Java 8 Streams & Collectors

Patterns, performance, parallelization

@JosePaumard

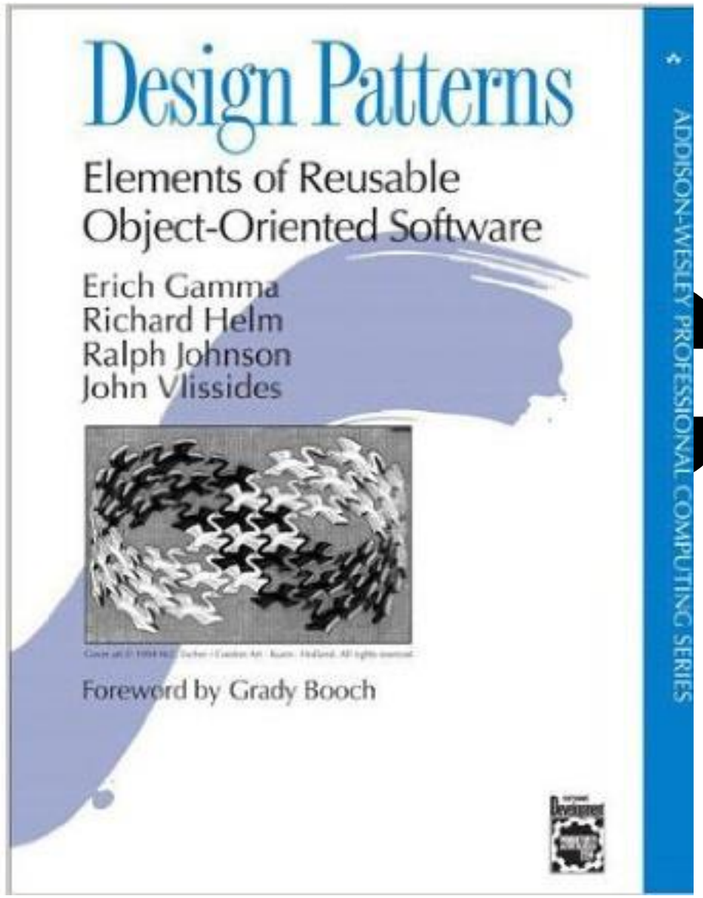
@JosePaumard

Stream

@JosePaumard

Stream Collectors

@JosePaumard



Stream Collectors

José PAUMARD

MCF Um. Paris 13

PhD Appl C.S.

Open source dev.

Independant

José PAUMARD

Open source dev.

Independant

José PAUMARD

Paris JUG

Devoxx FRANCE

José PAUMARD

pluralsight
hardcore dev and IT training

Parleys

Microsoft Virtual Academy

Code & slides

1st part: slides

- Stream
- Operations
- State
- Reduction
- Collectors

2nd part: live coding!

- Movies & actors
- Shakespeare

Questions ?

#J8Stream

@JosePaumard

Stream

What is a stream?

What is a stream?

Answer 1: a typed interface

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

   // ...
}
```

What is a stream?

Answer 1: a typed interface

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

   // ...
}
```

In fact: a new concept in the JDK

What is a stream?

What can I do with it?

What is a stream?

What can I do with it?

Answer: efficiently process high volumes of data, but also small ones

What is a stream?

What does *efficiently* mean?

What is a stream?

What does *efficiently* mean?

Two things:

What is a stream?

What does *efficiently* mean?

Two things:

1) in parallel, to leverage multicore CPU

What is a stream?

What does *efficiently* mean?

Two things:

1) in parallel, to leverage multicore CPU
2) in pipeline, to avoid the computation of intermediate data structures

What is a stream?

Why cant a collection be a stream?

What is a stream?

Why cant a collection be a stream?

In fact there are arguments for a Collection to be a Stream!

What is a stream?

Why cant a collection be a stream?

In fact there are arguments for a Collection to be a Stream!

1) this is where my data is!

What is a stream?

Why cant a collection be a stream?

In fact there are arguments for a Collection to be a Stream!

1) this is where my data is!
2) and I know the Collection API pretty well…

What is a stream?

Why cant a collection be a stream?

Two reasons:

1) we want to be free to create new concepts

What is a stream?

Why cant a collection be a stream?

Two reasons:

1) we want to be free to create new concepts
2) we dont want to add those concepts on well-known interfaces

So: what is a Stream?

Answers:

So: what is a Stream?

Answers:

1) an object on which I can define operations

So: what is a Stream?

Answers:

1) an object on which I can define operations
2) that does not hold any data

So: what is a Stream?

Answers:

1) an object on which I can define operations
2) that does not hold any data
3) that will not modify the data it processes

So: what is a Stream?

Answers:

1) an object on which I can define operations
2) that does not hold any data
3) that will not modify the data it processes
4) that will process data in « one pass »

So: what is a Stream?

Answers:

1) an object on which I can define operations
2) that does not hold any data
3) that will not modify the data it processes
4) that will process data in « one pass »
5) that is built on higly optimized algorithms, and that can work in parallel

How can I build a Stream?

Many patterns!

How can I build a Stream?

Many patterns!

Let us take one:

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
```

Operations on a Stream

Let us see a first operation: forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(p -> System.out.println(p)) ;
```

… that will display each person

Operations on a Stream

Let us see a first operation: forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(System.out::println) ;
```

… that will display each person

Operations on a Stream

Let us see a first operation: forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(System.out::println) ;
```

… that will display each person

```
o -> System.out.println(o) ≡ System.out::println
```

Operation: forEach()

First operation: forEach()

forEach(): takes a Consumer<T> as a parameter

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;
}
```

Operation: forEach()

First operation: forEach()

forEach(): takes a Consumer<T> as a parameter

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;
}
```

What is a functional interface?

Functional interface

Definition:

an interface with only one *abstract* method

Functional interface

Definition:

an interface with only one *abstract* method
(methods from Object do not count)

Functional interface

Definition:

an interface with only one *abstract* method
(methods from Object do not count)
(default & static methods do not count)

Functional interface

Definition:

an interface with only one *abstract* method
(methods from Object do not count)
(default & static methods do not count)

It « can » be annotated by @FunctionalInterface

Functional interface

Definition:

an interface with only one *abstract* method
(methods from Object do not count)
(default & static methods do not count)

It « can » be annotated by @FunctionalInterface

If present: the Java 8 compiler will help me

In fact…

Consumer is a bit more complex than that:

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

In fact…

What is a default method?

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Default methods

New concept in Java 8!

Default methods

New concept in Java 8!

Added to allow the adding of methods in interfaces without breaking the old implementations

Default methods

New concept in Java 8!

Added to allow the adding of methods in interfaces without breaking the old implementations

Ex: stream() on the Collection interface

Collection.stream()

```
public interface Collection<E> {

    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }




}
```

Collection.stream()

```
public interface Collection<E> {

    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }

    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, 0);
    }
}
```

Back to the Consumer interface

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Back to the Consumer interface

```
List<String> list = new ArrayList<>() ;

Consumer<String> c1 = s -> list.add(s) ;
Consumer<String> c2 = s -> System.out.println(s) ;
```

Back to the Consumer interface

```
List<String> list = new ArrayList<>() ;

Consumer<String> c1 = list::add ;
Consumer<String> c2 = System.out::println ;
```

Back to the Consumer interface

```
List<String> list = new ArrayList<>() ;

Consumer<String> c1 = list::add ;
Consumer<String> c2 = System.out::println ;

Consumer<String> c3 = c1.andThen(c2) ; // and we could chain more
```

Back to the Consumer interface

Beware concurrency!

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

Consumer<String> c1 = list::add ;

persons.stream()
       .forEach(c1) ; // concurrency?
```

Back to the Consumer interface

Beware concurrency!

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

Consumer<String> c1 = list::add ;

persons.stream().parallel()
        .forEach(c1) ; // concurrency?
```

Back to the Consumer interface

Beware concurrency!

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

Consumer<String> c1 = list::add ;

persons.stream().parallel()
        .forEach(c1) ; // concurrency? Baaaad pattern!
```

Back to the Consumer interface

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

Consumer<String> c1 = list::add ;
Consumer<String> c2 = System.out::println

persons.stream()
       .forEach(c1.andThen(c2)) ;
```

Problem: forEach() returns void

Back to the Consumer interface

But I also have a peek() method

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

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add) ; // Baaad pattern !
```

That returns another Stream

Back to Stream

So far we saw:

- forEach(Consumer)
- peek(Consumer)

Back to Stream

3rd method: filter(Predicate)

Method filter()

3rd method: filter(Predicate)

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Method filter()

3rd method: filter(Predicate)

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

```
Predicate<Person> p = person -> person.getAge() > 20 ;
```

Interface Predicate

Another functional interface

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;
}
```

Interface Predicate

In fact Predicate looks like this:

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;

   default Predicate<T> and(Predicate<? super T> other) { ... }

   default Predicate<T> or(Predicate<? super T> other) { ... }

   default Predicate<T> negate() { ... }
}
```

Interface Predicate

```
default Predicate<T> and(Predicate<? super T> other) {
   return t -> test(t) && other.test(t) ;
}
```

```
default Predicate<T> or(Predicate<? super T> other) {
   return t -> test(t) || other.test(t) ;
}
```

```
default Predicate<T> negate() {
   return t -> !test(t) ;
}
```

Interface Predicate: patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;
```

Interface Predicate: patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;

Predicate<Integer> p = p1.and(p2).or(p3) ; // (p1 AND p2) OR p3
```

Interface Predicate: patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;

Predicate<Integer> p = p1.and(p2).or(p3) ; // (p1 AND p2) OR p3
Predicate<Integer> p = p3.or(p1).and(p2) ; // (p3 OR p1) AND p2
```

Interface Predicate

In fact Predicate looks like this (bis):

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;

   // default methods

   static <T> Predicate<T> isEqual(Object o) { ... }
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

In fact things are bit more complicated:

```
static <T> Predicate<T> isEqual(Object o) {

   return (null == o)
            ? obj -> Objects.isNull(obj)
            : t -> o.equals(t) ;
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

In fact things are bit more complicated:

```
static <T> Predicate<T> isEqual(Object o) {

   return (null == o)
            ? Objects::isNull
            : o::equals ;
}
```

Interface Predicate

We also have:

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u);
}
```

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

stream and *filtered* are different objects

The filter() method returns a new object

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 1: what do we have in the *filtered* object?

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 1: what do we have in the *filtered* object?

Answer: the filtered data!

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 1: what do we have in the *filtered* object?

Answer: the filtered data… really?

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 1: what do we have in the *filtered* object?

Didnt we say: « a stream does not hold any data »?

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 1: what do we have in the *filtered* object?

Correct answer: nothing!

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 2: what does this code do?

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 2: what does this code do?

Answer: nothing…

Filtered Streams

In this code:

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Question 2: what does this code do?

Answer: nothing…

This call is a declaration, no data is processed!

Filtered Streams

The filter() call is a *lazy* call

Generally speaking:

A call to a method that returns a Stream
is a lazy call

Filtered Streams

The filter() call is a *lazy* call

Another way of saying it:

An operation that returns a Stream
is an intermediate operation

Intermediate call

Stating that a Stream does not hold any data

creates the notion of intermediate operation

adds the notion of lazyness to the Stream API

Intermediate call

Stating that a Stream does not hold any data

creates the notion of intermediate operation

adds the notion of lazyness to the Stream API

There must be some kind terminal operation somewhere…

Back to Consumer (bis)

What happens in that code?

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

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(resultat::add) ; // Baaad pattern !
```

Back to Consumer (bis)

What happens in that code?

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

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add) ; // Baaad pattern !
```

Answer is: nothing!

Back to Consumer (bis)

What happens in that code?

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

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add) ; // Baaad pattern !
```

1) nothing displayed

2) result is empty

What do we have so far?

We saw:

- forEach(Consumer)
- peek(Consumer)
- filter(Predicate)

What do we have so far?

We saw:

- forEach(Consumer)
- peek(Consumer)
- filter(Predicate)

The notion of lazyness / intermediate call

Let us carry on

And see the mapping operation

Mapping

Mapping operation

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<String> names =
   stream.map(person -> person.getName()) ;
```

Mapping

Mapping operation

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<String> names =
   stream.map(person -> person.getName()) ;
```

The map() method returns a Stream

It is thus lazy / intermediate

Interface Function

The map() method takes a Function:

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;
}
```

Interface Function

Which in fact is:

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   default <V> Function<V, R> compose(Function<V, T> before) ;

   default <V> Function<T, V> andThen(Function<R, V> after) ;
}
```

Interface Function

Which in fact is:

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   default <V> Function<V, R> compose(Function<V, T> before) ;

   default <V> Function<T, V> andThen(Function<R, V> after) ;
}
```

Beware the generics!

Interface Function

Which in fact is:

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   default <V> Function<V, R> compose(
      Function<? super V, ? extends T> before) ;

   default <V> Function<T, V> andThen(
      Function<? super R, ? extends V> after) ;
}
```

Interface Function

Which in fact is:

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   // default methods

   static <T> Function<T, T> identity() {
      return t -> t ;
   }
}
```

Other functions

There are special types of functions:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
}
```

Other functions

There are special types of functions:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);
}
```

Other functions

There are special types of functions:

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
}
```

What do we have so far?

We saw:

- forEach(Consumer)
- peek(Consumer)
- filter(Predicate)
- map(Function)

Let us carry on

Method flatMap()

Method flatMap()

flatMap() = flattens a Stream

The signature is:

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

The passed function returns a Stream

Method flatMap()

Let us take an example:

```
List<Movie> movies = ... ; // a movie has a set of Actor

movies.stream()
      .map(movie -> movie.actors()) // Stream<Set<Actors>>
```

Method flatMap()

Let us take an example:

```
List<Movie> movies = ... ; // a movie has a set of Actor

movies.stream()
      .map(movie -> movie.actors().stream()) // Stream<Stream<Actors>>
```

Method flatMap()

Let us take an example:

```
List<Movie> movies = ... ; // a movie has a set of Actor

movies.stream()
      .map(movie -> movie.actors().stream()) // Stream<Stream<Actors>>
```

```
Function<Movie, Stream<Actors>> mapper =
   movie -> movie.actors().stream();
```

Method flatMap()

Let us take an example:

```
List<Movie> movies = ... ; // a movie has a set of Actor

movies.stream()
      .flatMap(movie -> movie.actors().stream()) // Stream<Actors>
```

Method flatMap()

flatMap() = flattens a Stream

Signature:

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

Method flatMap()

flatMap() = flattens a Stream

Signature:

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

Since it returns a Stream, it is a lazy / intermediate call

What do we have so far?

We have 3 types of methods:

- forEach(): consumes
- peek(): consumes, and transmits
- filter(): filters
- map(): transforms
- flatMap(): transforms and flattens

What do we have so far?

And what about reductions?

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

Two things to stress out here

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, Intege::sum) ;
```

Two things to stress out here

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

0 is the « default value » of the reduction operation returned if the stream is « empty »

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

0 is the « default value » of the reduction operation

This « default value » has to be the identity element of the reduction operation

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

0 is the « default value » of the reduction operation

If it is not, then the computation will fail!

Reduction

Why?

- This 0 is returned in case of an empty Stream
- Red(0, p) is returned in case of a singleton Stream

In fact, 0 is used to compute the reduction of the Stream

Reduction

What is going to happen if the reduction operation has no identity value?

Reduction

What is going to happen if the reduction operation has no identity value?

But of course we do not use such reductions…

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

The reduction operation should be associative

Reduction

Reduction method:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

The reduction operation should be associative

If not, we will have trouble in parallel

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;
```

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ;
```

☺

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;                ☺
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ; ☺
red3 = (i1, i2) -> i1*i2 ;
```

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;                ☺
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ;      ☺
red3 = (i1, i2) -> i1*i2 ;                  ☺
red4 = (i1, i2) -> i1*i1 + i2*i2 ;
```

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;                ☺
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ;      ☺
red3 = (i1, i2) -> i1*i2 ;                  ☺
red4 = (i1, i2) -> i1*i1 + i2*i2 ;          ☹
red5 = (i1, i2) -> (i1 + i2) / 2 ;
```

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;                ☺
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ;      ☺
red3 = (i1, i2) -> i1*i2 ;                  ☺
red4 = (i1, i2) -> i1*i1 + i2*i2 ;          ☹
red5 = (i1, i2) -> (i1 + i2) / 2 ;          ☹
red6 = (i1, i2) -> i2 ;
```

Reduction

Associative, or not associative?

```
red1 = (i1, i2) -> i1 + i2 ;                ☺
red2 = (i1, i2) -> i1 > i2 ? i1 : i2 ;      ☺
red3 = (i1, i2) -> i1*i2 ;                  ☺
red4 = (i1, i2) -> i1*i1 + i2*i2 ;          ☹
red5 = (i1, i2) -> (i1 + i2) / 2 ;          ☹
red6 = (i1, i2) -> i2 ;                     ☺
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, (age1, age2) -> age1 > age2 ? age1 : age2) ;
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, Integer::max) ;
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, (age1, age2) -> age1 > age2 ? age1 : age2) ;
```

```
List<Integer> ages = new ArrayList<>() ;
Stream<Integer> stream = ages.stream() ; // empty stream

> max = 0
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, (age1, age2) -> age1 > age2 ? age1 : age2) ;
```

```
List<Integer> ages = Arrays.asList(2) ;
Stream<Integer> stream = ages.stream() ; // singleton stream

> max = 2
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, (age1, age2) -> age1 > age2 ? age1 : age2) ;
```

```
List<Integer> ages = Arrays.asList(-1) ;
Stream<Integer> stream = ages.stream() ; // singleton stream

> max = 0
```

Example of max

Let us reduce with a max

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer max =
   stream.reduce(0, (age1, age2) -> age1 > age2 ? age1 : age2) ;
```

```
List<Integer> ages = Arrays.asList(-1, -2) ;
Stream<Integer> stream = ages.stream() ;

> max = 0
```

Example of max

The reduction with a max has a problem : why?

Example of max

The reduction with a max has a problem : why?

Answer: the max reduction has no identity element, thus no default value

Example of max

The reduction with a max has a problem : why?

Answer: the max reduction has no identity element, thus no default value

Solution: introduce the notion of Optional

Method max()

Optional is the return type of the max() method

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Optional<Integer> max =
   stream.max(Comparator.naturalOrder()) ;
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

```
String s = opt.orElse("") ; // this is a default value that
                            // is valid for our application
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
String s = opt.orElseThrow(MyException::new) ; // lazy initialization
```

Optional can be built

One can explicitly build optionals

```
Optional<String> opt = Optional.<String>empty() ;
```

```
Optional<String> opt = Optional.of("one") ; // not null
```

```
Optional<String> opt = Optional.ofNullable(s) ; // may be null
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

```
Optional<U> map(Function<T, U> mapper) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

```
Optional<U> map(Function<T, U> mapper) ;
```

```
Optional<U> flatMap(Function<T, Optional<U>> mapper) ;
```

Optional: new patterns sighted!

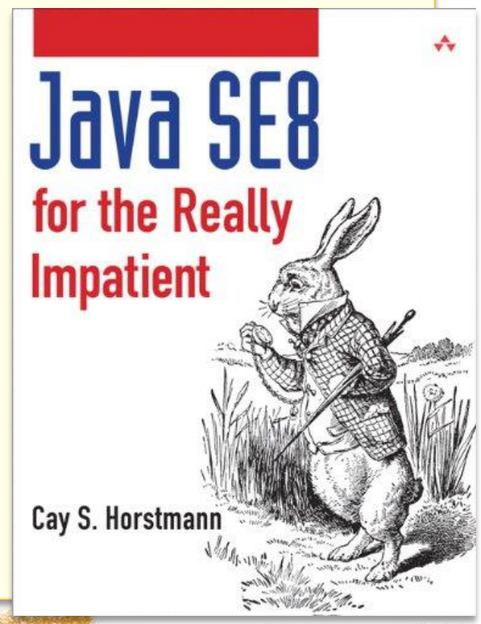
```
public class NewMath {

   public static Optional<Double> inv(Double d) {

      return d == 0.0d ? Optional.empty() :
                         Optional.of(1/d) ;
   }

   public static Optional<Double> sqrt(Double d) {

      return d < 0.0d ? Optional.empty() :
                        Optional.of(Math.sqrt(d)) ;
   }
}
```



Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.forEach(
   d1 -> NewMath.inv(d1)                                  // Optional<Double>
                .flatMap(d2 -> NewMath.sqrt(d2)) // Optional<Double>
                .ifPresent(result::add)
) ;
```

```
doubles : [-1.0, 0.0, 1.0]
result  : [1.0]
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.forEach(
   d1 -> NewMath.inv(d1)                                  // Optional<Double>
                .flatMap(d2 -> NewMath.sqrt(d2)) // Optional<Double>
                .ifPresent(result::add)
) ;
```

```
doubles : [-1.0, 0.0, 1.0]
result  : [1.0]
```

Optional: new patterns sighted!

```
Function<Double, Optional<Double>> f =
 d -> NewMath.inv(d)                              // Optional<Double>
             .flatMap(d -> NewMath.sqrt(d))// Optional<Double>
```

Optional: new patterns sighted!

```
Function<Double, Optional<Double>> f =
 d -> NewMath.inv(d)                          // Optional<Double>
                .flatMap(NewMath::sqrt);      // Optional<Double>
```

Optional: new patterns sighted!

```
d -> NewMath.inv(d)                          // Optional<Double>
              .flatMap(NewMath::sqrt)        // Optional<Double>
              .map(Stream::of)               // Optional<Stream<Double>>
```

Optional: new patterns sighted!

```
d -> NewMath.inv(d)                          // Optional<Double>
              .flatMap(NewMath::sqrt)        // Optional<Double>
              .map(Stream::of)               // Optional<Stream<Double>>
              .orElse(Stream.empty())        // Stream<Double>
```

Optional: new patterns sighted!

```
Function<Double, Stream<Double>> f =
d -> NewMath.inv(d)                          // Optional<Double>
                .flatMap(NewMath::sqrt)      // Optional<Double>
                .map(Stream::of)             // Optional<Stream<Double>>
                .orElse(Stream.empty()) ;    // Stream<Double>
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.stream()
        .flatMap(
           d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)           // Optional<Double>
              .map(Stream::of)                  // Optional<Stream<Double>>
              .orElse(Stream.empty())           // Stream<Double>
        )                                       // Stream<Double>
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.stream()
       .flatMap(
          d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)          // Optional<Double>
              .map(Stream::of)                 // Optional<Stream<Double>>
              .orElse(Stream.empty())          // Stream<Double>
       )                                       // Stream<Double>
       .collect(Collectors.toList()) ;
```

Back on the reduce() method

Two return types:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Optional<Integer> opt =
   stream.reduce((age1, age2) -> age1 + age2) ;
```

Back on the reduce() method

Two return types:

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer sum =
   stream.reduce(0, Integer::sum) ;
```

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Optional<Integer> opt =
   stream.reduce(Integer::sum) ;
```

More on the reduction

A reduction never returns a Stream

- max(), min() [optionals]
- count()

Boolean reductions:

- allMatch(), noneMatch, anyMatch()

Return optionals:

- findFirst(), findAny() (parallel!)

Note on the reduction

A reduction never returns a Stream

So it is not a lazy step / intermediate operation

A reduction always triggers the computation

They are terminal *operations*

Terminal operation

Example

```
List<Person> persons = ... ;

... =
persons.map(person -> person.getAge())   // Stream<Integer>
       .filter(age -> age > 20)          // Stream<Integer>
       .min(Comparator.naturalOrder()) ;
```

Terminal operation

Example

```
List<Person> persons = ... ;


... =
persons.map(Person::getAge)                  // Stream<Integer>
       .filter(age -> age > 20)              // Stream<Integer>
       .min(Comparator.naturalOrder()) ;
```

Terminal operation

Let us write a full map / filter / reduce

```
List<Person> persons = ... ;

Optional<Integer> age =
persons.map(person -> person.getAge())   // Stream<Integer>
       .filter(age -> age > 20)          // Stream<Integer>
       .min(Comparator.naturalOrder()) ; // terminal operation
```

Terminal operation

Let us write a full map / filter / reduce

```
List<Person> persons = ... ;

boolean b =
persons.map(person -> person.getLastName())
       .allMatch(length < 20) ;                    // terminal operation
```

Terminal operation

Let us write a full map / filter / reduce

```
List<Person> persons = ... ;

boolean b =
persons.map(Person::getAge)
       .allMatch(length < 20) ;                    // terminal operation
```

Terminal operation

Let us write a full map / filter / reduce

```
List<Person> persons = ... ;

boolean b =
persons.map(Person::getAge)
       .allMatch(length < 20) ;                    // terminal operation
```

Much more efficient to compute this in one pass over the data: « short cutting methods »

What is a Stream?

An object on which we can define operations on data

No *a priori* limit on the size of the data

Typical operations: map / filter / reduce

What is a Stream?

Pipeline approach:

1) we define the operations
2) we trigger the computations by calling a terminal operation

What is a Stream?

And bytheway:

We can only have one terminal operation for a Stream

In case, another Stream must be built

How one can build a Stream?

Let us have a look at the stream() default method

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

```
// Collection interface
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

How one can build a Stream?

Let us have a look at the stream() default method

```
// StreamSupport class
public static <T> Stream<T> stream(
   Spliterator<T> spliterator, boolean parallel) {

   Objects.requireNonNull(spliterator) ;
   return new ReferencePipeline.Head<>(
                  spliterator,
                  StreamOpFlag.fromCharacteristics(spliterator),
                  parallel) ;
}
```

How one can build a Stream?

The spliterator() is defined in ArrayList

```
// ArrayList class
@Override
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

How one can build a Stream?

The spliterator() is defined in ArrayList

```
// ArrayList class
@Override
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

The Spliterator defines how to access the data

ArrayList: it reads the array

Spliterator

Let us see the method we need to implement

```
// Spliterator interface
boolean tryAdvance(Consumer<? super T> action);
```

Consumes the next element, if it exists

Spliterator

Let us see the method we need to implement

```
// Spliterator interface
boolean tryAdvance(Consumer<? super T> action);
```

Consumes the next element, if it exists

A Stream does not assume that all the elements are available at build time

Spliterator

Let us see the method we need to implement

```
// Spliterator interface
Spliterator<T> trySplit();
```

Used in parallel computations: tries to split the data in two (fork / join)

Spliterator

Let us see the method we need to implement

```
// Spliterator interface
long estimateSize();
```

Returns an estimation of the size of this stream

Spliterator

And there are also default methods

```
// Spliterator interface
default void forEachRemaining(Consumer<? super T> action) {
   do { } while (tryAdvance(action));
}
```

```
// Spliterator interface
default long getExactSizeIfKnown() {
   return (characteristics() & SIZED) == 0 ? -1L : estimateSize();
}
```

Spliterator

And there is a last method:

```
// interface Spliterator
int characteristics();
```

Implémentation for ArrayList

```
// pour ArrayList
public int characteristics() {
    return Spliterator.ORDERED |
           Spliterator.SIZED |
           Spliterator.SUBSIZED;
}
```

Spliterator

And there is a last method:

```
// interface Spliterator
int characteristics();
```

Implémentation for HashSet

```
// pour HashSet
public int characteristics() {
    return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
            Spliterator.DISTINCT;
}
```

Characteristics of a Stream

Bits defined in the characteristics word

Characteristic	
ORDERED	The order matters
DISTINCT	No duplication
SORTED	Sorted
SIZED	The size is known
NONNULL	No null values
IMMUTABLE	Immutable
CONCURRENT	Parallelism is possible
SUBSIZED	The size is known

Characteristics of a Stream

Some operations will switch some bits

Method	Set to 0	Set to 1
filter()	SIZED	-
map()	DISTINCT, SORTED	-
flatMap()	DISTINCT, SORTED, SIZED	-
sorted()	-	SORTED, ORDERED
distinct()	-	DISTINCT
limit()	SIZED	-
peek()	-	-
unordered()	ORDERED	-

Characteristics of a Stream

Characteristics are taken into account when computing a result

```
// HashSet
HashSet<String> strings = ... ;
strings.stream()
       .distinct() // no processing is triggered
       .sorted()
       .collect(Collectors.toList()) ;
```

Characteristics of a Stream

Characteristics are taken into account when computing a result

```
// SortedSet
TreeSet<String> strings = ... ;
strings.stream()
       .distinct()
       .sorted()   // no processing is triggered
       .collect(Collectors.toList()) ;
```

Stream implementation

Complex!

Two parts:

1) algorithms are in ReferencePipeline
2) data access: Spliterator, can be overriden

A few words on Comparator

We wrote:

```
// Comparator interface
Comparator cmp = Comparator.naturalOrder() ;
```

A few words on Comparator

We wrote:

```
// Comparator interface
Comparator cmp = Comparator.naturalOrder() ;
```

```
// Comparator interface
@SuppressWarnings("unchecked")
public static <T extends Comparable<? super T>>
Comparator<T> naturalOrder() {

   return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
}
```

A few words on Comparator

```
// Comparators class
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

   INSTANCE;









}
```

A few words on Comparator

```
// Comparators class
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

   INSTANCE;









}
```

```
public class MySingleton {

   INSTANCE;

   private MySingleton() {}

   public static MySingleton getInstance() {
      // some buggy double-checked locking code
      return INSTANCE;
   }
}
```

A few words on Comparator

```
// Comparators class
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

   INSTANCE;

   public int compare(Comparable<Object> c1, Comparable<Object> c2) {
      return c1.compareTo(c2);
   }

   public Comparator<Comparable<Object>> reversed() {
      return Comparator.reverseOrder();
   }
}
```

A few words on Comparator

So we can write:

```
// Comparator interface
Comparator<Person> cmp =
   Comparator.comparing(Person::getLastName)
             .thenComparing(Person::getFirstName)
             .thenComparing(Person::getAge) ;
```

A few words on Comparator

Method comparing()

```
// Comparator interface
public static
   <T, U> Comparator<T>
   comparing(Function<T, U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

A few words on Comparator

Method comparing()

```
// Comparator interface
public static
   <T, U extends Comparable<U>> Comparator<T>
   comparing(Function<T, U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return (Comparator<T>)
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

A few words on Comparator

Method comparing()

```
// Comparator interface
public static
   <T, U extends Comparable<? super U>> Comparator<T>
   comparing(Function<? super T, ? extends U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return (Comparator<T> & Serializable)
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

A few words on Comparator

Method thenComparing()

```
// Comparator interface
default
   <U> Comparator<T>
   thenComparing(Function<T, U> keyExtractor) {

   return thenComparing(comparing(keyExtractor));
}
```

A few words on Comparator

Method thenComparing()

```
// Comparator interface
default
   <U extends Comparable<? super U>> Comparator<T>
   thenComparing(Function<? super T, ? extends U> keyExtractor) {

   return thenComparing(comparing(keyExtractor));
}
```

A few words on Comparator

Method thenComparing()

```
// Comparator interface
default Comparator<T> thenComparing(Comparator<? super T> other) {

   Objects.requireNonNull(other);
   return (Comparator<T> & Serializable) (c1, c2) -> {
      int res = compare(c1, c2);
      return (res != 0) ? res : other.compare(c1, c2);
   };
}
```

What do we have so far?

API Stream

- intermediate operations
- terminal operations
- implementation built on 2 classes

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
Stream<Persons> stream = persons.limit(1_000) ;
```

… selects the 1000 *first* people of the list

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
Stream<Persons> stream = persons.limit(1_000) ;
```

… selects the 1000 *first* people of the list

This operations needs a counter

How will it work in parallel?

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

« the order of the names is the same as the order of the people »

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

« the order of the names is the same as the order of the people »

How will it behave in parallel?

Stateless / stateful operations

This code:

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .unordered()
          .collect(Collectors.toList()) ;
```

« the order of the names is the same as the order of the people »

We should relax this constraint!

Wrap-up on Stream

One can define operations on a Stream:

- intermediate & terminal
- stateless & stateful

A Stream can be processed in parallel

A Stream has a state, used to optimize computations

Stream & performance

Two elements:

- lazy processing, on one pass over the data
- parallel processing

Stream & performance

Two elements:

- lazy processing, on one pass over the data
- parallel processing

Stream<T> versus
IntStream, LongStream, DoubleStream

Stream & performance

Let see this example again:

```
ArrayList<Person> persons = ... ;


persons.stream()
       .map(Person::getAge)
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Let see this example again:

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Let see this example again:

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Let see this example again:

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)         // Stream<Integer> re-boxing re-☹
       .sum() ;
```

Stream & performance

Let see this example again:

```
ArrayList<Person> persons = ... ;


persons.stream()                          // Stream<Person>
       .map(Person::getAge)               // Stream<Integer> boxing ☹
       .filter(age -> age > 20)           // Stream<Integer> re-boxing re-☹
       .sum() ;                           // no such sum() method on Stream<T>
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)  // Stream<Integer> re-boxing re-☹
       .mapToInt(age -> age.getValue()) // IntStream ☺
       .sum() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)  // Stream<Integer> re-boxing re-☹
       .mapToInt(Integer::getValue)     // IntStream ☺
       .sum() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .sum() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

int sum =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .sum() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

??? =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .max() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

OptionalInt opt =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .max() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

??? =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .average() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

OptionalInt opt =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .average() ;
```

Stream & performance

Let see this example again

```
ArrayList<Person> persons = ... ;

IntSummaryStatistics stats =
persons.stream()
       .mapToInt(Person::getAge)
       .filter(age -> age > 20)
       .summaryStatistics() ;
```

In one pass we have:
count, sum, min, max, average

Parallel streams

How to build a parallel Stream?

```
ArrayList<Person> persons = ... ;

Stream<Person> stream1 = persons.parallelStream() ;

Stream<Person> stream2 = persons.stream().parallel() ;
```

Built on top of fork / join

Parallel streams

By default, a parallel streams uses the default « fork join common pool », defined at the JVM level

The same pool is shared by all the streams

Parallelism

By default, a parallel streams uses the default « fork join common pool », defined at the JVM level

By default, the parallelism level is the # of cores

We can set it with a sytem property:

```
System.setProperty(
   "java.util.concurrent.ForkJoinPool.common.parallelism", 2) ;
```

Parallelism

We can also decide the FJ pool for a given Stream

```
List<Person> persons = ... ;

ForkJoinPool fjp = new ForkJoinPool(2) ;
fjp.submit(
   () ->                                  //
   persons.stream().parallel()            // this is an implementation
          .mapToInt(p -> p.getAge())      // of Callable<Integer>
          .filter(age -> age > 20)        //
          .average()                      //
).get() ;
```

Reduction

A reduction can be seen as a SQL agregation (sum, min, max, avg, etc…)

Reduction

A reduction can be seen as a SQL agregation (sum, min, max, avg, etc…)

But it can be seen in a more general sense

Reduction

Example: sum()

Reduction

Example: sum()

The result is an integer, its initial value being 0

Reduction

Example: sum()

The result is an integer, its initial value being 0

Adding an element to the result is… adding

Reduction

Example: sum()

The result is an integer, its initial value being 0

Adding an element to the result is… adding

Combining two partial results (think parallel) is also « adding »

Reduction

So we have 3 operations:

Reduction

So we have 3 operations:

- the definition of a container, which holds the result

Reduction

So we have 3 operations:

- the definition of a container, which holds the result
- adding an element to that container

Reduction

So we have 3 operations:

- the definition of a container, which holds the result
- adding an element to that container
- combining two partialy filled containers

Reduction

So we have 3 operations:

- the definition of a container, which holds the result
- adding an element to that container
- combining two partialy filled containers

This is a much more broader vision than just an agregation

Reduction

So a reduction is based on 3 operations:

- a constructor: Supplier

- an accumulator: Function

- a combiner: Function

Reduction

So a reduction is based on 3 operations:

- a constructor: Supplier

```
() -> new StringBuffer() ;
```

- an accumulator: Function

- a combiner: Function

Reduction

So a reduction is based on 3 operations:

- a constructor: Supplier

```
() -> new StringBuffer() ;
```

- an accumulator: Function

```
(StringBuffer sb, String s) -> sb.append(s) ;
```

- a combiner: Function

Reduction

So a reduction is based on 3 operations:

- a constructor: Supplier

```
() -> new StringBuffer() ;
```

- an accumulator: Function

```
(StringBuffer sb, String s) -> sb.append(s) ;
```

- a combiner: Function

```
(StringBuffer sb1, StringBuffer sb2) -> sb1.append(sb2) ;
```

Reduction

So a reduction is based on 3 operations:

- a constructor: Supplier

```
StringBuffer::new ;
```

- an accumulator: Function

```
StringBuffer::append ;
```

- a combiner: Function

```
StringBuffer::append ;
```

Reduction: implementation

Example 1: reducing in a StringBuffer

```
List<Person> persons = ... ;

StringBuffer result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          StringBuffer::new,     // constructor
          StringBuffer::append, // accumulator
          StringBuffer::append  // combiner
       ) ;
```

Reduction: collectors

Example 1: using a Collectors

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining()
       ) ;
```

Reduction: collectors

Collecting in a String

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining(", ")
       ) ;
```

Reduction: collectors

Collecting in a String

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining(", ", "{", "}")
       ) ;
```

Reduction

Example 2:

- a constructor: Supplier

```
() -> new ArrayList() ;
```

- an accumulator: Function

```
(list, element) -> list.add(element) ;
```

- a combiner: Function

```
(list1, list2) -> list1.addAll(list2) ;
```

Reduction

Example 2:

- a constructor: Supplier

```
ArrayList::new ;
```

- an accumulator: Function

```
Collection::add ;
```

- a combiner: Function

```
Collection::allAll ;
```

Reduction: implementation

Example 2: reduction in a List

```
List<Person> persons = ... ;

ArrayList<String> names =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          ArrayList::new,     // constructor
          Collection::add,    // accumulator
          Collection::addAll // combiner
       ) ;
```

Reduction: collectors

Example 2: reduction in a List

```
List<Person> persons = ... ;

List<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toList()
       ) ;
```

Reduction: implementation

Example 3: reduction in a Set

```
List<Person> persons = ... ;

HashSet<String> names =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          HashSet::new,       // constructor
          Collection::add,    // accumulator
          Collection::addAll // combiner
       ) ;
```

Reduction: collectors

Example 3: reduction in a Set

```
List<Person> persons = ... ;

Set<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toSet()
       ) ;
```

Reduction: collectors

Example 4: reduction in a given collection

```
List<Person> persons = ... ;

TreeSet<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toCollection(TreeSet::new)
       ) ;
```

Collectors

Collectors class: 33 static methods

An amazing toolbox!

Collectors: groupingBy

HashMap: age / list of the people

```
Map<Integer, List<Person>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(Person::getAge)
        ) ;
```

Collectors: groupingBy

HashMap: age / # of people

```
Map<Integer, Long> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.counting() // « downstream collector »
           )
        ) ;
```

Collectors: groupingBy

HashMap: age / list of the names

```
Map<Integer, List<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping(     //
                 Person::getLastName // downstream collector
              )                      //
           )
        ) ;
```

Collectors: groupingBy

HashMap: age / names joined in a single String

```
Map<Integer, String> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping( // 1st downstream collector
                 Person::getLastName
                 Collectors.joining(", ") // 2nd downstream collector
              )
           )
        ) ;
```

Collectors: groupingBy

HashMap: age / names sorted alphabetically

```
Map<Integer, TreeSet<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping(
                 Person::getLastName
                 Collectors.toCollection(TreeSet::new)
              )
           )
        ) ;
```

Collectors: groupingBy

HashMap: the same, sorted by age

```
TreeMap<Integer, TreeSet<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              TreeMap::new,
              Collectors.mapping(
                 Person::getLastName
                 Collectors.toCollection(TreeSet::new)
              )
           )
        ) ;
```

Wrap-up

Stream + Collectors =

New tools for data processing (map / filter / reduce)

1) lazy execution
2) parallelism

Live coding to come!

Wrap-up

More to come:

- use cases
- live coding!


DEVOXX
Coffee time !
#Stream8
@JosePaumard

Java 8 Streams & Collectors

Patterns, performance, parallelization

@JosePaumard

Questions ?

#J8Stream

@JosePaumard

« movies and actors »

https://github.com/JosePaumard/jdk8-lambda-tour

http://introcs.cs.princeton.edu/java/data/

Movies and actors

File: « movies-mpaa.txt »

With 14k movies from 1916 to 2004

With:

- the title of the movie
- the release year
- the list of the actors (170k)

Greatest release year

1st question: which year saw the most movies?

Greatest release year

1st question: which year saw the most movies?

1st step: we can build a hashmap

- the keys are the release year
- the values the # of movies in that year

Greatest release year

1st question: which year saw the most movies?

1st step: we can build a hashmap

- the keys are the release year
- the values the # of movies in that year

2nd step: get the key value pair with the greatest value

Greatest release year

Live coding

Greatest release year

1st question: which year saw the most movies?

Solution: map / reduce

Most seen actor

2nd question: most seen actor?

Most seen actor

2nd question: most seen actor?

Once again, we can build a hashmap:

- the keys are the actors
- the values are the # of movies

Most seen actor

2nd question: most seen actor?

Let us build the set of all the actors

```
Set<Actor> actors =
    movies.stream()
    .flatMap(movie -> movie.actors().stream())
    .collect(Collector.toSet()) ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream()
 .collect(









 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream()
 .collect(
    Collectors.toMap(




    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream()
 .collect(
    Collectors.toMap(
       actor -> actor,



    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream()
 .collect(
    Collectors.toMap(
       Function.identity(),




    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream()
 .collect(
    Collectors.toMap(
       Function.identity(),
       actor -> movies.stream()
                      .filter(movie -> movie.actors().contains(actor))
                      .count()
    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream().parallel() // T = 40s ☹
 .collect(
    Collectors.toMap(
       Function.identity(),
       actor -> movies.stream()
                      .filter(movie -> movie.actors().contains(actor))
                      .count()
    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
actors.stream().parallel() // 170k
 .collect(
    Collectors.toMap(
       Function.identity(),
       actor -> movies.stream() // 14k                    2,4G!!
                      .filter(movie -> movie.actors().contains(actor))
                      .count()
    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor

2nd question: most seen actor?

```
Set<Actor> actors =
    movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream of actors
    .collect(Collector.toSet()) ;
```

Most seen actor

2nd question: most seen actor?

```
Set<Actor> actors =
    movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream of actors
    .collect(Collector.toSet()) ;
```

And in this stream, if an actor played in 2 movies, she appears twice

Most seen actor

Live coding

Most seen actor

2nd question: most seen actor?

Answer: Frank Welker

Most seen actor

2nd question: most seen actor?

Answer: Frank Welker

… and in fact he’s not really the most « seen » actor

Most seen actor

2nd question: most seen actor?

Answer: Frank Welker

Most seen actor

2nd question: most ~~seen~~ heard actor?

Answer: Frank Welker

Most seen actor in a year

3rd question: most seen actor in a given year?

Most seen actor in a year

3rd question: most seen actor in a given year?

So it is a hashmap in which:

- the keys are the actors
- the values are other hashmaps

Most seen actor in a year

3rd question: most seen actor in a given year?

So it is a hashmap in which:

- the keys are the actors
- the values are other hashmaps
  - in which keys are the release years
  - and the values the # of movies → AtomicLong

Most seen actor in a year

3rd question: most seen actor in a given year?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream())
    .collect(
       Collectors.groupingBy( // build a Map<year, ...>
          movie -> movie.releaserYear(),
          // build a Map<Actor, # of movies>
       )
    )
    .entrySet().stream()
    .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor in a year

3rd question: most seen actor in a given year?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream())
    .collect(
       Collectors.groupingBy( // build a Map<year, ...>
          movie -> movie.releaserYear(),
          // build a Map<Actor, # of movies> -> custom DS Collector
       )
    )
    .entrySet().stream()
    .max(Map.Entry.comparingByValue()).get() ;
```

Most seen actor in a year

3rd question: most seen actor in a given year?

So it is a hashmap in which:

- the keys are the actors
- the values are other hashmaps
  - in which keys are the release years
  - and the values the # of movies → AtomicLong

And we want the max by the # of movies

Most seen actor in a year

3rd question: most seen actor in a given year?

We need 3 things to build a collector

- a constructor for the resulting container
- an accumulator to add an element to the container
- a combiner that merges two partially filled containers

Most seen actor in a year

Live coding

Most seen actor in a year

3rd question: most seen actor in a given year?

Most seen actor in a year

3rd question: most seen actor in a given year?

Answer: Phil Hawn, in 1999
played in 24 movies

Most seen actor in a year

3rd question: most seen actor in a given year?

Answer: Phil Hawn, in 1999
played in 24 movies

« Shakespeare plays Scrabble »

https://github.com/JosePaumard/jdk8-lambda-tour

http://introcs.cs.princeton.edu/java/data/

Scrabble

Two files:

- OSPD = official personnal Scrabble dictionary
- The words used by Shakespeare

Scrabble

Two files:

- OSPD = official personnal Scrabble dictionary
- The words used by Shakespeare
- And some documentation (Wikipedia)

```
private static final int [] scrabbleENScore = {
// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,  q, r, s, t, u, v, w, x, y,  z
   1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10} ;
```

Scrabble: score of a word

1st question: how to compute the score of a word

```
H E L L O
4 1 1 1 1 // from the scrabbleENScore array
```

Scrabble: score of a word

1st question: how to compute the score of a word

```
H E L L O
4 1 1 1 1 // from the scrabbleENScore array
SUM
= 8
```

Scrabble: score of a word

1st question: how to compute the score of a word

```
H E L L O // this is a stream of the letters of the word HELLO
4 1 1 1 1
SUM
= 8
```

Scrabble: score of a word

1st question: how to compute the score of a word

```
H E L L O // this is a stream of the letters of the word HELLO
4 1 1 1 1 // mapping : letter -> score of the letter
SUM
= 8
```

Scrabble: score of a word

1st question: how to compute the score of a word

```
H E L L O // this is a stream of the letters of the word HELLO
4 1 1 1 1 // mapping : letter -> score of the letter
SUM       // reduction : sum
= 8
```

Scrabble: score of a word

Live coding

Scrabble: score of a word

1st question: how to compute the score of a word

= map / reduce

Scrabble: score of Shakespeare

2nd question: compute the best word of Shakespeare

Scrabble: score of Shakespeare

2nd question: compute the best word of Shakespeare

1) Histogram: score of the words / list of the words

Scrabble: score of Shakespeare

2nd question: compute the best word of Shakespeare

1) Histogram: score of the words / list of the words

2) Max by key value

Scrabble: score of Shakespeare

2nd question: compute the best word of Shakespeare

1) Histogram: score of the words / list of the words
   - HashMap, Collectors.groupingBy()
2) Max by key value

Scrabble: score of Shakespeare

2nd question: compute the best word of Shakespeare

1) Histogram: score of the words / list of the words
   - HashMap, Collectors.groupingBy()
2) Max by key value
   - Shouldnt be too hard

Scrabble: score of Shakespeare

Live coding

Scrabble: score of Shakespeare

3rd question: filter out the wrong words

We need to add a filtering step

Scrabble: score of Shakespeare

Live coding

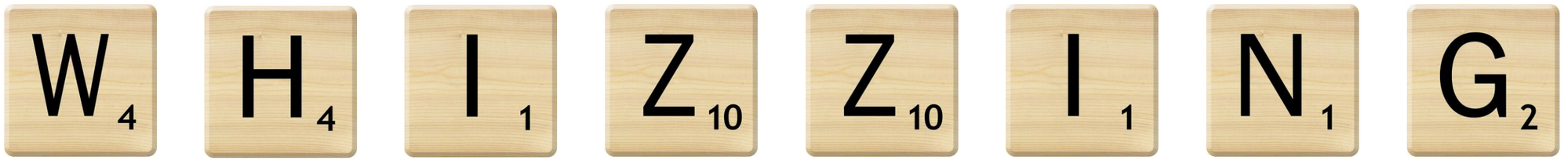
Scrabble: score of Shakespeare

3rd question: filter out the wrong words

Solution: map / filter / reduce

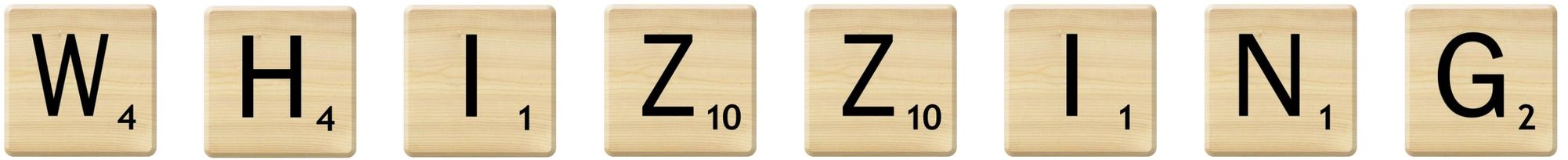
Scrabble: score of Shakespeare

Question: can one really write this word?



Scrabble: score of Shakespeare

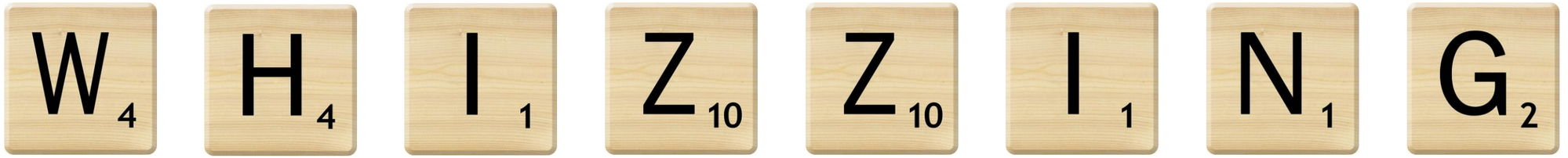
Question: can one really write this word?




```
private static final int [] scrabbleENDistribution = {
// a, b, c, d, e,  f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Scrabble: score of Shakespeare

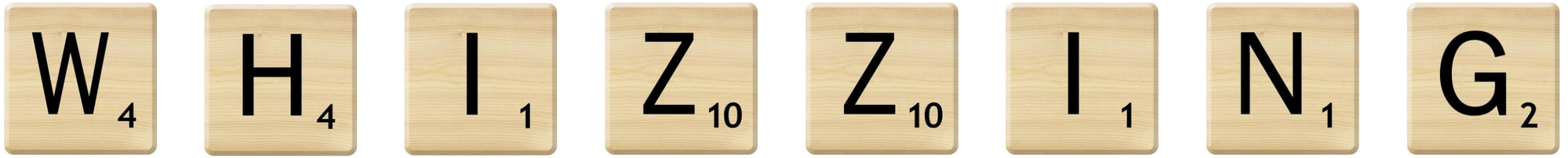
Answer: no…




```
private static final int [] scrabbleENDistribution = {
// a, b, c, d, e,  f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Filtering V2

4th question: how to check if I have the available letters?




```
private static final int [] scrabbleENDistribution = {
// a, b, c, d, e,  f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Filtering V2

4th question: how to check if I have the available letters?

1 1 2 2 1 1 needed letters

Filtering V2

4th question: how to check if I have the available letters?

1	1	2	2	1	1	needed letters
1	4	9	1	6	3	available letters

Filtering V2

4th question: how to check if I have the available letters?

1	**1**	**2**	**2**	**1**	**1**	needed letters
1	**4**	**9**	**1**	**6**	**3**	available letters

I need a allMatch() reducer!

Filtering V2

Live coding

Filtering V2

4th question: how to check if I have the available letters?

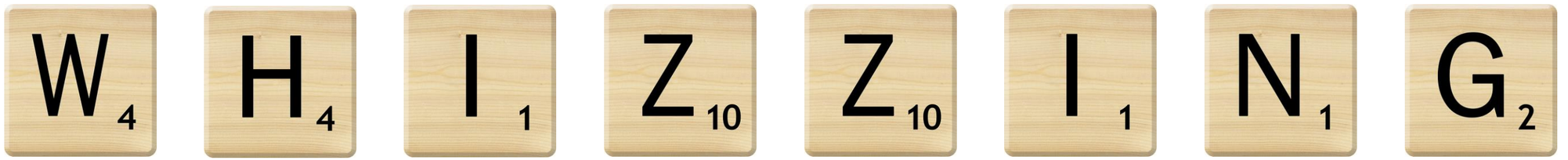
Solution: map / filter / reduce

Scrabble: use of blanks

5th question: and what about using blanks?

Scrabble: use of blanks

5th question: and what about using blanks?




```
private static final int [] scrabbleENDistribution = {
// a, b, c, d,  e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Scrabble: use of blanks

5th question: and what about using blanks?


```
private static final int [] scrabbleENDistribution = {
// a, b, c, d,  e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

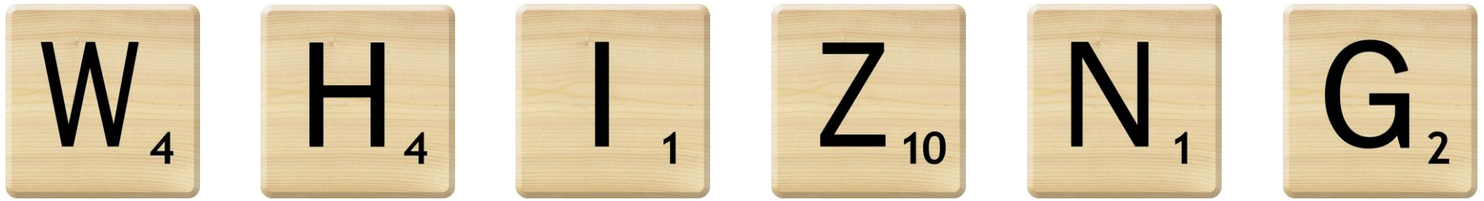
Scrabble: use of blanks

It has 2 impacts:

- on the filtering of the words
- on the computation of the score

Scrabble: use of blanks

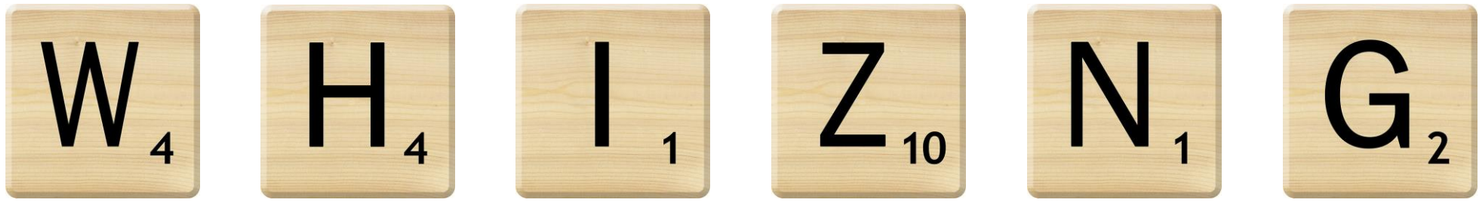
5th question: how to compute the score?



1 1 2 2 1 1

Scrabble: use of blanks

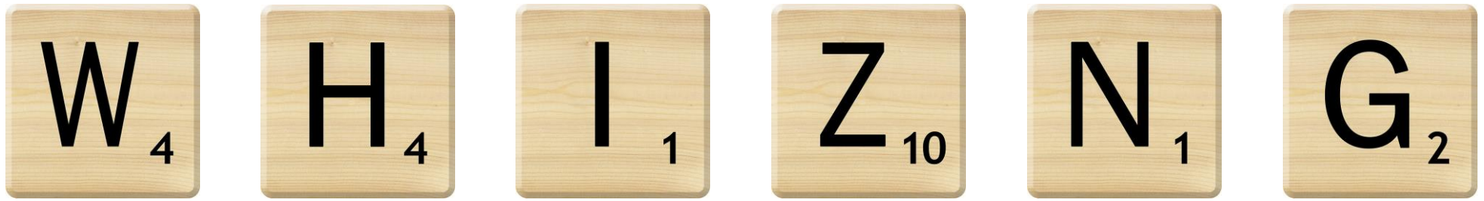
5th question: how to compute the score?



1	1	2	2	1	1
1	4	9	1	6	3

Scrabble: use of blanks

5th question: how to compute the score?



1	1	2	2	1	1	
1	4	9	1	6	3	
10	1	2	10	1	2	→ 26

Scrabble: use of blanks

Live coding

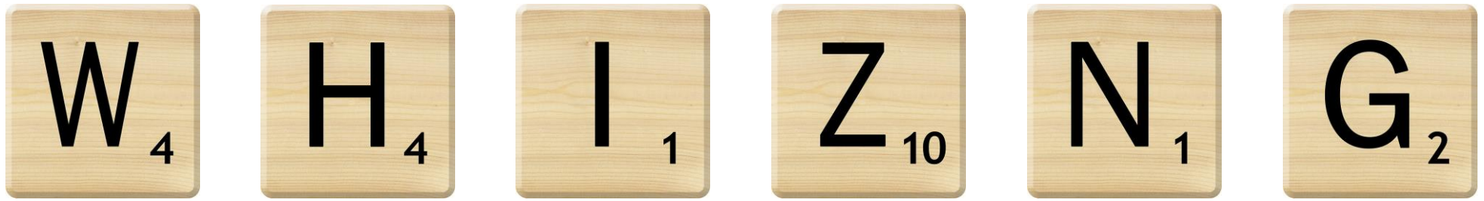
Scrabble: use of blanks

5th question: how to compute the score?

Solution: map / filter / reduce

Scrabble: use of blanks

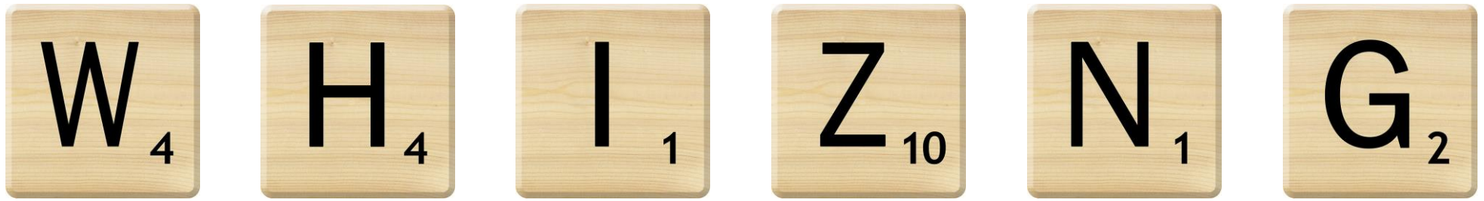
6th question: how many blanks do we need?



1 1 2 2 1 1

Scrabble: use of blanks

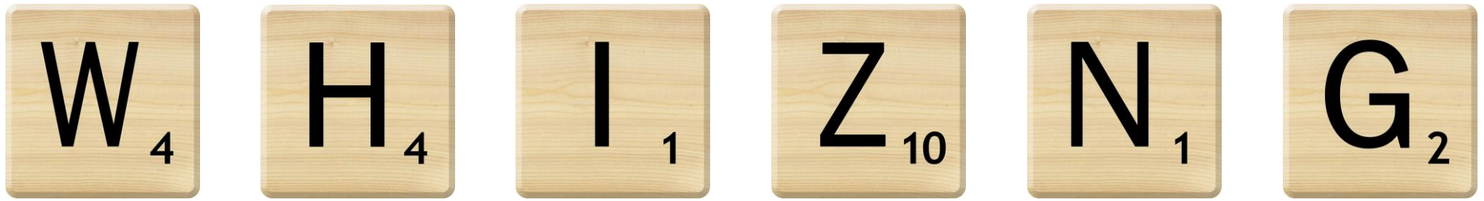
6th question: how many blanks do we need?



1	1	2	2	1	1
1	4	9	1	6	3

Scrabble: use of blanks

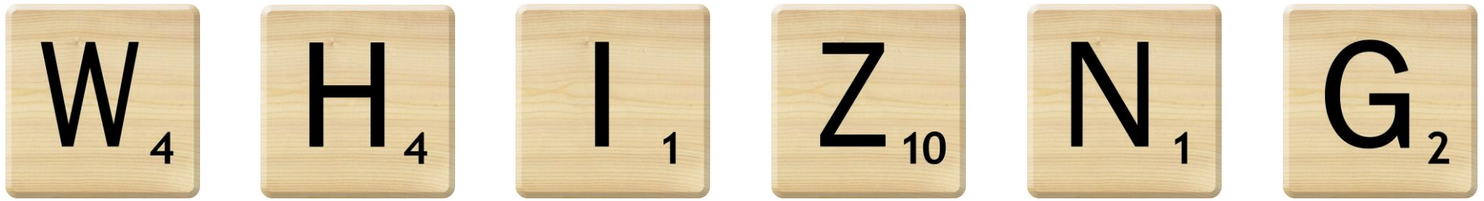
6th question: how many blanks do we need?



1	1	2	2	1	1
1	4	9	1	6	3
0	-3	-7	+1	-5	-2

Scrabble: use of blanks

6th question: how many blanks do we need?



1	1	2	2	1	1	
1	4	9	1	6	3	
0	0	0	+1	0	0	→ 1

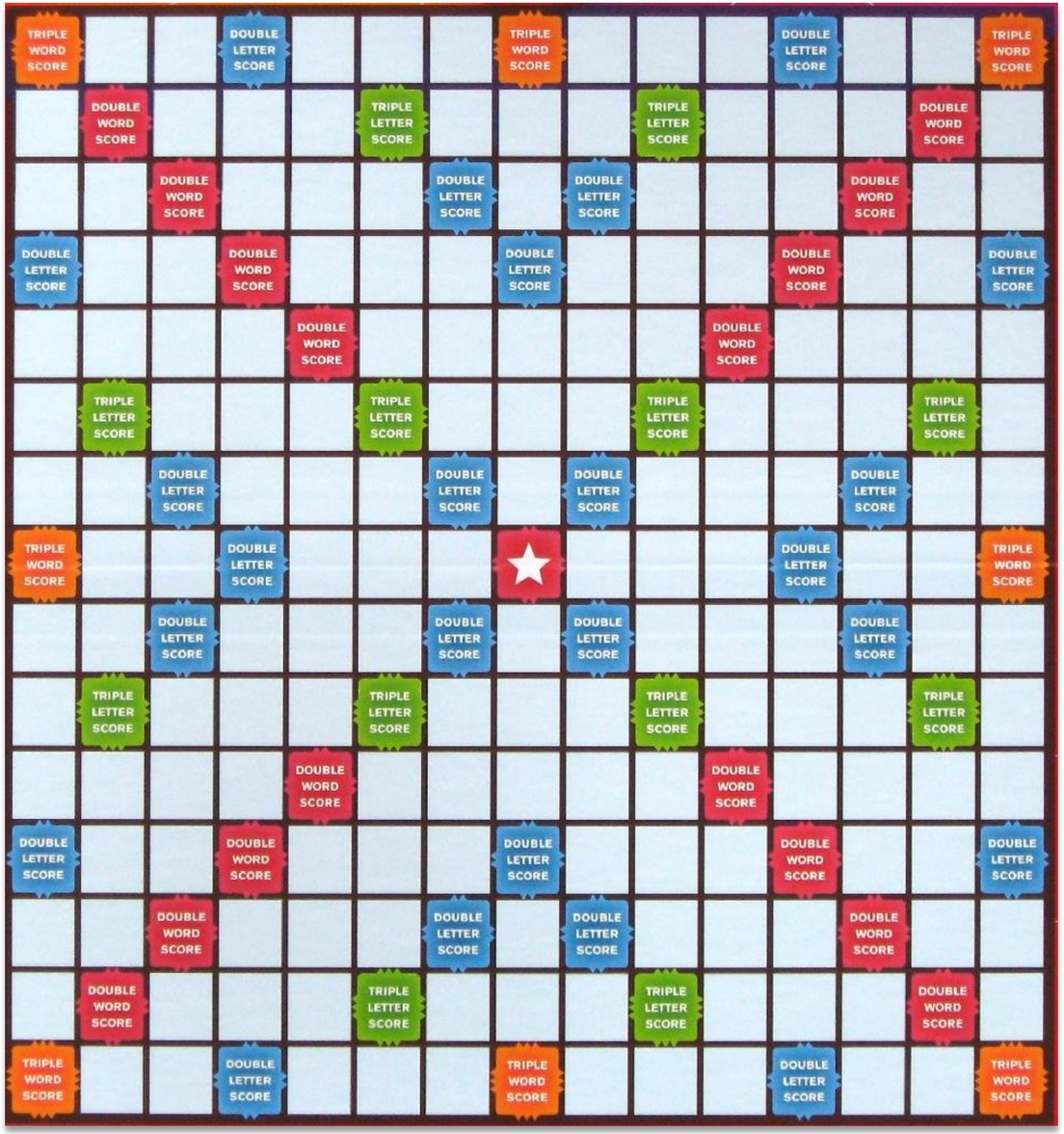
Scrabble: use of blanks

Live coding

Use of a blank

6th question: how many blanks do we need?

Solution: map / reduce


TRIPLE WORD SCORE
DOUBLE LETTER SCORE
DOUBLE WORD SCORE
TRIPLE LETTER SCORE

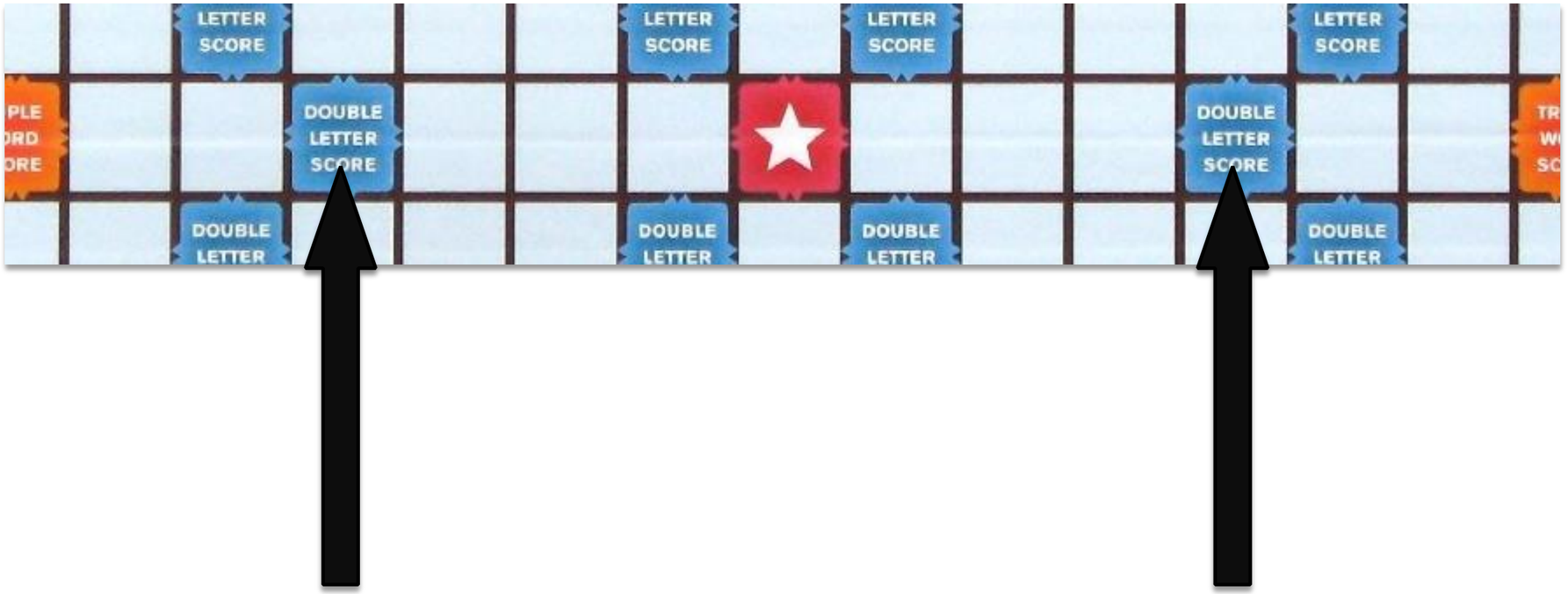

TRIPLE WORD SCORE
DOUBLE LETTER SCORE
DOUBLE WORD SCORE
TRIPLE LETTER SCORE
LETTER SCORE
DOUBLE LETTER SCORE

Scrabble: and what about the board?

7th question: what about the Double letter score?

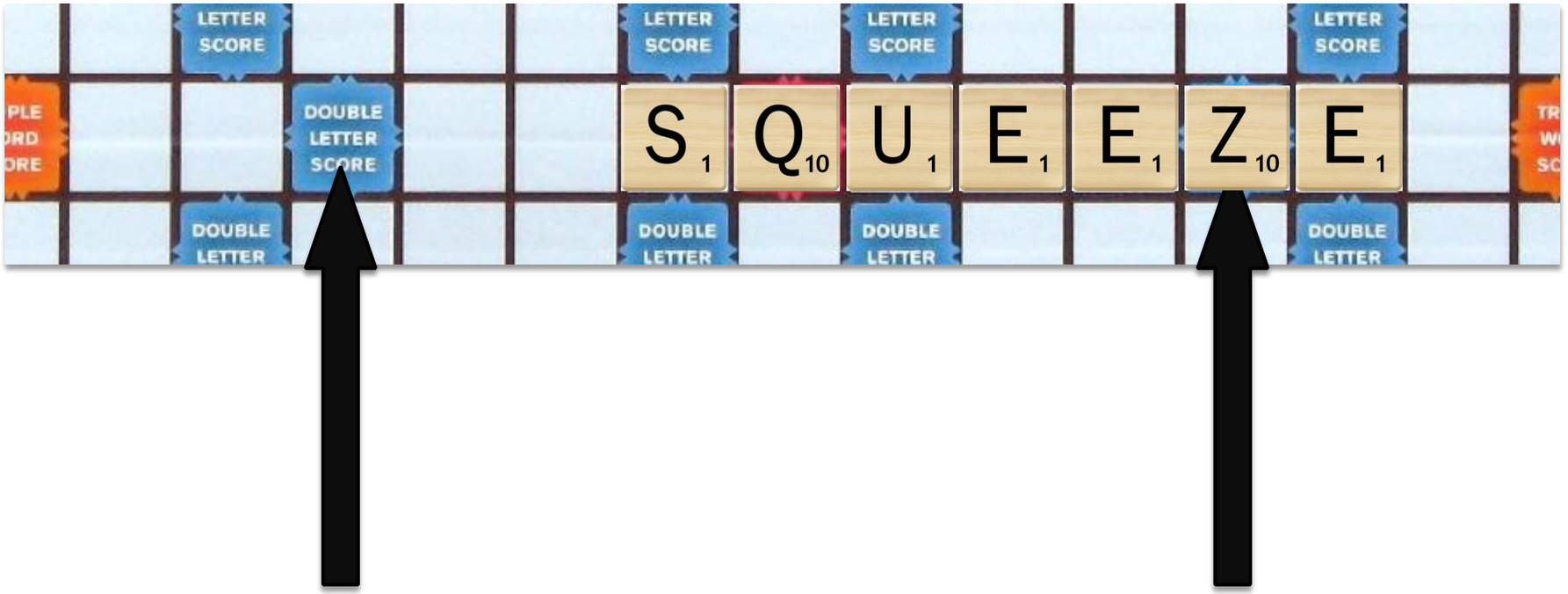
Scrabble: and what about the board?

7th question: what about the Double letter score?



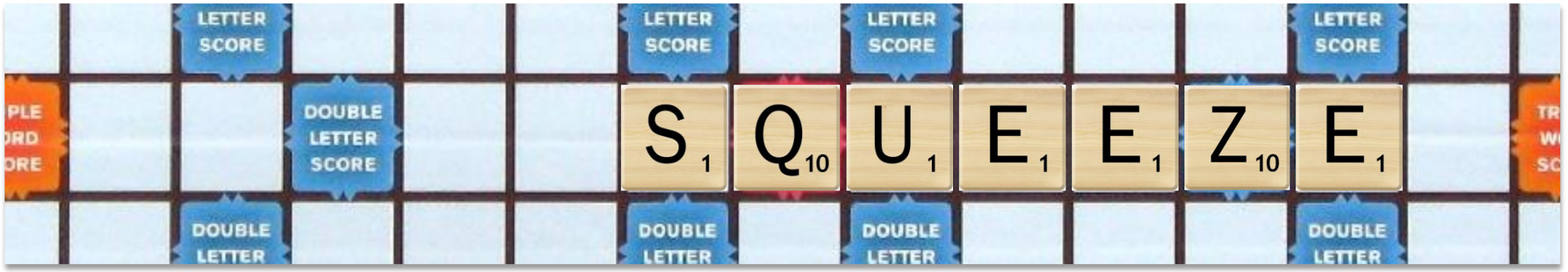
Scrabble: and what about the board?

7th question: what about the Double letter score?



Scrabble: and what about the board?

7th question: what about the Double letter score?

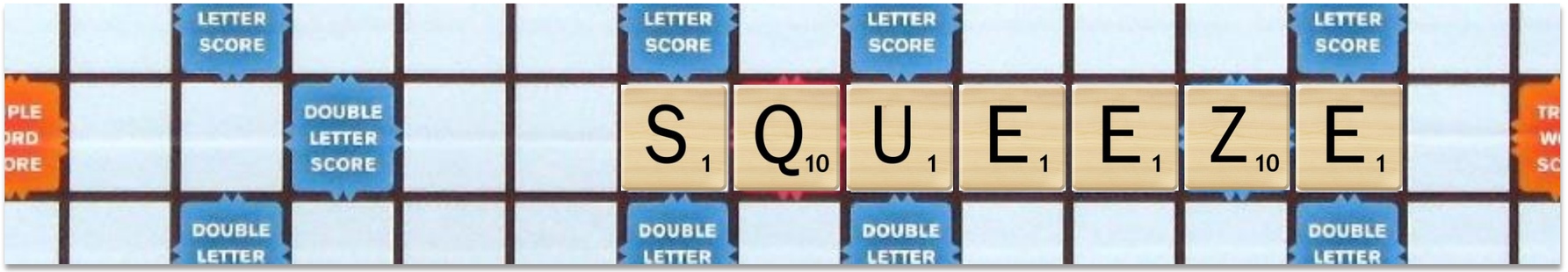


6 solutions to put the word:

- right square = 3 last letters
- left square= 3 first letters

Scrabble: and what about the board?

7th question: what about the Double letter score?



6 solutions to put the word:

- right square = 3 last letters
- left square= 3 first letters

If the word is long enough (7 letters)!

Scrabble: and what about the board?

7th question: what about the Double letter score?

So we need to take a max, on which set?

```
word.chars().skip(4)                                    // 1st stream
word.chars().limit(Integer.max(0, word.length() - 4)) // 2nd stream
```

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
IntStream.concat(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.map(scrabbleENScore)
.max()
```

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
IntStream.concat(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.map(scrabbleENScore)
.max()
```

Problem: concat does not parallel well

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
Stream.of(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
) // returns a Stream of Stream!
```

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
Stream.of(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMapToInt(Function.identity()) // chars() is an IntStream
.map(scrabbleENScore)
.max()                             // we have an IntStream!
```

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
Stream.of(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMapToInt(Function.identity()) // chars() is an IntStream
.map(scrabbleENScore)
.max()                             // we have an IntStream!
```

What should we do with this Optional?

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
Stream.of(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMapToInt(Function.identity()) // chars() is an IntStream
.map(scrabbleENScore)
.max()                             // we have an IntStream!
```

… that can be empty!

Scrabble: and what about the board?

7th question: what about the Double letter score?

```
Stream.of(
    word.chars().skip(4),
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMapToInt(Function.identity()) // chars() is an IntStream
.map(scrabbleENScore)
.max()                             // we have an IntStream!
.orElse(0) ;
```

… that can be empty!

Scrabble: and what about the board?

Live coding

Scrabble: and what about the board?

7th question: can we take the Double letter score?

Solution: map / reduce

https://github.com/JosePaumard/jdk8-lambda-tour

Conclusion

Streams + Collectors = no need to use the Iterator pattern any more

Conclusion

Streams + Collectors = no need to use the Iterator pattern any more

Combined with lambdas = new way of writing data processing and applications in Java

Conclusion

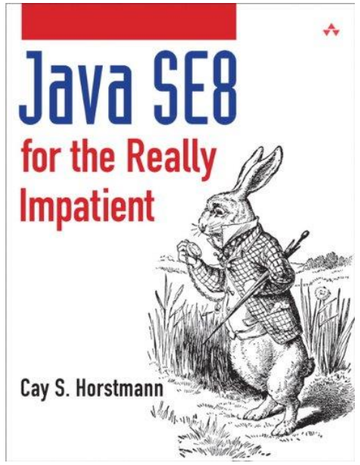
Streams + Collectors = no need to use the Iterator pattern any more

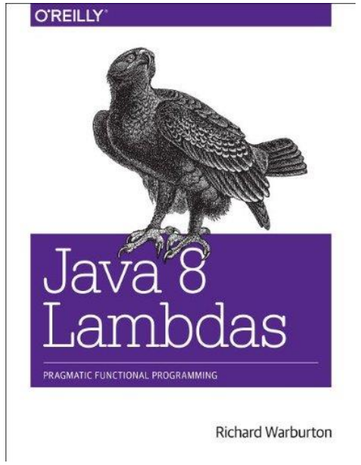
Combined with lambdas = new way of writing data processing in Java

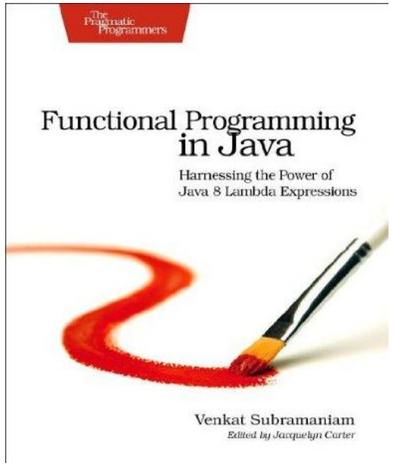
- writing code
- architecture

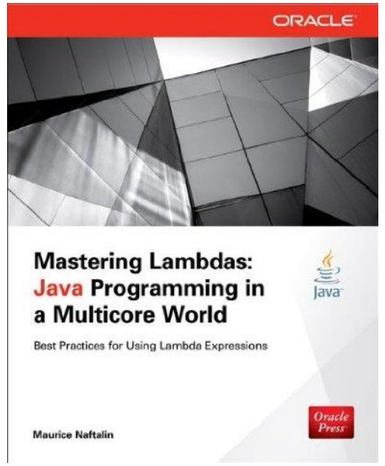
Conclusion

Some books:










DEVOXX
Thank you!
#Stream8
@JosePaumard


DEVOXX
Q/R
#Stream8
@JosePaumard