

Programming in Java – Part 03 Streams



Paolo Vercesi

ESTECO SpA

© 2019 ESTECO SpA

Agenda



Streams

Working with streams

Specialized stream

Additional content - Parallel streams

© 2019 ESTECO SpA

Solution of exercises

© 2019 ESTECO SpA

Exercise

Write a class (or a set of classes) that given a string it produces a Term Frequency table.
Consider the option to provide a list of stop words, normalization, etc.
Provide an option to print the table in alphabetical order and by frequency.

"Term frequency (TF) means how often a term occurs in a document. In the context of natural language, terms correspond to words or phrases ..."



Term	Frequency
english	8
language	7
words	12

...

input	7
cactus	1
fireworks	3

© 2019 ESTECO SpA

The TermFrequency class

```
public class TF {

    private final Map<String, Integer> map;

    public TF(Map<String, Integer> map) {
        this.map = new TreeMap<>(map);
    }

    private void print(Map<String, Integer> map) {
        for (var entry : map.entrySet()) {
            System.out.println(entry);
        }
    }

    …
```

```
    …

    public void printAlphabetically() {
        print(map);
    }

    public void printByFrequency() {
        Comparator<String> c = new Comparator<>() {
            @Override
            public int compare(String o1, String o2) {
                int f1 = map.get(o1);
                int f2 = map.get(o2);
                return f1 == f2 ? o1.compareTo(o2) : f1-f2;
            }
        };
        Map<String, Integer> treeMap = new TreeMap<>(c);
        treeMap.putAll(map);
        print(treeMap);
    }
}
```



© 2019 ESTECO SpA

A TermFrequency builder

Write a class (or a set of classes) that given a string it produces a Term Frequency table

```
public class TFBuilder {

  private Tokenizer tokenizer;
  private Normalizer normalizer;
  private Filter filter;

  public void setTokenizer(Tokenizer tokenizer) {
     this.tokenizer = tokenizer;
  }

  public void setNormalizer(Normalizer normalizer) {
     this.normalizer = normalizer;
  }

  public void setFilter(Filter filter) {
    this.filter = filter;
  }
  …
```

```
  …
  TF build(String text) {
    Collection<String> tokens = tokenizer.tokenize(text);
    Map<String, Integer> map = new HashMap<>();
    for (String term : tokens) {
      term = normalizer.normalize(term);
      if (filter.accept(term)) {
        map.merge(term, 1, new BiFunction<>() {
          @Override
          public Integer apply(Integer v, Integer d) {
            return v + 1;
          }
        });
      }
    }
    return new TF(map);
  }
}
```



© 2019 ESTECO SpA

Tokenizer & C.

```
public interface Tokenizer {

    Collection<String> tokenize(String text);
}

public interface Normalizer {

    String normalize(String token);
}

public interface Filter {

    boolean accept(String token);
}
```



© 2019 ESTECO SpA

A usage example

```
public static void main(String[] args) {
    Set<String> stopWords = Set.of("a", "the", "an", "of");

    TFBuilder tfBuilder = new TFBuilder();
    tfBuilder.setTokenizer(new Tokenizer() {
        @Override
        public Collection<String> tokenize(String text) {
            return Arrays.asList(text.split("\\s"));
        }
    });
    tfBuilder.setNormalizer(new Normalizer() {
        @Override
        public String normalize(String token) {
            return token.replaceAll("\\.|,|", "").toLowerCase();
        }
    });
    tfBuilder.setFilter(new Filter() {
        @Override
        public boolean accept(String token) {
            return !stopWords.contains(token);
        }
    });
    TF tf = tfBuilder.build("...");
    tf.printAlphabetically();
    tf.printByFrequency();
}
```

© 2019 ESTECO SpA

Streams

© 2019 ESTECO SpA

Exercise

```
public record Dish(String name, boolean vegetarian, int calories, Type type) {

    public enum Type {MEAT, FISH, OTHER}

    @Override
    public String toString() {
        return name;
    }
}
```

```
Set<Dish> menu = Set.of(
        new Dish("pork", false, 800, Type.MEAT),
        new Dish("beef", false, 700, Type.MEAT),
        new Dish("chicken", false, 400, Type.MEAT),
        new Dish("french fries", true, 530, Type.OTHER),
        new Dish("rice", true, 350, Type.OTHER),
        new Dish("season fruit", true, 120, Type.OTHER),
        new Dish("pizza", true, 550, Type.OTHER),
        new Dish("prawns", false, 300, Type.FISH),
        new Dish("salmon", false, 450, Type.FISH)
);
```

Given a list of Dishes, create the list of dishes with less than 400 calories and sort them by the number of calories



© 2019 ESTECO SpA

The “imperative” solution - How

```
List<Dish> lowCaloricDishes = new ArrayList<>();

for (Dish dish : menu) {
    if (dish.calories() < 400) {
        lowCaloricDishes.add(dish);
    }
}

lowCaloricDishes.sort(Comparator.comparingInt(Dish::calories));

List<String> lowCaloricDishesName = new ArrayList<>();

for (Dish dish : lowCaloricDishes) {
    lowCaloricDishesName.add(dish.name());
}
```

Intermediate data

Explicit iterations

```
[season fruit, prawns, rice]
```

We provide the solution by mixing the logic to solve the problem with explicit iterations and intermediate data



© 2019 ESTECO SpA

The “declarative” solution - What

SELECT name FROM dish WHERE calories < 400 ORDER BY calories

```
List<String> lowCaloricDishesName = menu.stream()
                .filter(d -> d.calories() < 400)
                .sorted(Comparator.comparing(Dish::calories))
                .map(Dish::name)
                .toList();
```

```
[season fruit, prawns, rice]
```

The last operator decides the return type of the method chain, fluent interface

With streams, you make a continuous use of lambdas and method references, even if lambdas and method references are not part of that API!



© 2019 ESTECO SpA

Imperative vs declarative

Imperative you focus on writing
how to obtain what you want

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for (Dish dish : menu) {
    if (dish.calories() < 400) {
        lowCaloricDishes.add(dish);
    }
}

lowCaloricDishes.sort(Comparator.comparingInt(Dish::calories));

List<String> lowCaloricDishesName = new ArrayList<>();
for (Dish dish : lowCaloricDishes) {
    lowCaloricDishesName.add(dish.name());
}
```

Declarative you focus on declaring
what you want

```
List<String> lowCaloricDishesName = menu.stream()
    .filter(d -> d.calories() < 400)
    .sorted(Comparator.comparing(Dish::calories))
    .map(Dish::name)
    .toList();
```

This is not coming for free; the declarative approach needs higher level abstractions provided by the Stream API that introduce an internal DSL (Domain Specific Language) on top of Java

In other areas of the Java language, you don't have such kind of abstractions and maybe you can create them by yourself



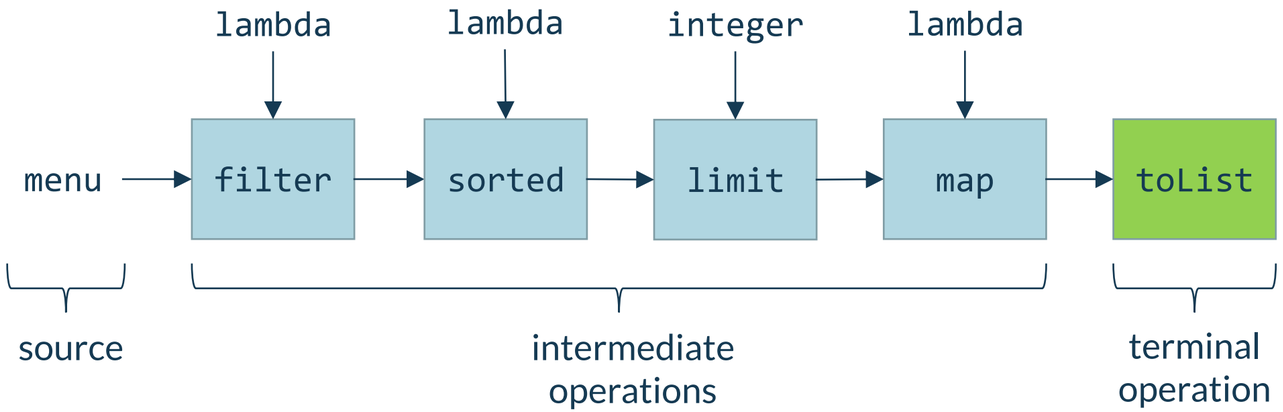
© 2019 ESTECO SpA

Stream operations

Streams are used to perform computations made of operations composed into stream pipelines

A stream pipeline consists of

- a source, that might be an array, a collection, a generator function, an I/O channel, …
- a chain of zero or more intermediate operations, transforming a stream into another stream
- a terminal operation, producing a result or a side-effect, e.g., count() or forEach(Consumer)




```
List<String> firstFiveLowCalories = menu.stream()
  .filter(d -> d.calories() < 400)
  .sorted(Comparator.comparing(Dish::calories))
  .limit(5)
  .map(Dish::name)
  .toList();
```

The result type depends on the terminal operation, only

Streams are lazy, computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed

Streams cannot be reused; you can pipe just one intermediate or final operation

© 2019 ESTECO SpA

What's a Stream

A sequence of elements from a source,
that supports data-processing operations



© 2019 ESTECO SpA

Some intermediate & terminal operators

Operation	Type	Return type	Argument of the operation	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`limit`	Intermediate	`Stream<T>`	`int`	
`skip`	Intermediate	`Stream<T>`	`int`	
`sorted`	Intermediate	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`distinct`	Intermediate	`Stream<T>`		
`forEach`	Terminal	`void`	`Consumer<T>`	`T -> void`
`count`	Terminal	`long`		
`toList`	Terminal	`List<T>`		
`collect`	Terminal	(generic)	`Collector<T, A, R>`	not a functional interface



© 2019 ESTECO SpA



Working with streams

© 2019 ESTECO SpA

Intermediate & terminal operations

Intermediate operations

Filtering
- selecting elements with predicate
- selecting unique elements

Sorting

Slicing
- slicing using a predicate
- truncating a stream
- skipping a stream

Mapping
- transform each element
- flattening streams

Peeking

Terminal operations

Counting elements

Consuming

Matching
- anyMatch, allMatch, noneMatch

Finding
- findAny, findFirst

Reducing

Collecting

Grouping

Partitioning



© 2019 ESTECO SpA

Intermediate operations

© 2019 ESTECO SpA

Filtering with filter() and distinct()

Selecting elements by using a predicate

```
List<Dish> vegetarianMenu = menu.stream()
        .filter(Dish::vegetarian)
        .toList();
```

Predicate taking a `Dish` and returning a boolean

Selecting 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);
```

Remove duplicates accordingly to `hashcode()` and `equals()`



© 2019 ESTECO SpA

Sorting

```
List<String> cities = List.of("Trieste", "Gorizia", "Udine", "Pordenone");
cities.stream()
        .sorted()
        .forEach(IO::println);
```

Sorting based on the natural order of a `Comparable` class

Sorting based on a custom `Comparator`

```
List<Dish> menu = Dish.menu;
menu.stream()
        .sorted(Comparator.comparing(Dish::calories))
        .forEach(IO::println);
```



© 2019 ESTECO SpA

Conditional slicing with takeWhile() and dropWhile()

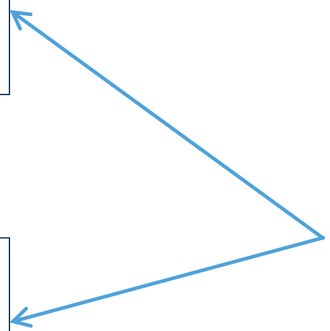
```
List<Dish> specialMenu = List.of(
    new Dish("seasonal fruit", true, 120, Type.OTHER),
    new Dish("prawns", false, 300, Type.FISH),
    new Dish("rice", true, 350, Type.OTHER),
    new Dish("chicken", false, 400, Type.MEAT),
    new Dish("french fries", true, 530, Type.OTHER));
```

Select dishes with less than 320 calories

```
List<Dish> filteredMenu = specialMenu.stream()
        .takeWhile(dish -> dish.calories() < 320)
        .toList();
```

Select dishes with more than 320 calories

```
List<Dish> filteredMenu = specialMenu.stream()
        .dropWhile(dish -> dish.calories() < 320)
        .toList();
```



Assume the stream is ordered by calories ascending

© 2019 ESTECO SpA

Unconditional slicing with limit() and skip()

```
List<Dish> specialMenu = Arrays.asList(
    new Dish("seasonal fruit", true, 120, Type.OTHER),
    new Dish("prawns", false, 300, Type.FISH),
    new Dish("rice", true, 350, Type.OTHER),
    new Dish("chicken", false, 400, Type.MEAT),
    new Dish("french fries", true, 530, Type.OTHER));
```

Select the first three dishes with less than 500 calories

```
List<Dish> filteredMenu = specialMenu.stream()
        .filter(dish -> dish.calories() < 500)
        .limit(3)
        .toList();
```

Stream size is limited to 3

Skip the first two dishes with more than 300 calories

```
List<Dish> filteredMenu = specialMenu.stream()
        .filter(dish -> dish.calories() > 300)
        .skip(2)
        .toList();
```

Skip the first two elements



© 2019 ESTECO SpA

Mapping

```
List<String> dishNames = menu.stream()
        .map(Dish::name)
        .toList();
```

Map each dish to its name

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

```
List<String> dishNames = menu.stream()
        .map(Dish::name)
        .map(String::toUpperCase)
        .toList();
```

Map each dish to its uppercase name

```
[PORK, BEEF, CHICKEN, FRENCH FRIES, RICE, SEASON FRUIT, PIZZA, PRAWNS, SALMON]
```

```
List<Integer> dishNameLengths = menu.stream()
        .map(Dish::name)
        .map(String::length)
        .toList();
```

Map each dish to its name length

```
[4, 4, 7, 12, 4, 12, 5, 6, 6]
```



© 2019 ESTECO SpA

Flattening streams (working with streams of streams)

The `flatMap` operation is used when you need to transform the element of a stream into new streams and you want to flatten these new streams into a single stream

Example create a single stream from two streams

```
Stream.of(Stream.of(1, 2), Stream.of(3, 4))
    .flatMap(x -> x)
    .forEachOrdered(IO::println);
```

Can be replaced by `Function.identity`

Example create a single stream from two lists

```
Stream.of(List.of(1, 2), List.of(3, 4))
    .flatMap(x -> x.stream())
    .forEachOrdered(IO::println);
```

Usually, you use `flatMap` with a function that transform the current type into a `Stream`



© 2019 ESTECO SpA

Flattening streams

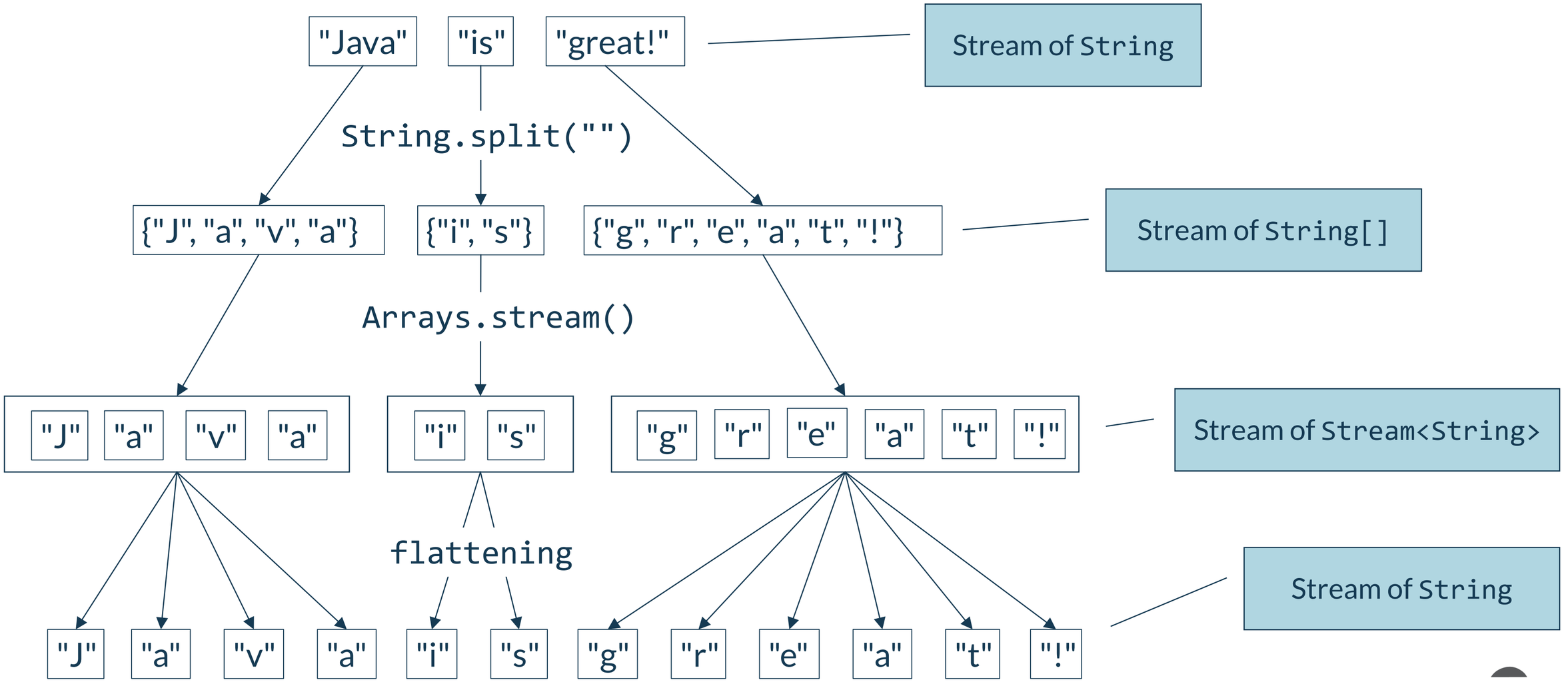
Example: we have a list of strings and we want to create a list of unique characters contained in these strings

```
List<String> strings = List.of("Java", "is", "great!");
List<String> distinct = strings.stream()
        .map(s -> s.split(""))
        .flatMap(Arrays::stream)
        .distinct()
        .toList();
IO.println(distinct);
```

```
[J, a, v, i, s, g, r, e, t, !]
```

© 2019 ESTECO SpA

Flattening streams



© 2019 ESTECO SpA

Cartesian product

Exercise: Given two lists of numbers, how would you return all pairs of numbers?
For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)].

```
List<Integer> list1 = List.of(1, 2, 3); List<Integer> list2 = List.of(3, 4);
```

We start by traversing list1, then for each element of list1 we must traverse list2 and create a list with the two elements coming from the two streams

```
list1.stream().map(i1 -> list2.stream().map(i2 -> List.of(i1, i2)))
```

Stream<Stream<List<Integer>>>

We mapped i1 to a stream of couples of i1 with elements from list2, now we must flat this map

```
list1.stream()
        .map(i1 -> list2.stream().map(i2 -> List.of(i1, i2)))
        .flatMap(Function.identity())
        .toList()
```

Stream<List<Integer>>

or

```
list1.stream()
        .flatMap(i1 -> list2.stream().map(i2 -> List.of(i1, i2))).toList()
```



© 2019 ESTECO SpA

peek()

```
Stream<T> peek(Consumer<? super T> action)
```

Returns the same stream after consuming the item,
it doesn't change the stream

```
Optional<Integer> value = Stream.of(1, 2, 3, 4)
        .peek(x -> IO.println("processing: " + x))
        .filter(n -> n % 2 == 0)
        .peek(y -> IO.println("accepted " + y))
        .findFirst();
```

only for debugging!



© 2019 ESTECO SpA

Terminal operations

© 2019 ESTECO SpA

Counting and consuming elements

Counting elements

```
long count = Stream.of(1, 2, 3, 4)
        .filter(n -> n % 2 == 0)
        .count();
```

```
2
```

Consuming elements with forEach

```
Stream.of(1, 2, 3, 4)
        .filter(n -> n % 2 == 0)
        .forEach(IO::println);
```

```
2
4
```



© 2019 ESTECO SpA

Matching

Is there at least one vegetarian dish?

```
boolean atLeastOneVegetarian = menu.stream()
    .anyMatch(Dish::vegetarian);
```

Is the menu vegetarian only?

```
boolean vegetarianOnly = menu.stream()
    .allMatch(Dish::vegetarian);
```

Are all dishes free from meat and fish?

```
boolean noMeatNorFish = menu.stream()
    .noneMatch(d -> d.type() == Type.MEAT || d.type() == Type.FISH);
```



© 2019 ESTECO SpA

Finding with findAny()

Find any vegetarian dish

```
Optional<Dish> dish = menu.stream()
        .filter(Dish::vegetarian)
        .findAny();
```

The operation `findAny()` returns an `Optional` because it might not find any match

Find a vegetarian dish with meat

```
Optional<Dish> dish = menu.stream()
        .filter(Dish::vegetarian)
        .filter(d -> d.type() == Type.MEAT)
        .findAny();
```

The `Optional<T>` class to model the fact that a method call can return something that does not exists



© 2019 ESTECO SpA

Working with Optional

```
Optional<String> optionalName = menu.stream()
        .filter(Dish::vegetarian)
        .map(Dish::name)
        .findAny();

if (optionalName.isPresent()) {
    IO.println(optionalName.get());
}
```

Imperative style, get the value out of the Optional by using `isPresent()` and `get()` and do some if-programming

Never use `get()` unless you have checked if the value `isPresent()`

```
menu.stream()
        .filter(Dish::vegetarian)
        .map(Dish::name)
        .findAny()
        .ifPresent(name -> IO.println(name));
```

Declarative (functional) style, ask the optional to do something by using `ifPresent(Consumer<T> action)`

The API of `Optional` contains many other methods for functional style programming

```
public T get() throws NoSuchElementException

public boolean isPresent()

public void ifPresent(Consumer<? super T> action)
```



© 2019 ESTECO SpA

Digression - Using `Optional` as return value

If you have a method that can optionally return null, you can consider to return an `Optional`

`Optional` is a final class with a private constructor, to build an optional we use one of three provided factory methods

```
public static <T> Optional<T> ofNullable(T value)

public static <T> Optional<T> empty()

public static <T> Optional<T> of(T value) throws NullPointerException
```

There is no way to wrap a null value into an `Optional` object, an `Optional` can be empty but it cannot contain `null`

If a method declares to return `Optional`, then you should never return `null` instead of an `Optional`



© 2019 ESTECO SpA

Finding with `findFirst()`

Find any vegetarian dish

```
Optional<Dish> dish = menu.stream()
        .filter(Dish::vegetarian)
        .findAny();
```

Find the first vegetarian dish

```
Optional<Dish> dish = menu.stream()
        .filter(Dish::vegetarian)
        .findFirst();
```

Why we have both `findAny()` and `findFirst()`?

Some streams are said to be ordered because they have a defined encounter order

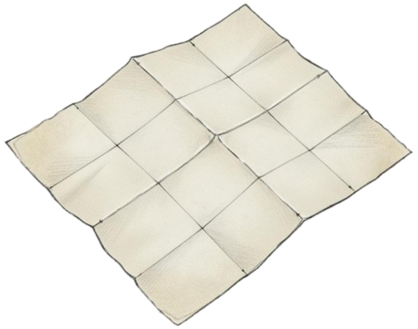
E.g., streams obtained from `List` or from a `sort()` intermediate operation do have an encounter order
When you process an ordered stream in parallel, `findFirst()` is more constraining than `findAny()`

ordered ⇔ encounter order ⇔ subject to ordering constraint



© 2019 ESTECO SpA

Reducing (folding)



Reducing is about combining the elements of a stream into a single value

To answer questions like

1. What's the sum of all the calories in the menu?
    - We need to sum up all the calories' values
2. What's the highest calories dish in the menu?
    - We need to compare all the calories values

So far, we have seen terminal operations that do not combine the elements of the stream, e.g., matchAll, matchNone, findAny, findFirst, etc.

We need to introduce new terminal operations to reduce the stream into a single value by combining all the elements of a stream to compute that single value

In the functional programming jargon, we need a fold operation

To perform a reduction, we need an associative and stateless function that combines two values into a single one

A bifunction like (a,b) -> c
where a is the previous value, b is the current value and c is the result
If a, b, and c are all of the same type we need just a binary operator (e.g., +)

© 2019 ESTECO SpA

Reducing API

The `Stream` interface contains three methods to reduce the stream

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity, BinaryOperator<T> accumulator)

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
```



© 2019 ESTECO SpA

Reducing example 1/3

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Find the highest calories dish

```
Optional<Integer> dish = menu.stream()
    .map(Dish::calories)
    .reduce((a, b) ->  a > b ? a : b);
```

Can we use a method reference?

```
Optional<Integer> dish = menu.stream()
    .map(Dish::calories)
    .reduce(Integer::max);
```

Sum up all the calories in the menu

```
Optional<Integer> sum = menu.stream()
    .map(Dish::calories)
    .reduce((a, b) -> a + b);
```

Can we use a method reference?

```
Optional<Integer> sum = menu.stream()
    .map(Dish::calories)
    .reduce(Integer::sum);
```

Why Optional?



© 2019 ESTECO SpA

Reducing example 2/3

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

The `identity` value is used as initial value for the reduction process, so no optional is needed

The `identity` value is an identity for the accumulator function, so that `accumulator.apply(indentity, identity)` evaluates as the identity value

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



© 2019 ESTECO SpA

Reducing example 3/3

```
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
```

The `combiner` is used to combine the values coming from parallel computations

The `identity` value is an identity for the `combiner` function, so `combiner.apply(indentity, identity)` returns the `identity` value

The `accumulator` takes the accumulated value, the current value and returns the new accumulated value

```
Integer calories = menu.stream()
    .reduce(0, (a, c) -> a + c.calories(), Integer::sum);
```



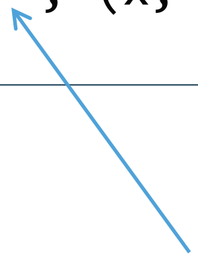
© 2019 ESTECO SpA

One more reducing example

We want to concatenate strings by separating them by a comma

```
List<String> cities = Arrays.asList("Trieste", "Gorizia", "Udine", "Pordenone");
String reduced1 = cities.stream().reduce("", (x, y) -> x + ", " + y);
IO.println(reduced1);
```

```
, Trieste, Gorizia, Udine, Pordenone
```



The provided indentity IS NOT an indentity for the accumulator function

```
Optional<String> reduced2 = cities.stream().reduce((x, y) -> x + ", " + y);
reduced2.ifPresent(IO::println);
```

```
Trieste, Gorizia, Udine, Pordenone
```

© 2019 ESTECO SpA

Collecting

Collecting is a reduction operation that accumulates elements into a “mutable” container

```
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
```

Provides the mutable container of type R

Accumulates the elements of type T into the mutable container of type R

Combines the elements from intermediate container of type R

```
List<String> dishNames = menu.stream()
        .map(Dish::name)
        .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

```
List<String> dishNames = menu.stream()
        .collect(ArrayList::new, (a, d) -> a.add(d.name()), ArrayList::addAll);
```



© 2019 ESTECO SpA

Collecting using a Collector

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

```
public interface Collector<T, A, R> {

    Supplier<A> supplier();


    BiConsumer<A, T> accumulator();


    BinaryOperator<A> combiner();


    Function<A, R> finisher();

    …
}
```

A function that creates and returns a new mutable result container

A function that folds a value into a mutable result container

A function that accepts two partial results and merges them, the combiner function may fold state from one argument into the other and return that, or may return a new result container

A function which transforms the intermediate result to the final result, e.g., to return an unmodifiable collection

The `Collectors` utility class provides lots of static methods to conveniently create an instance of the most common collectors



© 2019 ESTECO SpA

Collecting using `Collectors`, `toList()`, and `toArray()`

```
List<String> dishNames = menu.stream()
        .map(Dish::name)
        .collect(Collectors.toList());

List<String> dishNames = menu.stream()
        .map(Dish::name)
        .collect(Collectors.toUnmodifiableList());

Map<String, Dish> nameDishMap = menu.stream()
        .collect(Collectors.toMap(Dish::name, Function.identity()));
```

```
List<String> dishNames = menu.stream()
        .map(Dish::name)
        .toList();

String[] dishNames = menu.stream()
        .map(Dish::name)
        .toArray();
```



© 2019 ESTECO SpA

Joining Strings

```
String result = menu.stream()
        .map(Dish::name)
        .map(name -> name.split(""))
        .flatMap(Arrays::stream)
        .distinct()
        .collect(Collectors.joining(", "));

IO.println(result);
```

```
p, o, r, k, b, e, f, c, h, i, n,  , s, a, u, t, z, w, l, m
```

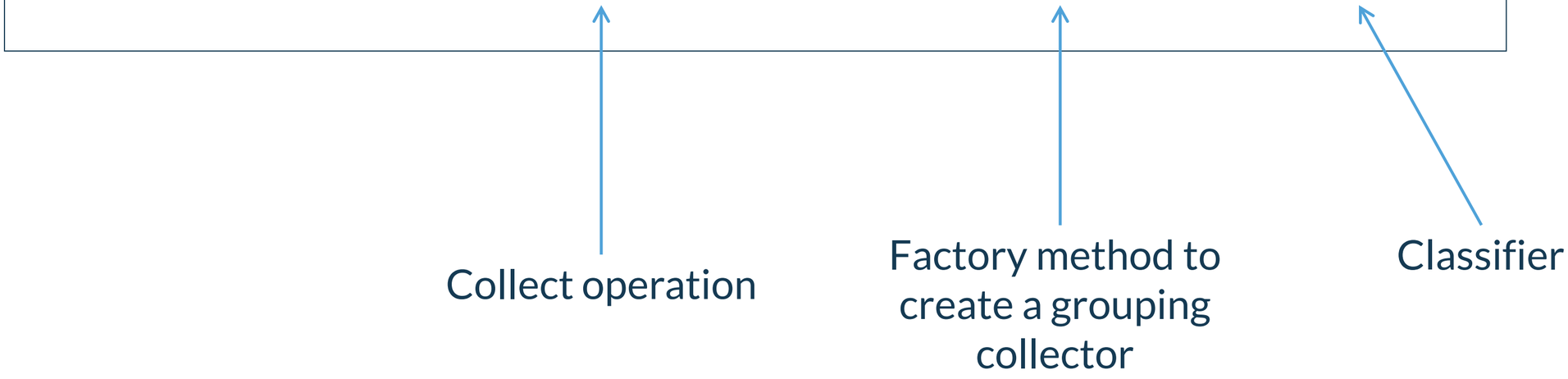


© 2019 ESTECO SpA

Grouping

In the `collect()` operation we can specify a grouping operation to classify the element of the stream in different groups

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



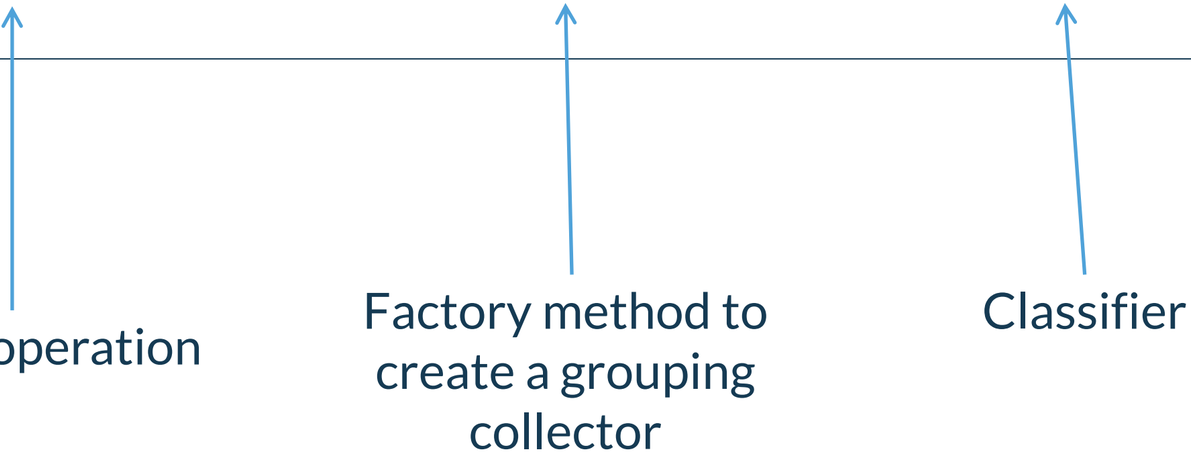

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit,
pizza], MEAT=[pork, beef, chicken]}
```

© 2019 ESTECO SpA

Partitioning

Partitioning is a special case of grouping, that have a predicate called a partitioning function as a classification function

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




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

© 2019 ESTECO SpA

Specialized streams and stream creation

© 2019 ESTECO SpA

Numeric streams

The Streams API provides primitive stream specializations that support specialized methods to work with streams of numbers, to make common numeric reductions more efficient

These specialized streams are `IntStream`, `DoubleStream`, and `LongStream`

We can move from object streams to primitive streams by using on of the `mapToInt`/`Double`/`Long` methods

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);

OptionalInt maxCalories = intStream.max();
```

We can move from specialized primitive streams to `Object` streams by using the `boxed` method

```
Stream<Integer> integerStream = intStream.boxed();
```

or by using the `mapToObj` method

```
intStream.mapToObj(x -> new int[] {x, x*2})
```



© 2019 ESTECO SpA

Numeric streams reductions

OptionalDouble	average()
OptionalInt/OptionalDouble/OptionalLong	max()
OptionalInt/OptionalDouble/OptionalLong	min()
int/double/long	sum()
IntSummaryStatistics/ DoubleSummaryStatistics/ LongSummaryStatistics	summaryStatistics()

```
IntStream stream = IntStream.of(1, 2, 3, 5, 7, 11, 13, 17, 19);
IntSummaryStatistics statistics = stream.summaryStatistics();
IO.println(statistics);
```

```
IntSummaryStatistics{count=9, sum=78, min=1, average=8,666667, max=19}
```

© 2019 ESTECO SpA

Building streams from arrays and collections

Methods in Collection interface

```
default Stream<E> parallelStream()
default Stream<E> stream()
```

```
List.of("SDM", 2022, 2023).stream();
```

Returns a stream of `Object`

Methods in Arrays class

```
static DoubleStream stream(double[] array)
static DoubleStream stream(double[] array, int startInclusive, int endExclusive)
static IntStream stream(int[] array)
static IntStream stream(int[] array, int startInclusive, int endExclusive)
static LongStream stream(long[] array)
static LongStream stream(long[] array, int startInclusive, int endExclusive)
static <T> Stream<T> stream(T[] array)
static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)
```

```
int[] numbers = {2, 3, 5, 7, 11, 13};
Arrays.stream(numbers);
```

Returns an `IntStream`

© 2019 ESTECO SpA

Building streams by enumerating the elements

`Stream.empty()`
`Stream.of(T …)`
`Stream.ofNullable(T t)`
	`Stream.of("Java", "is", "great!");`
	`Stream.of(null, new Object(), List.of("Java"));`
	`IntStream.of(2, 3, 5, 7, 11);`
	`DoubleStream.of(3.14, 6.28, 9.42);`

`Stream.builder()`	`Builder<String> b = Stream.builder();`
	`b.add("Java");`
	`b.add("is");`
	`b.add("great!");`
	`Stream<String> s = b.build();`

`Stream.concat(`
`  Stream<? extends T> a,`
`  Stream<? extends T> b)`
	`Stream<String> a = Stream.of("Java", "is", "great!");`
	`Stream<String> b = Stream.of("Hello", "World!");`
	`Stream<String> c = Stream.concat(a, b);`

`Int/LongStream.range(`
`  startInclusive,`
`  endExclusive)`
`Int/LongStream.rangeClosed(`
`  startInclusive,`
`  endInclusive)`
	`IntStream.range(0, 100);`
	`[0, 1, 2, …, 99]`
	`IntStream.rangeClosed(0, 99);`
	`[0, 1, 2, …, 99]`



© 2019 ESTECO SpA

Building infinite streams

Java provides three methods to generate infinite streams

```
static <T> Stream<T> generate(Supplier<? extends T> s)
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)
```

This part of the stream API is used to generate data rather than to process data

```
DoubleStream randomGenerator = DoubleStream.generate(Math::random);
double[] randomNumbers = randomGenerator
        .limit(100)
        .toArray();
```

We can limit the generated data by using `limit(int)` or `takeWhile(Predicate)`

There are no other ways to stop the stream, but throwing an unchecked exception



© 2019 ESTECO SpA

Fibonacci

```
Stream.iterate(new int[] {0, 1}, a -> new int[] { a[1], a[0]+ a[1]})
        .limit(10)
        .map(a -> a[0])
        .forEach(IO::println);
```

Generates the first ten Fibonacci numbers

```
Stream.iterate(new int[] {0, 1}, a -> new int[] { a[1], a[0]+ a[1]})
        .map(a -> a[0])
        .takeWhile(a -> a < 100)
        .forEach(IO::println);
```

Generates Fibonacci numbers smaller than 100

```
Stream.iterate(new int[] {0, 1}, a -> a[0] < 100, a -> new int[] { a[1], a[0]+ a[1]})
        .map(a -> a[0])
        .forEach(IO::println);
```

Exercise: rewrite the iteration by using a for loop

In principle, you can replace any for loop with a stream iteration
(if you don't mutate variables outside the for loop)

Is there any benefit in using streams over for/while/foreach loops?



© 2019 ESTECO SpA

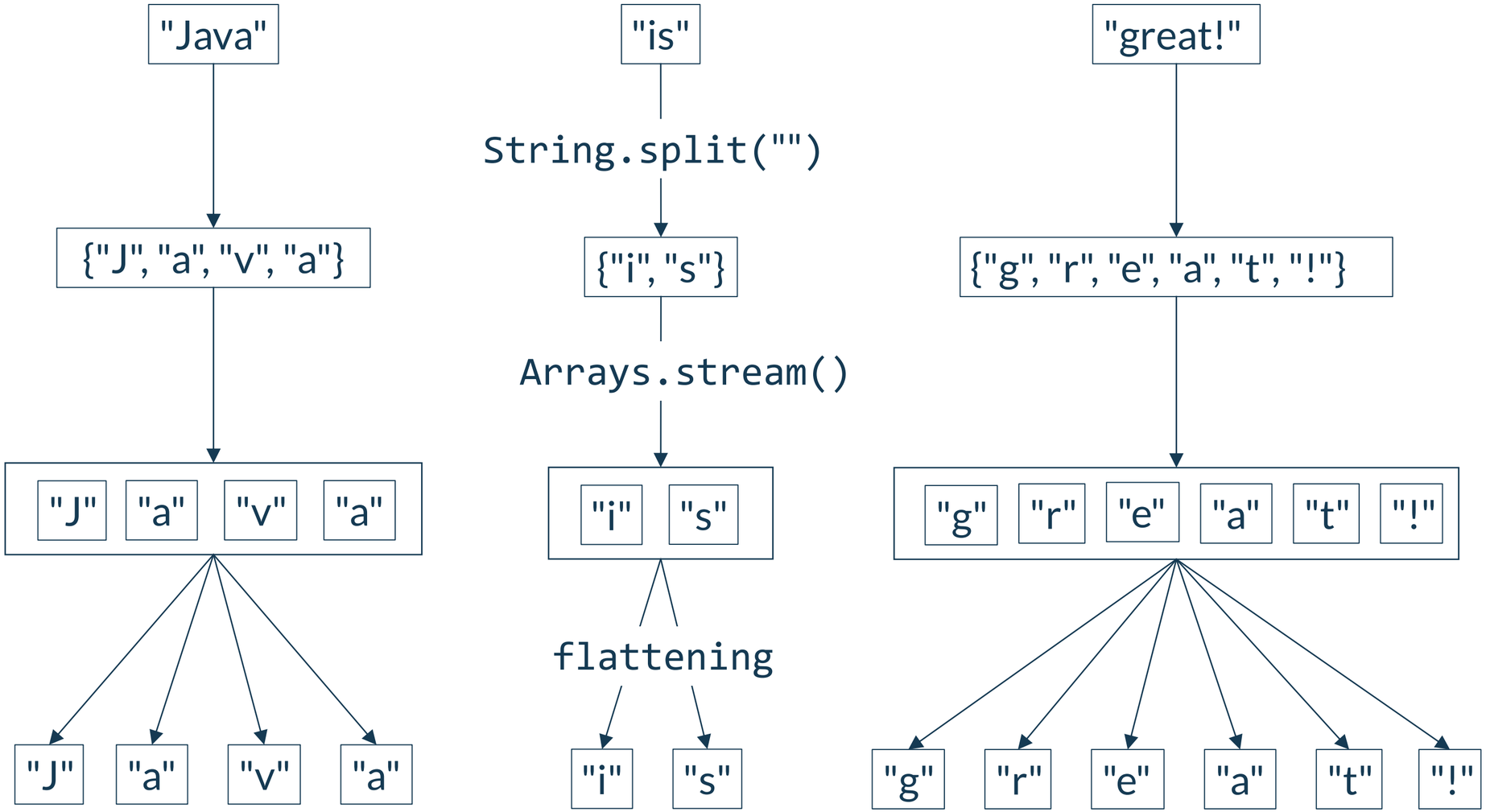


Additional content - Parallel streams



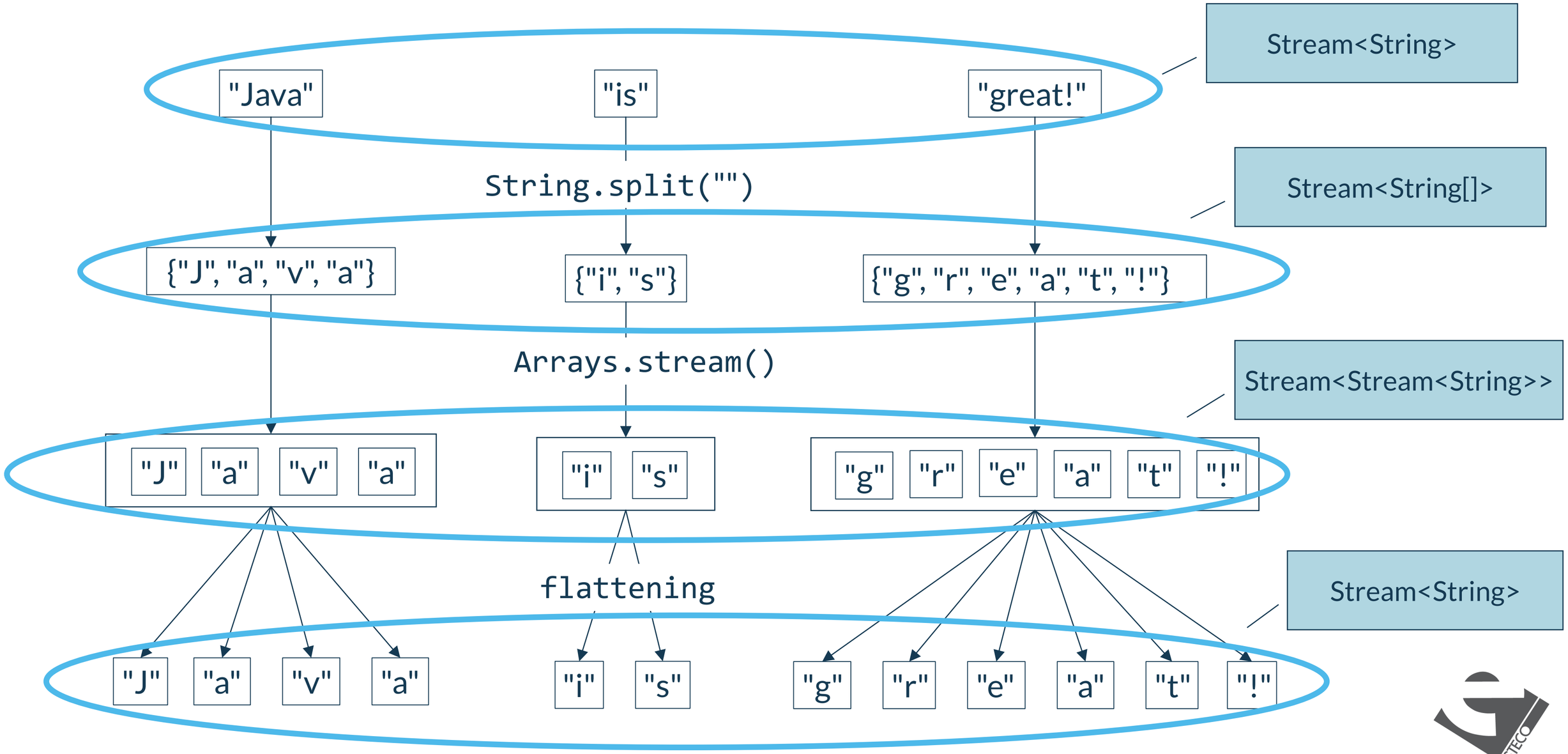
© 2019 ESTECO SpA

Parallel data processing



© 2019 ESTECO SpA

Parallel data processing



© 2019 ESTECO SpA

Parallel execution of a Stream

```
List<String> strings = List.of("Java", "is", "great!");
List<String> distinct = strings.stream()
        .map(s -> s.split(""))
        .parallel()
        .flatMap(Arrays::stream)
        .distinct()
        .toList();
IO.println(distinct);
```

© 2019 ESTECO SpA

Sequential and parallel stream processing

A stream has just one execution mode that can be parallel or sequential

```
List<String> strings = List.of("Java", "is", "great!");
List<String> distinct = strings.stream()
        .map(s -> s.split(""))
        .parallel()
        .flatMap(Arrays::stream)
        .sequential()
        .distinct()
        .toList();
IO.println(distinct);
```

The position of the parallel() or sequential() method in the pipeline is irrelevant

It is not possible to specify which operations must be performed in parallel and which sequentially

If there are more parallel() and sequential() method invocations in the pipeline, the last one sets the execution mode



© 2019 ESTECO SpA

Ordered and unordered streams

Some stream sources are naturally ordered and they define an encounter order (ordering constraint) for the elements of the stream, e.g., arrays, lists, etc.

Some operations process the stream elements by following that encounter order, e.g., `forEachOrdered()`, `findFirst()`

Processing the stream under the ordering constraint imposes a significant penalty for the parallel processing, this penalty is not relevant for sequential processing

It possible to reset the stream to an unordered state by invoking `unordered()`

```
List<String> strings = List.of("Java", "is", "great!");
List<String> distinct = strings.stream()
        .map(s -> s.split(""))
        .flatMap(Arrays::stream)
        .parallel()
        .unordered()
        .distinct()
        .toList();
IO.println(distinct);
```

The ordering constraint is set also by the `sorted()` operation on otherwise unordered streams



© 2019 ESTECO SpA

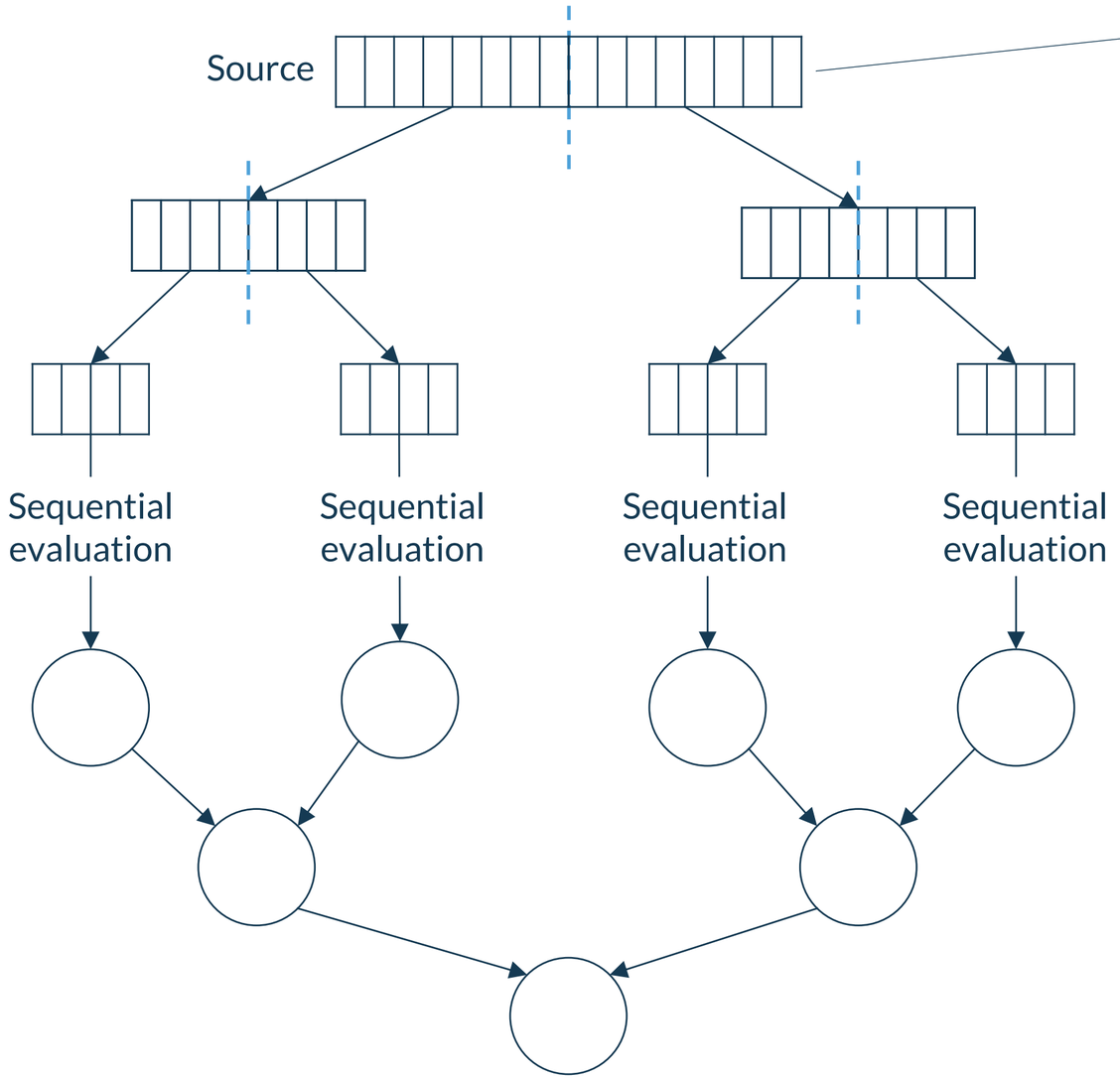
Using parallel and sequential effectively

- It is not possible to give a rule on when to use parallel or not
  - size of data set, cost of the processing
  - hardware, operating system, JVM settings, e.g., memory assigned to the JVM
- Always compare the performance of sequential and parallel with an appropriate benchmark
- Try to reduce the boxing/unboxing operations, use primitive stream when possible
- Operations which rely on ordering don't perform well on ordered streams in parallel
  - use alternatives findAny instead of findFirst
  - make the stream unordered
- Given N the size of the data set, Q the approximate cost of processing one element, and N*Q the total cost of the computation
  - higher Q can imply good performances in parallel
  - low N can imply bad performances in parallel
- Operations that are blocking on asynchronous calls, e.g., I/O operations, can take benefit from a different parallelization framework
- Consider the cost of the merging step of the terminal operation
- The source data stream can impact the effectiveness of parallelization, that is based on the fork/join framework



© 2019 ESTECO SpA

Fork/join and parallel streams



Stream parallelization starts to operate at the level of the data source

Note: in this context the terms fork and join refers to forking and joining both data and processes

© 2019 ESTECO SpA

The “illusion” of abstraction

You need to know the origin of your stream a.k.a. how it was created

Source	Decomposability
`ArrayList`	Excellent
`IntStream.range`	Excellent
`HashSet`	Good
`TreeSet`	Good
`LinkedList`	Poor
`Stream.iterate`	Poor

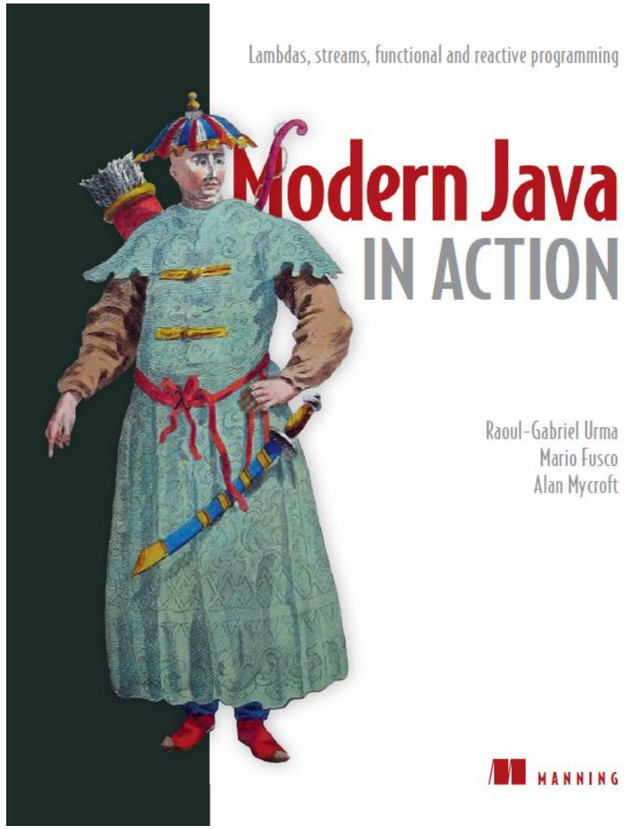


© 2019 ESTECO SpA

References

© 2019 ESTECO SpA

To know more



Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft

Modern Java in Action

© 2019 ESTECO SpA

Self assessment

© 2019 ESTECO SpA

Quiz 1: Streams

1. **Which category of stream operation produces a new stream as its output and is typically chained together to form the data transformation logic?**
   A. Intermediate
   B. Terminal
   C. Source
   D. Collecting
2. **Computation on the source data is only performed when the terminal operation is initiated. What term describes this fundamental behavior of stream execution?**
   A. Eager
   B. Evolving
   C. Stateful
   D. Lazy



© 2019 ESTECO SpA

Quiz 2: Working with streams

1. **For the conditional slicing methods, `takeWhile()` and `dropWhile()`, to work correctly, what state must the elements of the stream be in?**
   A. Distinct
   B. Unique
   C. Sorted
   D. Unlimited
2. **Which operation is used to flatten a stream of streams (like a `Stream<List<T>>`) into a single, merged `Stream<T>`?**
   A. `Filter`
   B. `Map`
   C. `FlatMap`
   D. `Collect`



© 2019 ESTECO SpA

Quiz 2: Working with streams

3. **To guarantee a correct parallel reduce operation, the combining function must have which mathematical property?**
   A. Commutative
   B. Idempotent
   C. Associative
   D. Distributive

4. **What is the fundamental data structure returned by the `Collectors.groupingBy()` terminal operation?**
   A. `Set`
   B. `List`
   C. `Queue`
   D. `Map`



© 2019 ESTECO SpA

Quiz 2: Working with streams

5. **Which discouraged `Optional` method will throw a `NoSuchElementException` if the value is absent?**
   A. `orElse()`
   B. `get()`
   C. `ifPresent()`
   D. `map()`

© 2019 ESTECO SpA

Quiz 3: Specialized streams and stream creation

1. **Using primitive streams (`IntStream`) instead of object streams (`Stream<Integer>`) primarily avoids the performance cost of what process?**
   A. Serialization
   B. Autoboxing
   C. Garbage Collection
   D. Optimization
2. **Which method of `Stream` uses a `Supplier` to produce new elements independently, making it ideal for random values?**
   A. `iterate()`
   B. `concat()`
   C. `generate()`
   D. `range()`



© 2019 ESTECO SpA

Correct answers

Quiz 1

1. A
2. D

Quiz 2

1. C
2. C
3. C
4. D
5. B

Quiz 3

1. B
2. C

© 2019 ESTECO SpA



Thank you!

esteco.com

Read the ESTECO Copyright Policy © 2019 ESTECO SpA