Java Stream

Object Oriented Programming

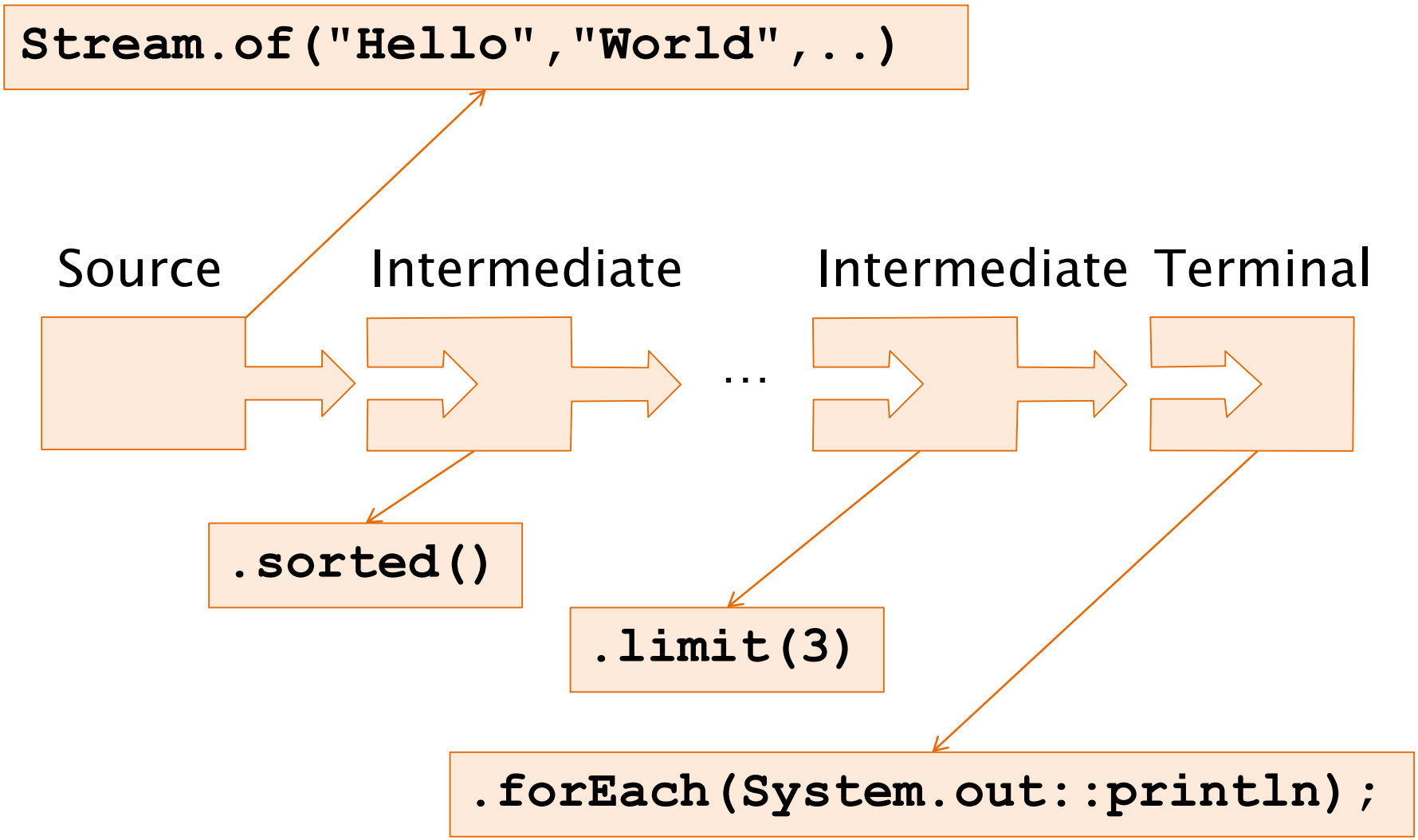
SOftEng
http://softeng.polito.it

Version 1.2.2 - April 2016

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

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);
```

Intermediate operations

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

Filter

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

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

  - Can use 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 element 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
`findAny`	`Optional<T>`	Returns any element
`min`	`Optional<T>`	Finds the min element base on the comparator argument
`max`	`Optional<T>`	Finds the max element base on the comparator argument

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

Terminal operations

Operation	Arguments	Purpose
`forEach`	Consumer<T>	Consumes each element from a stream and applies a lambda to each of them. The operation returns **`void`**.
`count`		Returns the number of elements in a stream as a **`long`**.
`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.

Example:

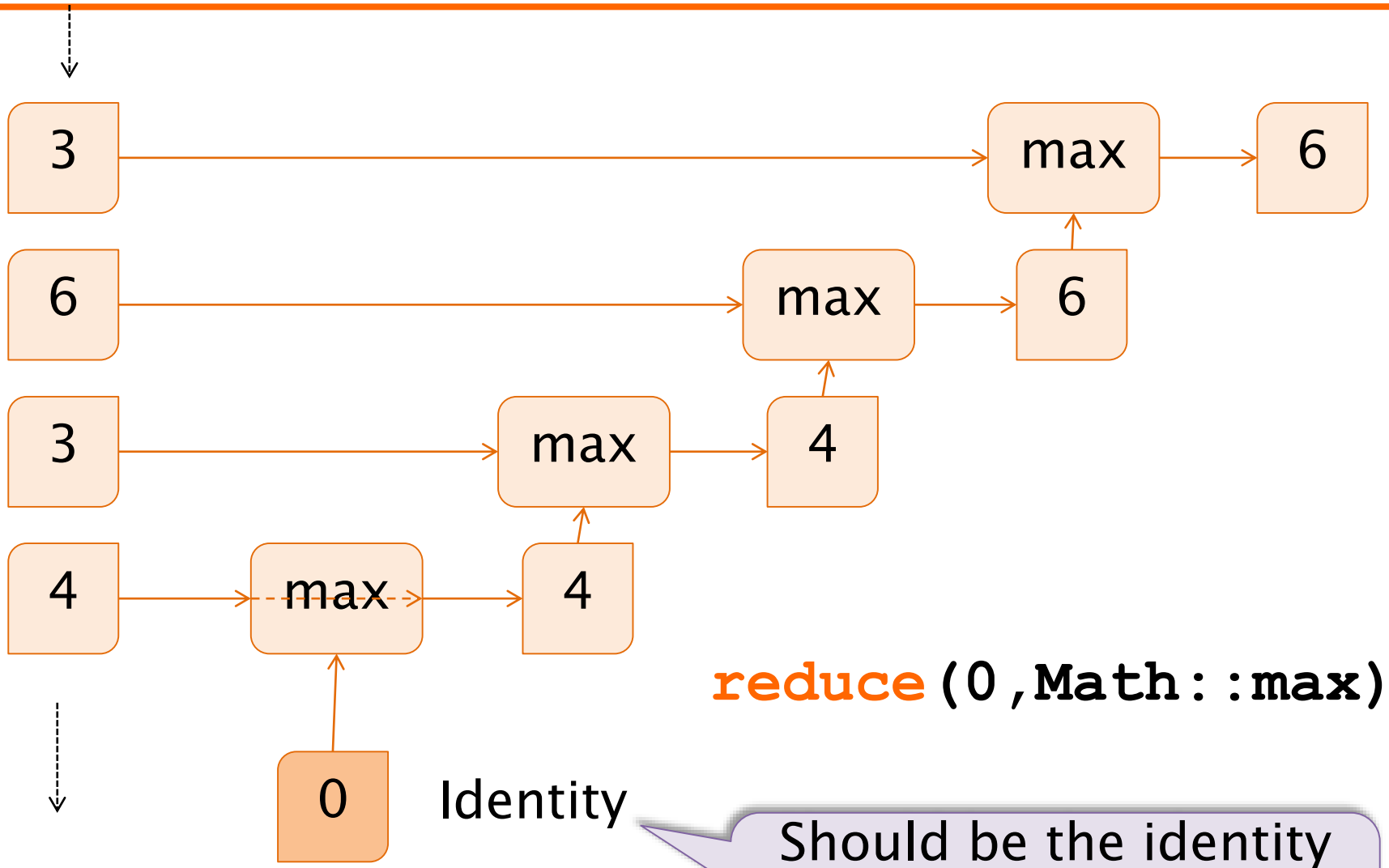
```
…
.forEach(System.out::println);
```

Reducing

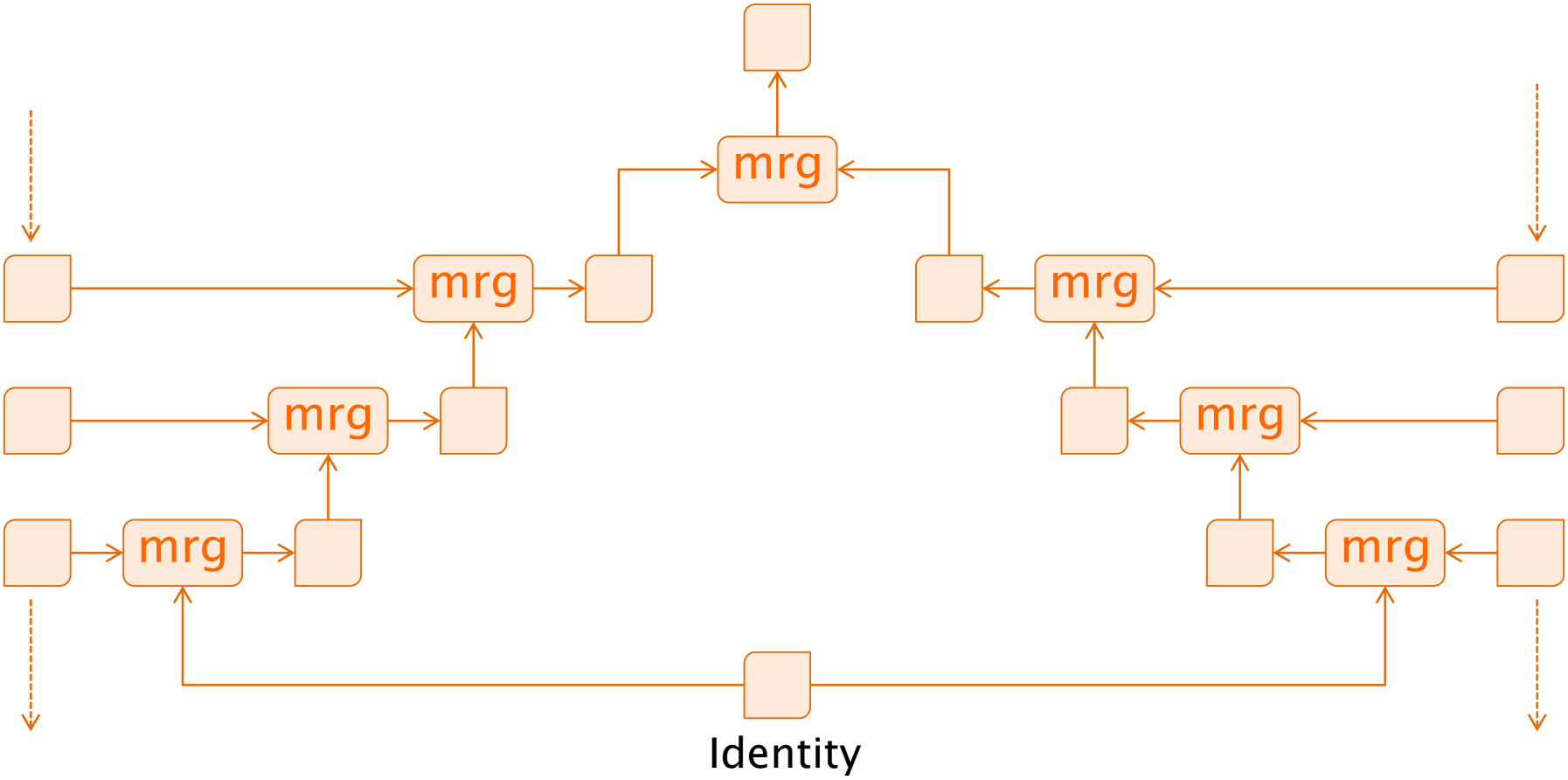
- **`T reduce(T identity, BinaryOperator<T> merge)`**
  - Reduces the elements of this stream, using the provided identity value and an associative accumulation function

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

Reducing


3
max
6
6
max
6
3
max
4
4
max
4
0
Identity
reduce(0,Math::max)
Should be the identity operand w.r.t. the merge operator

Parallelized reduce



Operation State

- Stateless operations
  - No internal storage is required
    - E.g. map, filter
- Stateful operations
  - Require internal storage, can be
  - Bounded
    - E.g. reduce, limit
  - Unbounded
    - E.g. sorted, collect

Numeric streams

- More efficient
  - No boxing and unboxing
- Provided for numeric types
  - **`DoubleStream`**
  - **`IntStream`**
  - **`LongStream`**
- Conversion method from Stream<T>
  - **`mapToDouble()`**
  - **`mapToInt()`**
  - **`mapToLong()`**

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 parallelizing)

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 ArrayList<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;});
```

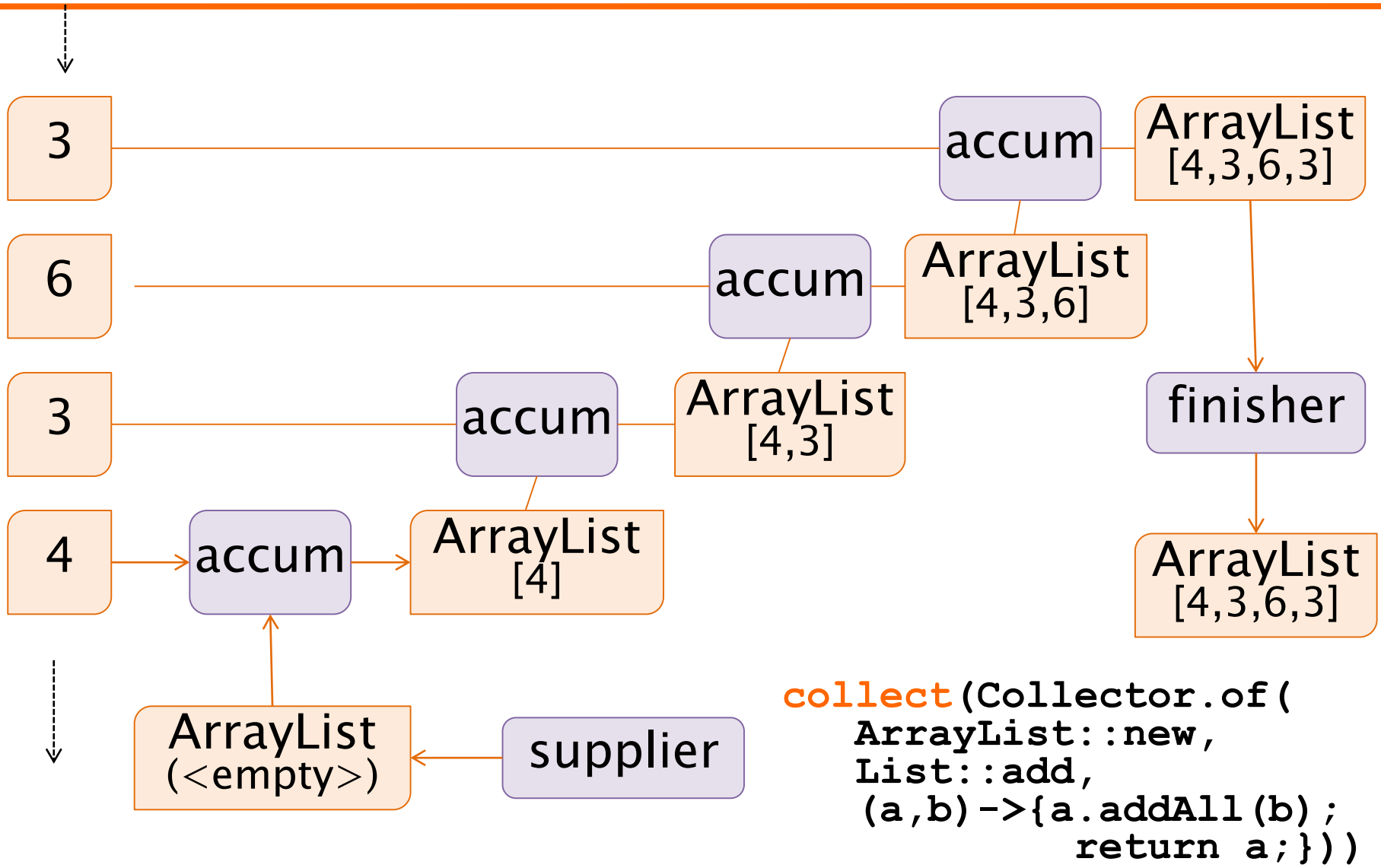
supplier

accumulator

combiner

Implicit finisher => identity transformation

Collector




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

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 of a stream of Integer
  - 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 int length;
   private int 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 w/ a specific operation
    - 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 May accept: 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> x)`	`Map<K, List<T>>`	Map according to the key extracted (`x`) and add to list. May accept: - Downstream Collector (nested) - Map supplier
`partitioningBy (Function<T, Boolean> p)`	`Map<Boolean, List<T>>`	Split according to partition function (`p`) and add to list May accept: - Downstream Collector (nested) - Map supplier

Grouping collectors

- Grouping by feature

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

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 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`)

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());
```

Examples (1)

```
List<Person> persons = new ArrayList<>();


persons.add(new Person("Aldo", Sex.MALE,
   LocalDate.of(1980, 10, 24), "aldo@polito.it"));
persons.add(new Person("Chiara", Sex.FEMALE,
   LocalDate.of(1975, 1, 2), "chiara@polito.it"));
persons.add(new Person("Enzo", Sex.MALE,
   LocalDate.of(1946, 3, 14), "enzo@polito.it"));
persons.add(new Person("Paolo", Sex.MALE,
   LocalDate.of(1953, 5, 7), "paolo@polito.it"));
persons.add(new Person("Elettra", Sex.FEMALE,
   LocalDate.of(2005, 12, 26), "elettra@polito.it"));
persons.add(new Person("Anna", Sex.FEMALE,
   LocalDate.of(1978, 8, 15), "anna@polito.it"));
```

Examples (2)

```
System.out.println("Ordered names of persons :");
persons
   .stream()
   .sorted((p1, p2)
                -> p1.getName().compareTo(p2.getName()))
   .forEach(e -> System.out.println(e.getName()));
```

```
Ordered names of persons :
Aldo
Anna
Chiara
Elettra
Enzo
Paolo
```

Examples (3)

```
System.out.println("Male persons :");
persons
   .stream()
   .filter(e -> e.getGender() == Sex.MALE)
   .forEach(e -> System.out.println(e.getName()));
```

```
Male persons :
Aldo
Enzo
Paolo
```

Examples (4)

```
System.out.println("Males between 14 and 53 :");
persons
   .stream()
   .filter(p -> p.getGender() == Sex.MALE
                && p.getAge() >= 14
                && p.getAge() <= 53)
   .map(p -> p.getEmailAddress())
   .forEach(email -> System.out.println(email));
```

```
Males between 14 and 53 :
aldo@polito.it
```

Examples (5)

```
double average =
    persons
      .stream()
      .filter(p -> p.getGender() == Sex.MALE)
      .mapToInt(p -> p.getAge())
      .average()
      .getAsDouble();
System.out.println("Average age of males : " +
                        average);
```

Average age of males : 57.33

Examples (6)

```
System.out.println("Male names : ");
List<String> maleNames =
   persons
       .stream()
       .filter(p -> p.getGender() == Sex.MALE)
       .map(p -> p.getName())
       .collect(Collectors.toList());
for(String s : maleNames)
        System.out.println(s);
```

```
Male names :
Aldo
Enzo
Paolo
```

Examples (7)

```
System.out.println("Persons grouped by gender :");
Map<Sex,List<Person>> personsBySex =
  persons
  .stream()
  .collect(Collectors.
                    groupingBy(Person::getGender));
System.out.println(personsBySex);
```

Persons grouped by gender:

{FEMALE=[Anna, Chiara, Elettra], MALE=[Aldo, Enzo, Paolo]}

Examples (8)

```
System.out.println("Males partitioned by age 54:");
Map<Boolean,List<Person>> malesBy54 =
    persons
      .stream()
      .filter(p -> p.getGender() == Sex.MALE)
      .collect(Collectors.
              partitioningBy(p -> p.getAge() <= 54));
System.out.println(malesBy54);
```

```
Males partitioned by age 54:
{false=[Enzo, Paolo], true=[Aldo]}
```

Examples (9)

```
System.out.println("Number of males older than 54:");
Long malesOlder54 =
  persons
      .stream()
      .filter(p -> p.getGender() == Sex.MALE
                  && p.getAge() >= 54)
      .collect(Collectors.counting());
System.out.println(malesOlder54);
```

```
Number of males older than 54:
2
```

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