T`** is a **`Collection<R>`** (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 Operations

Operation	Return	Purpose
`findAny()`	`Optional<T>`	Returns the first element (order **does not** count)
`findFirst()`	`Optional<T>`	Returns the first element (order counts)
`min()` / `max()`	`Optional<T>`	Finds the min/max element based on the comparator argument
`count()`	`long`	Returns the number of elements in the stream
`forEach()`	`void`	Applies the Consumer function to all elements in the stream

Terminal Operation – Predicate

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

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

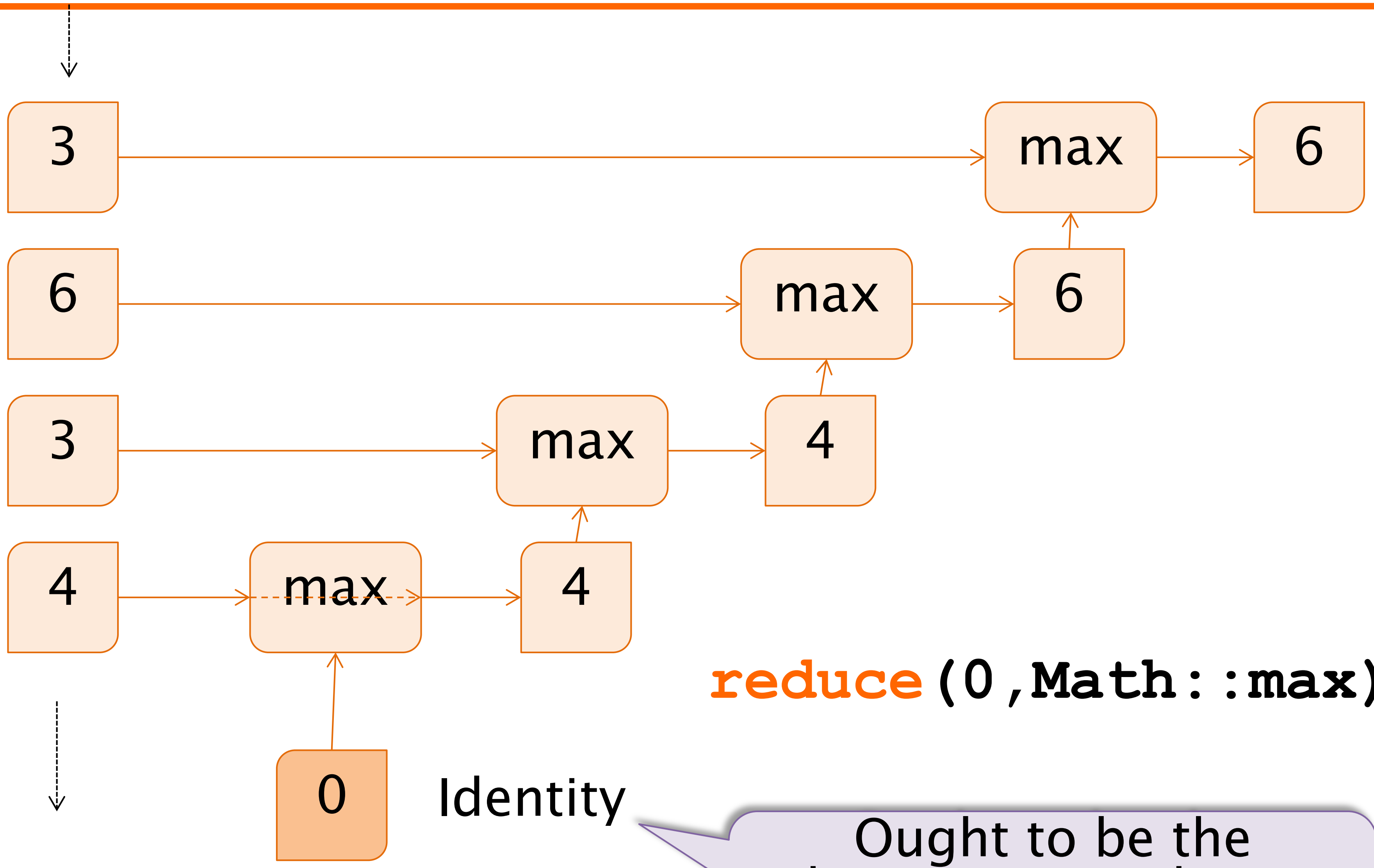
Operation	Arguments	Purpose
`reduce()`	`T, BinaryOperator<T>`	Reduces the elements using an identity value and an associative merge operator
`collect()`	`Collector<T,A,R>`	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

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)
```

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

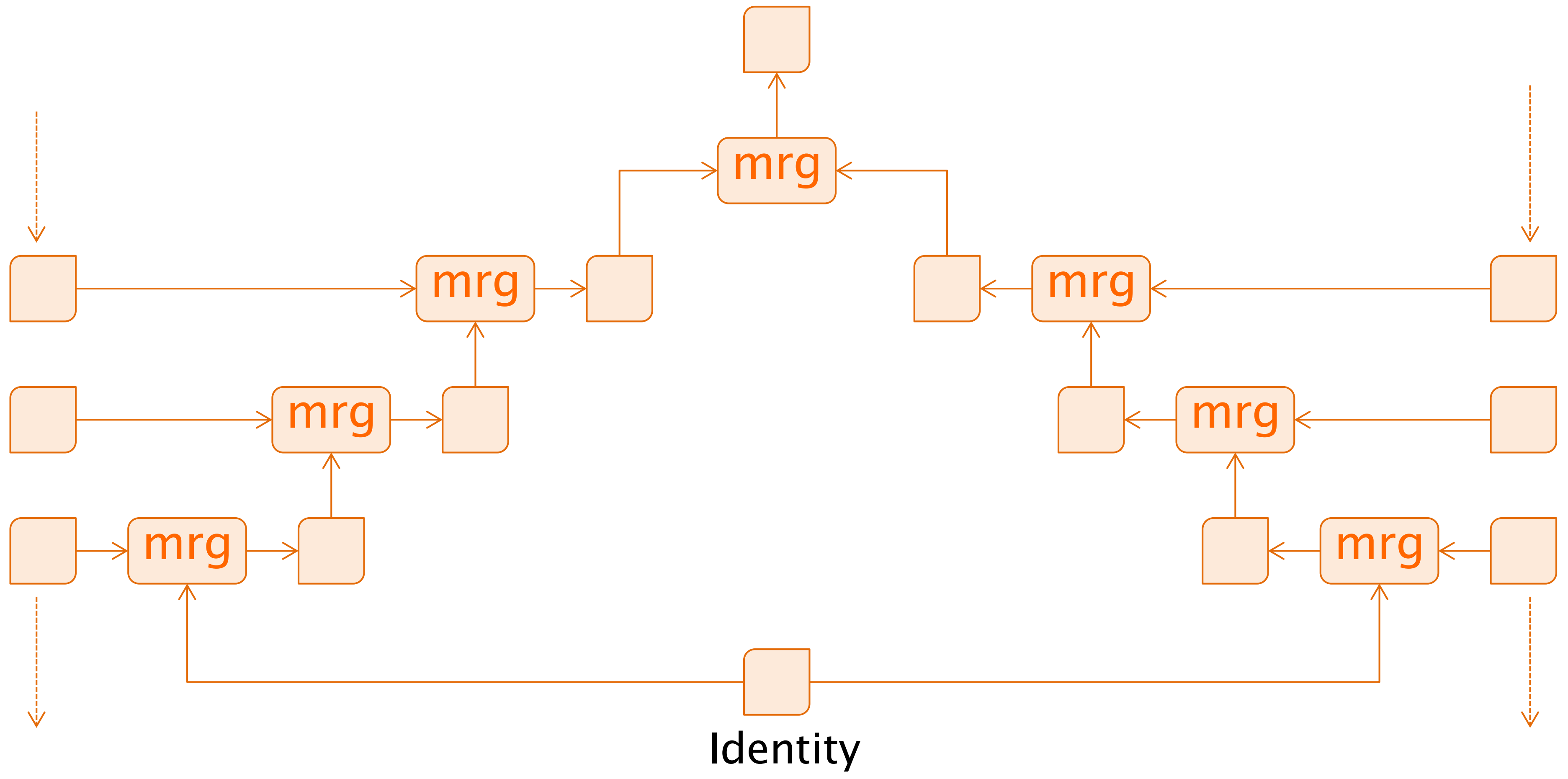
Parallel streams

```
IntStream.of(numbers)
      .reduce(0,Math::max);
```

```
IntStream.of(numbers)
      .parallel()
      .reduce(0,Math::max);
```

Up to n times faster
($n =$ number of CPU cores)

Parallelized reduce

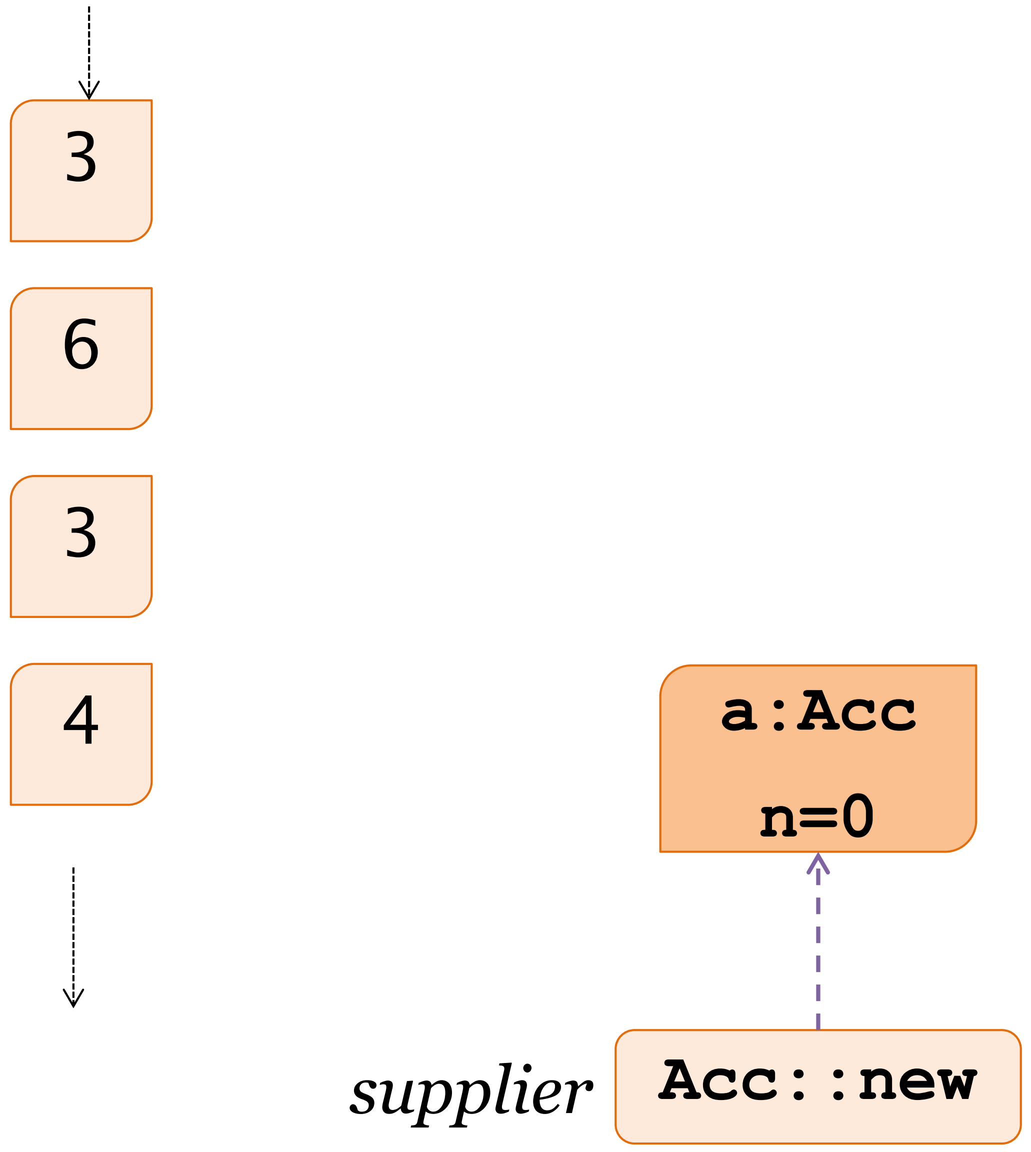


Collecting

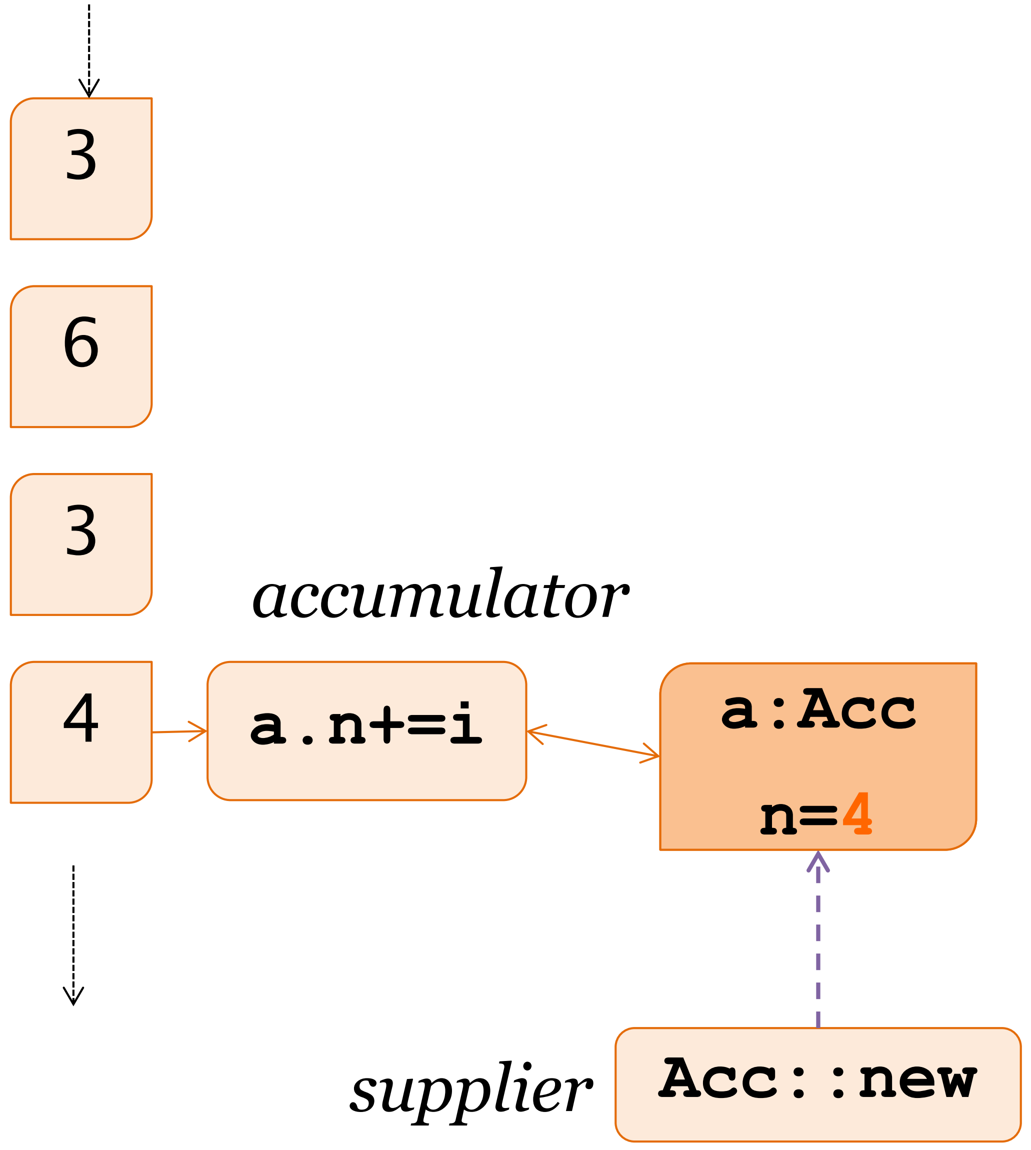
- `Stream.collect()` takes as argument a recipe for accumulating the elements of a stream into a summary result.
  - It is a stateful operation

```
   class Acc { int n; }
   int s = Stream.of(numbers).
    collect(Acc::new,              // supplier
            (a,i) -> a.n+=i,       // accumulator
            (a1,a2)->a1.n+=a2.n    // combiner
        ).n;
```

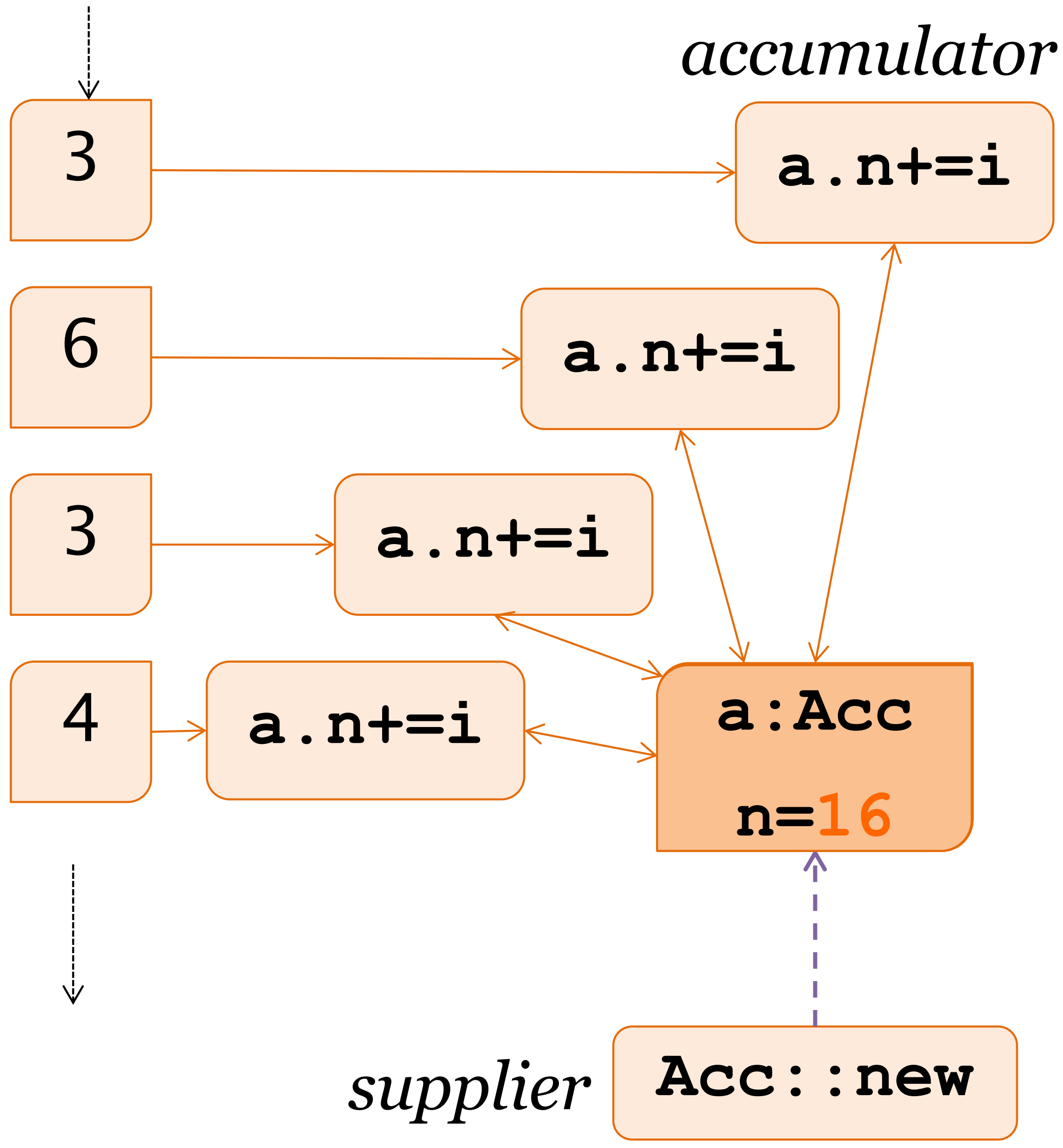
Collecting



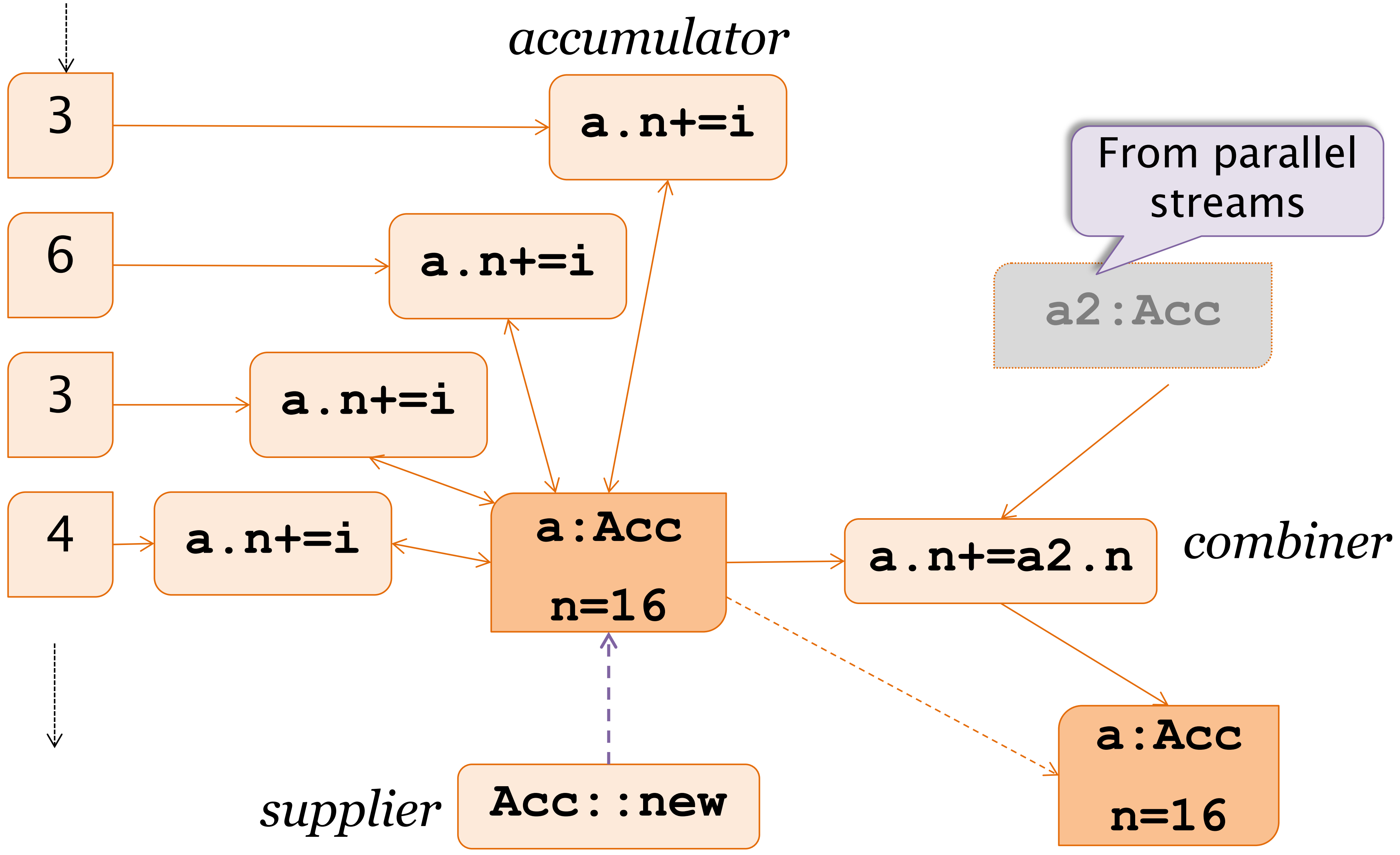
Collecting



Collecting



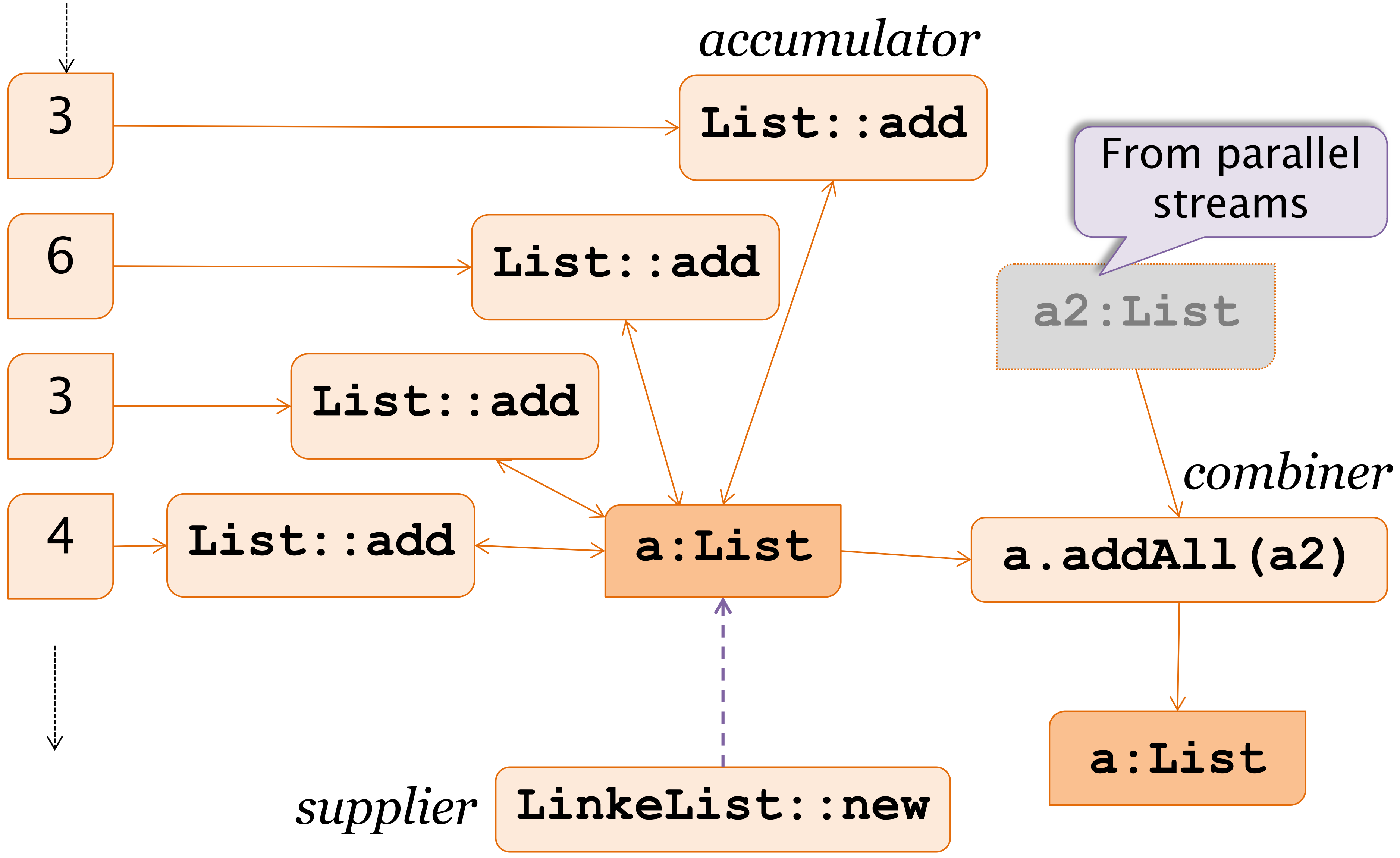
Collecting


accumulator
3
6
3
4
a.n+=i
a.n+=i
a.n+=i
a.n+=i
From parallel streams
a2:Acc
a:Acc
n=16
a.n+=a2.n
combiner
a:Acc
n=16
supplier
Acc::new

Collecting example

```
List<Integer> n = Stream.of(numbers).
collect(LinkedList::new,// supplier
        List::add,      // accumulator
        List::addAll);  // combiner
```

Collecting



accumulator
3
6
3
4
List::add
List::add
List::add
List::add
From parallel streams
a2:List
combiner
a:List
a.addAll(a2)
a:List
supplier
LinkeList::new

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

Java Stream API

PREDEFINED COLLECTORS

Predefined collectors

- Predefined recipes are returned by static methods in `Collectors` class
  - Method are easier to access through:

```
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 into a `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

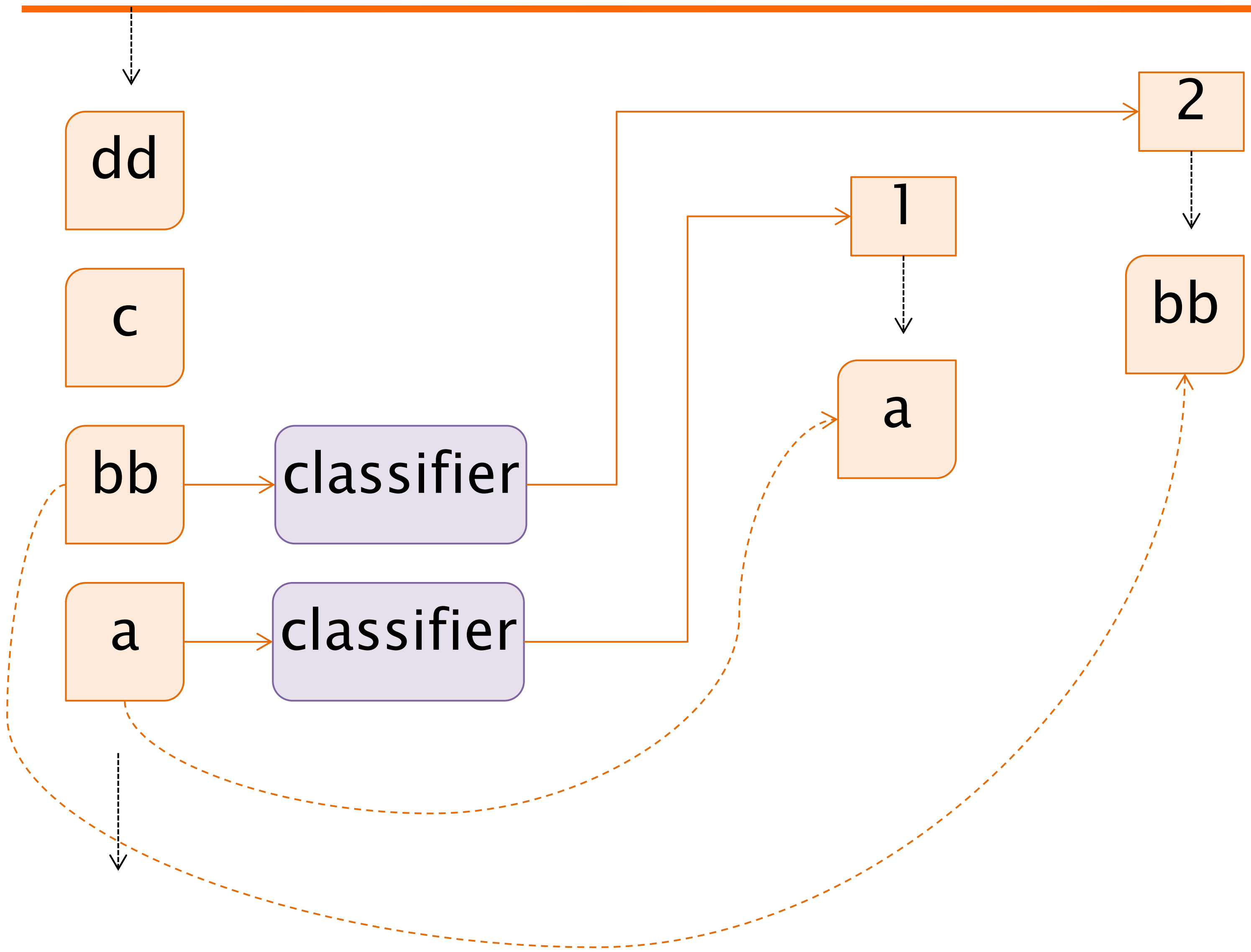
Collector	Return	Purpose
`groupingBy (Function<T,K> classifier)`	`Map<K, List<T>>`	Map according to the key extracted (by **`classifier`**) and add to list. Optional arguments: – Downstream Collector (nested) – Map factory supplier
`partitioningBy (Predicate<T> p)`	`Map<Boolean, List<T>>`	Split according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

Example: grouping collectors

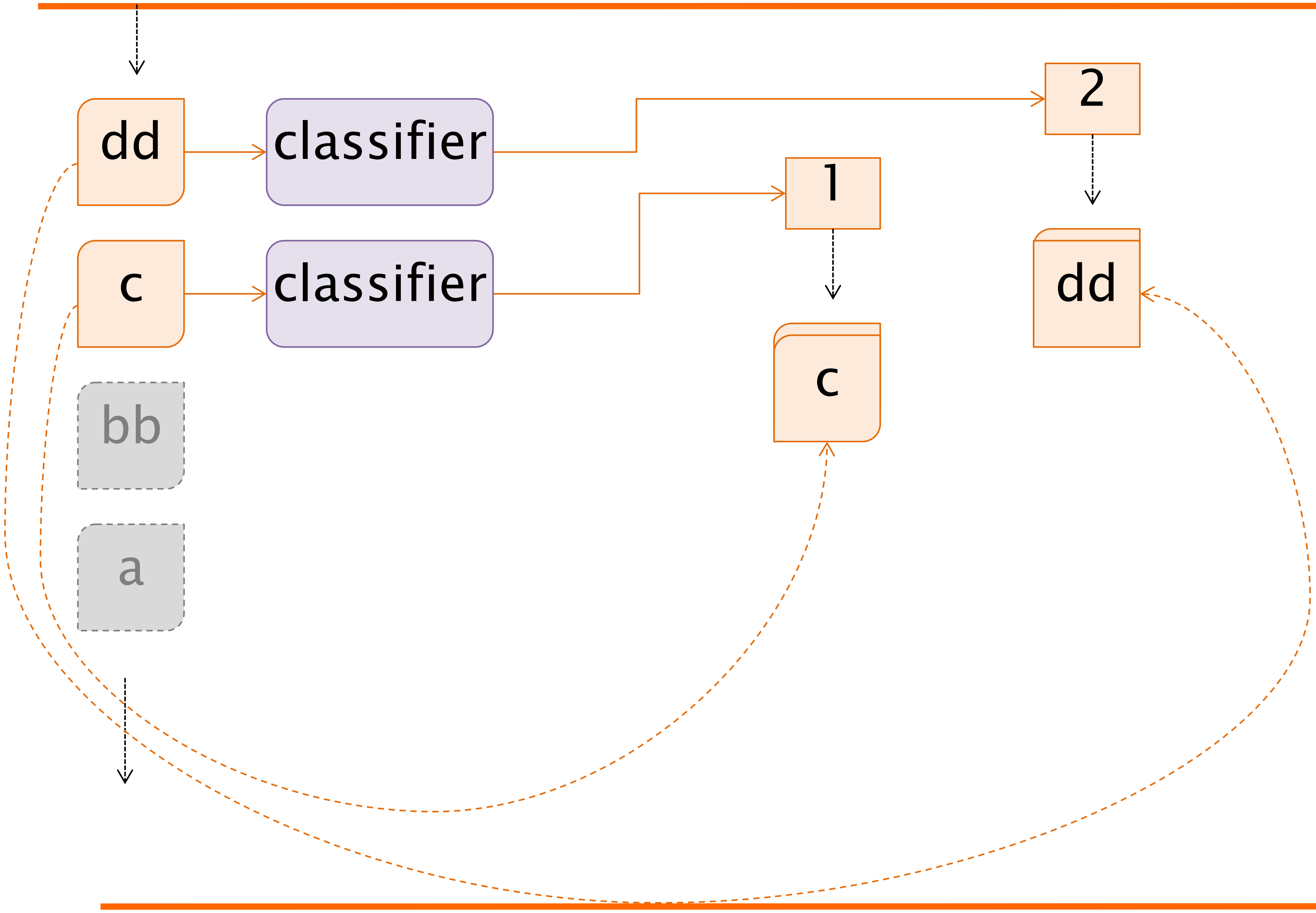
- Grouping by feature

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

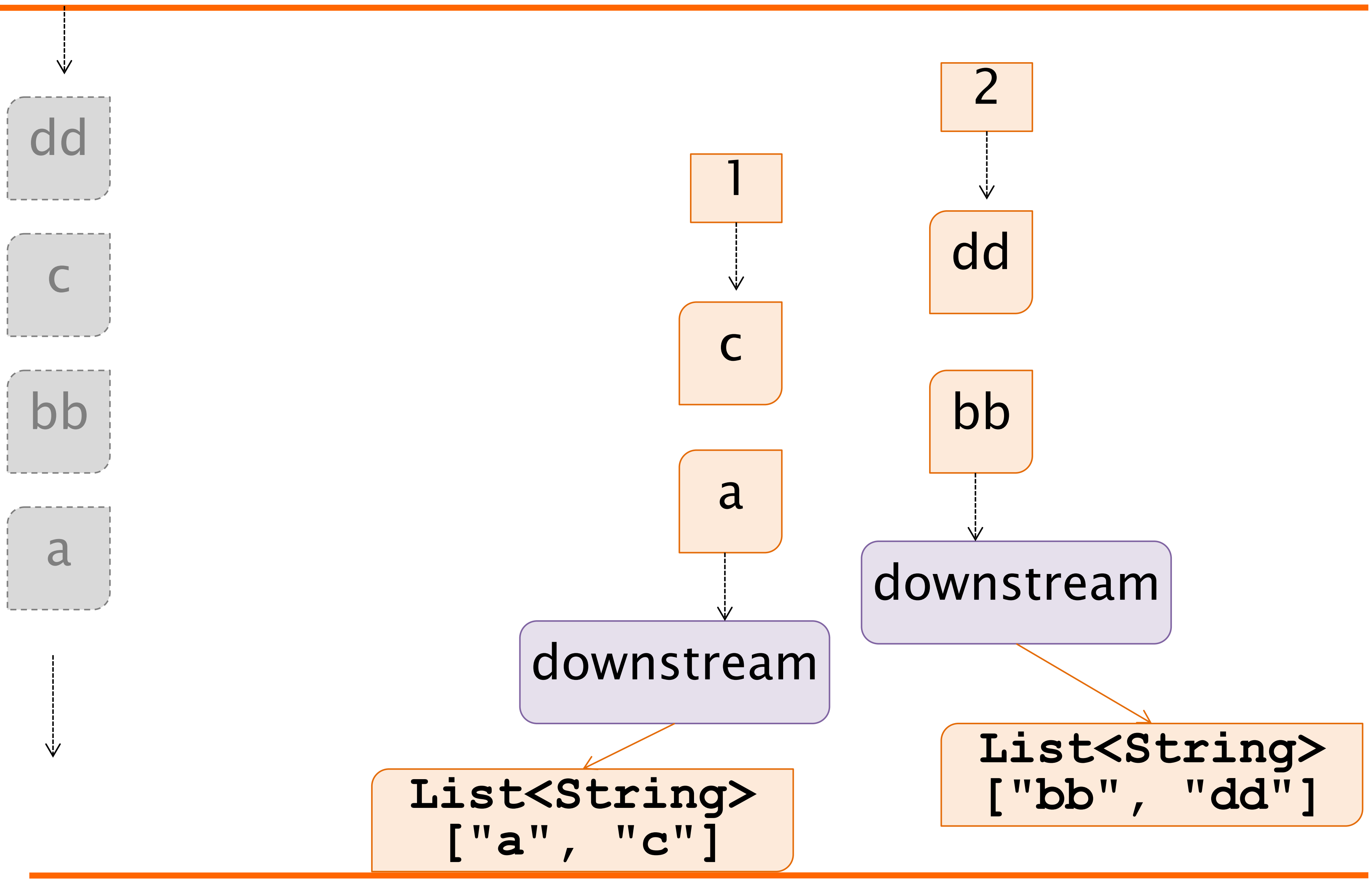
Grouping Collector



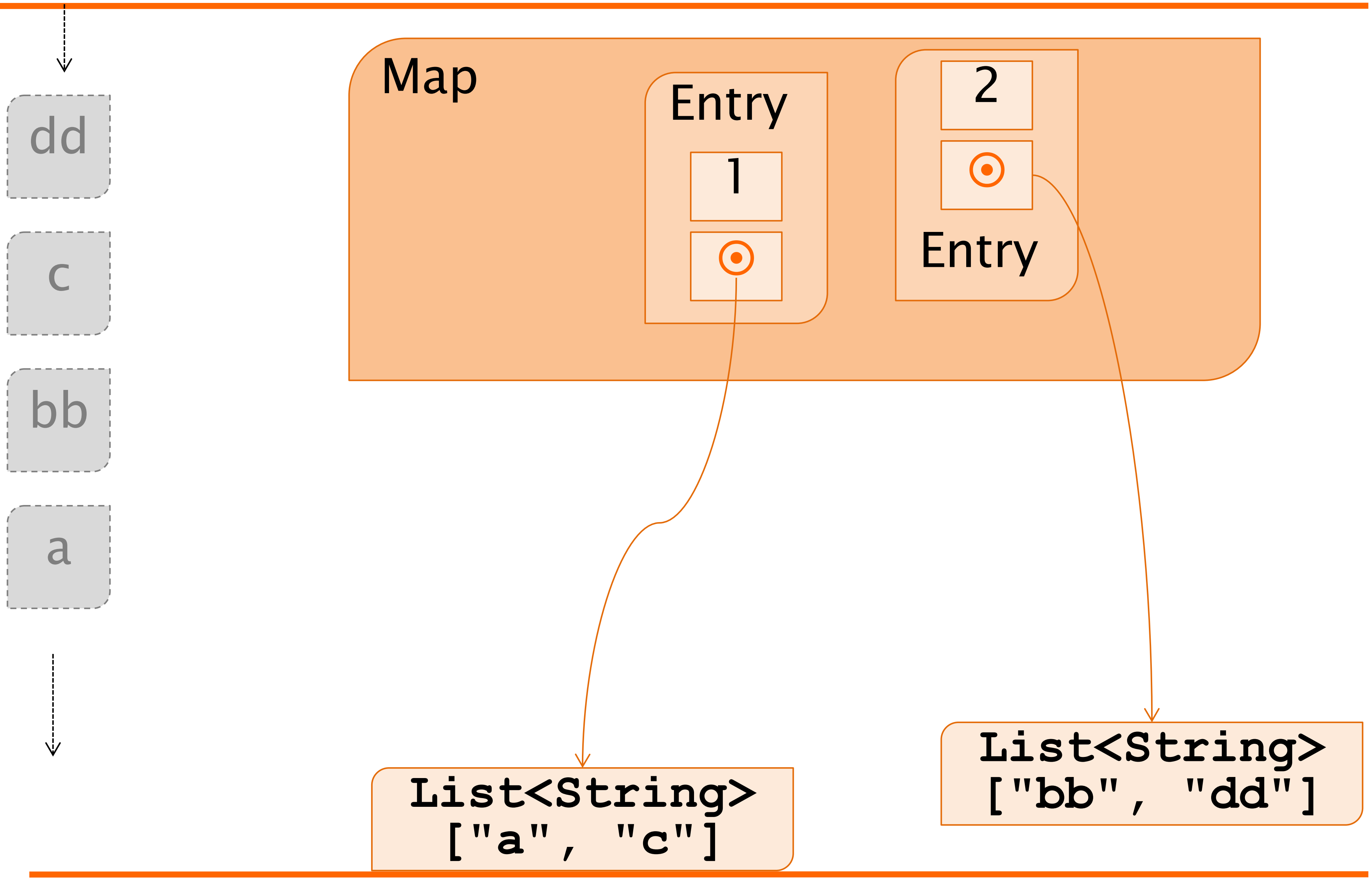
Grouping Collector



Grouping Collector



Grouping Collector



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

Example: grouping collectors

- Grouping and counting

```
Map<String,Long> frequency =
Stream.of(txta)
.collect(groupingBy(
          w->w,
          counting()));
```

Grouped by word

Downstream is counting

Example: grouping collectors

- Stream of map entries:

```
List<String> freqSorted =
Stream.of(txta)
.collect(groupingBy(w->w, counting()))
.entrySet().stream()
.sorted(
  comparing(Map.Entry<String,Long>::getValue)
  .reversed()
  .thenComparing(Map.Entry::getKey))
.map( e -> e.getValue() + ″:" + e.getKey())
.collect(toList());
```

Collector Composition

Collector	Purpose
`collectingAndThen (Collector<T,?,R> cltr, Function<R,RR> mapper)`	Apply a transformation (**`mapper`**) after performing collection (**`cltr`**)
`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(                          collecting
        groupingBy(String::length,
           ()->new TreeMap<>(reverseOrder()),
           toList())
        ,                                             and then
        m -> m.entrySet().stream()
           .limit(3)
           .flatMap(e->e.getValue().stream())
           .collect(toList()) );
```

Example: collecting and then

- Stream of map entries:

```
Stream.of(txta)
.collect(collectingAndThen(
  groupingBy(w->w, counting())                    collecting
,
  m->m.entrySet().stream()                        and then
  .sorted(comparing(Map.Entry::getValue)
  .map(e->e.getValue()+ ":" +e.getKey())
  .collect(toList())))
```

Example: mapping

- Stream of map entries:

```
Stream.of(txta)
.collect(collectingAndThen(
   groupingBy( w->w, counting())
,
   m->m.entrySet().stream()
   .collect(groupingBy(
        Map.Entry::getValue,
        mapping(Map.Entry::getKey,
            toList()))))));
```

Custom collectors

Collector

T : element

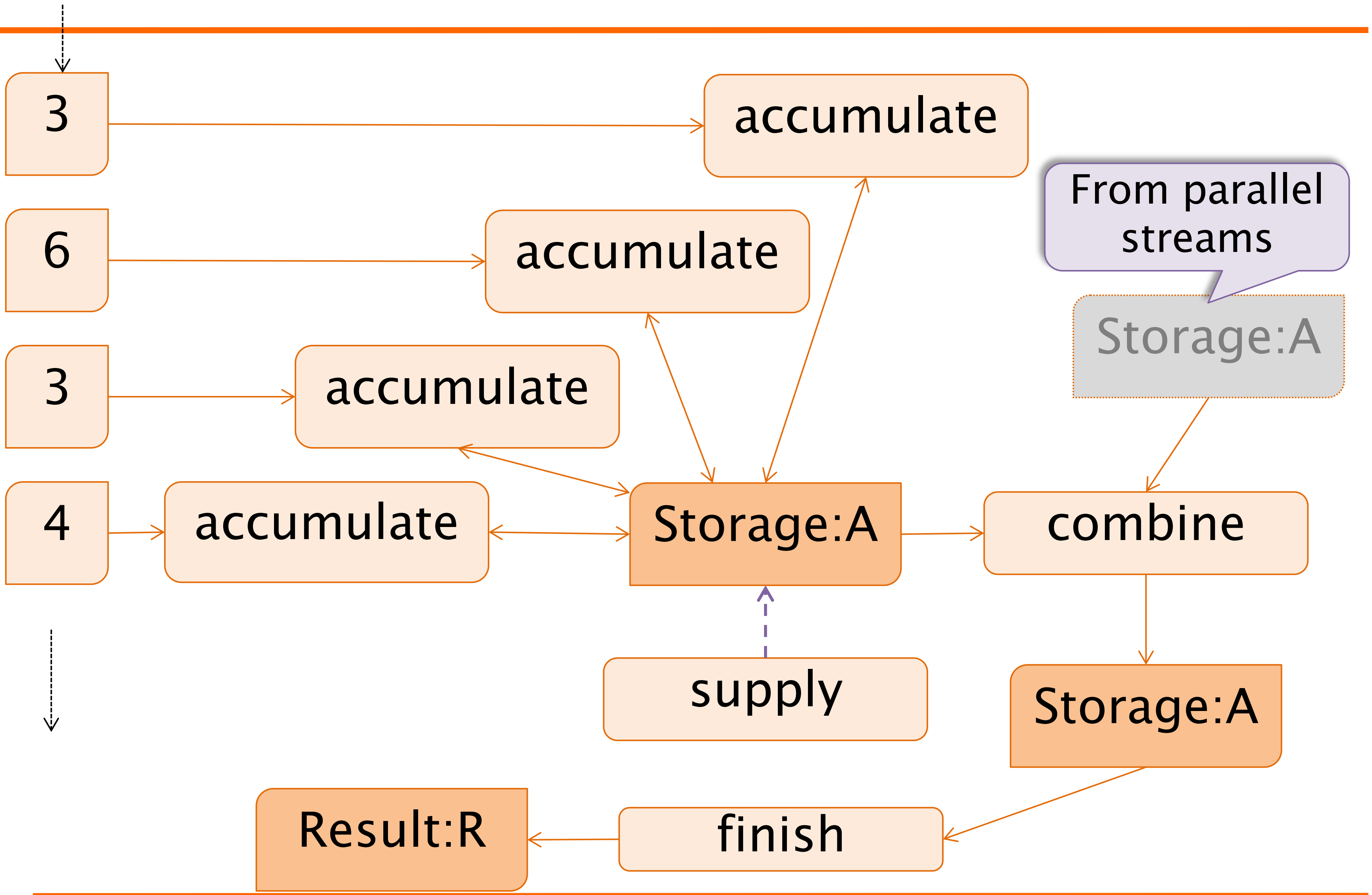
A : accumulator

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 parallelizing)

Operator, not consumer!
```
    Function<A,R> finisher();
```
– Performs a final transformation step
```
    Set<Characteristics> characteristics();
```
– Capabilities of this collector
```
}
```

Collecting


3
6
3
4
accumulate
accumulate
accumulate
accumulate
From parallel streams
Storage:A
Storage:A
combine
supply
Storage:A
finish
Result:R

Collector.of

```
static Collector<T,A,R> of(
  Supplier<A> supplier,
  BiConsumer<A,T> accumulator,
  BinaryOperator<A> combiner,
  Function<A,R> finisher,
  Characteristic... characts)
```

optional

- More compact than extending the interface **Collector**

Collector.of

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

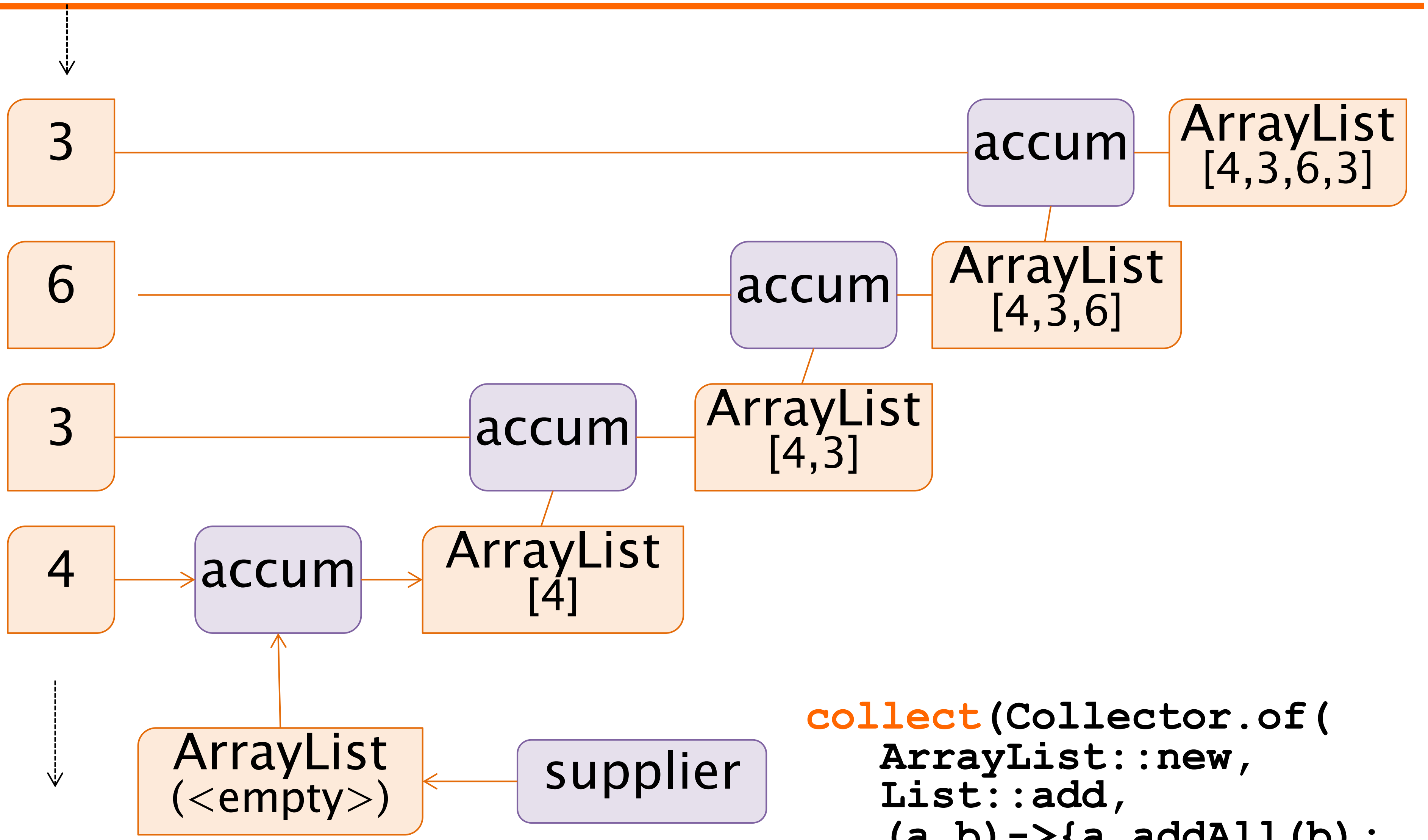
supplier

accumulator

combiner

Implicit finisher => identity transformation
No characteristics

Collector



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

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; },
        List::toString));
```

supplier

accumulator

combiner

finisher

Characteristics

- **`IDENTITY_FINISH`**
  - Finisher function is the identity function therefore it can be elided
- **`CONCURRENT`**
  - Accumulator function can be called concurrently on the same container
- **`UNORDERED`**
  - The operation does require stream elements order to be preserved

Characteristics

- Characteristics can be used to optimize execution
- If both **`CONCURRENT`** and **`UNORDERED`**, then, when operating in parallel,
  - Accumulator method is invoked concurrently by several threads
  - Combiner is not used

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;}
}
```

Exceptions and Functional Interfaces

Exceptions and Functional If.

- Functional interfaces largely used in Stream API do not include exceptions
- Lambdas and method reference are not allowed to throw (checked) exceptions
  - This is mainly due to the intermediate code that does not handle exceptions
  - Unchecked exceptions works as usual and interrupt the stream processing

Exceptions in streams

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> convert(s) )
.sum();
```

Unhandled exception

```
static int convert(String s)
                          throws NotANumber {
    try { return Integer.parseInt(s);
    }catch(NumberFormatException e) {
      throw new NotANumber(e); }}
```

Exception & streams: remedies

- Bury the exception:
  - catch the exception and ignore it
- Wrap the exception
  - use **`RuntimeException`** pointing to the original one
- Sneaky exceptions
  - throw as unchecked (using a trick)
- Annotate the results

Bury the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
   try{ return convert(s); }
   catch(NotANumber e)
   { return 0; }
})
.sum();
```

No exception is thrown. Just uses 0 when a parsing error occurs

Wrap the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
   try{ return convert(s); }
   catch(NotANumber e)
   {throw new RuntimeException(e);
   }})
.sum();
```

A `RuntimeException` is thrown whose cause is the `NotANumber`

Sneakily throw the exception

```
Stream.of("1","2","B","6","30")
.mapToInt( s -> {
   try{ return convert(s); }
   catch(NotANumber e)
   {sneakyThrow(e);
    return -1; }})
.sum();
```

Never executed but required for syntax reasons

A **NotANumber** is thrown but the compiler does not complain

Sneaky throw method

- Using generics type erasure it is possible to "uncheck" an exception

Replaced by **`RuntimeException`**

```
static <E extends Throwable>
void sneakyThrow(
  Exception exception) throws E {
  throw (E)exception;
}
```

Due to type erasure the cast is never applied in practice

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