Autumn Collections

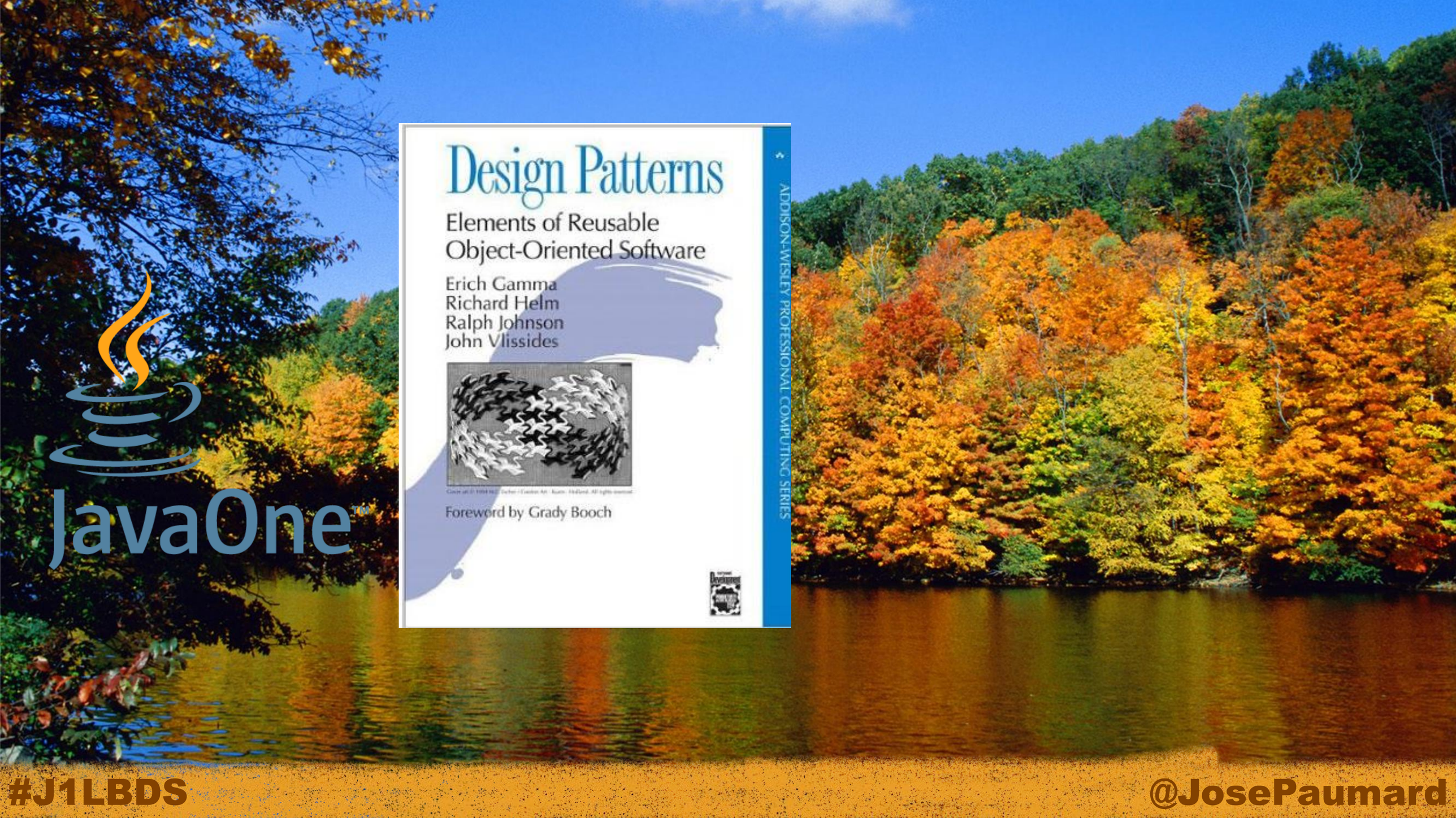
JavaOne℠

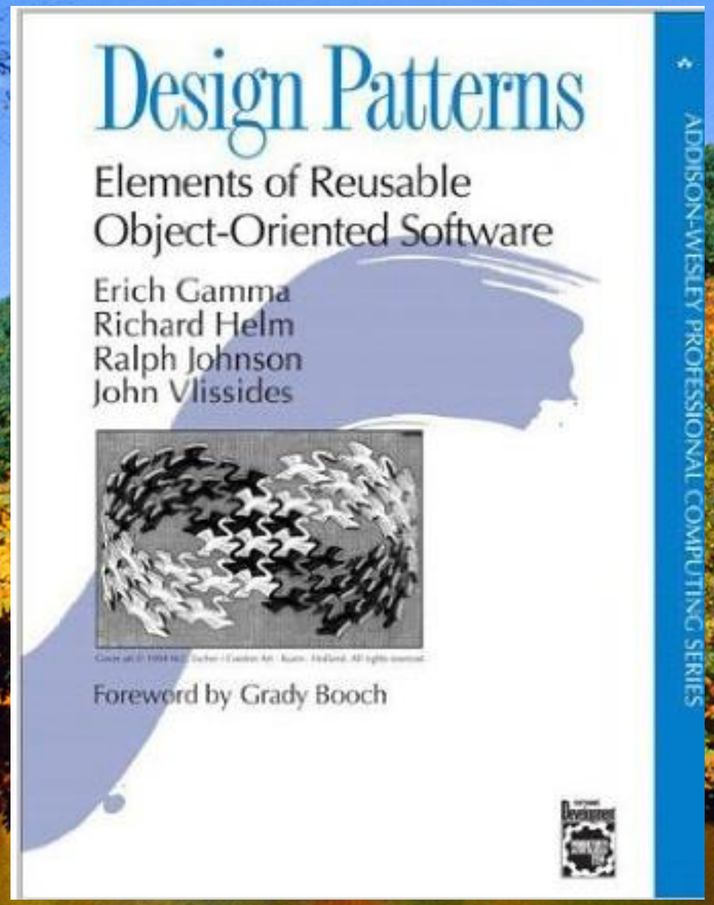
From Iterable to Lambdas,

Streams and Collectors

#J1LBDS

@JosePaumard

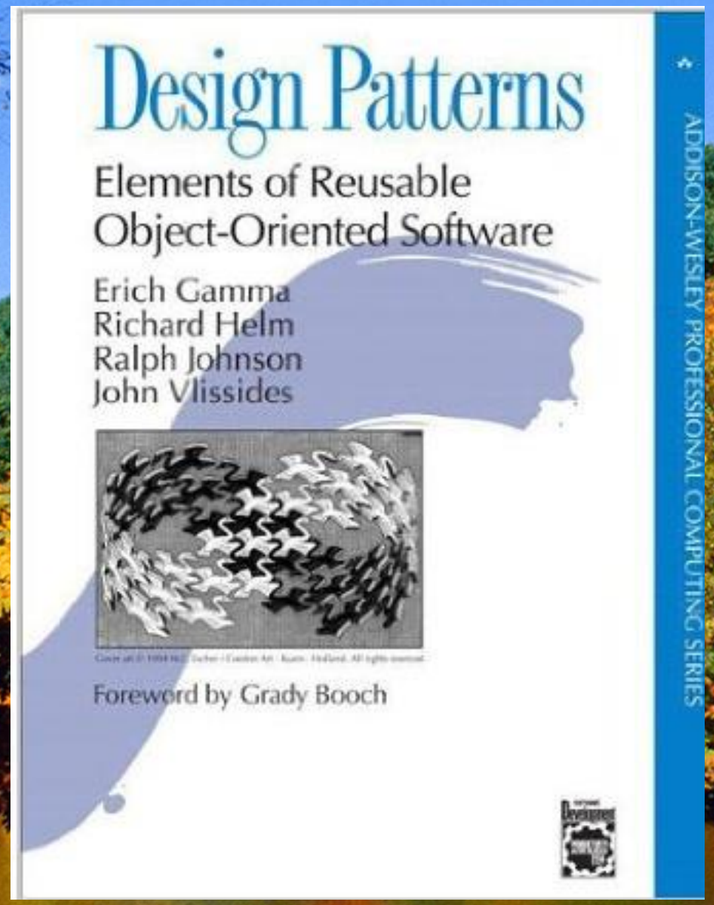

JavaOne
Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
#J1LBDS
@JosePaumard



Published

November 10, 1994

JavaOne℠



Published

November 10, 1994

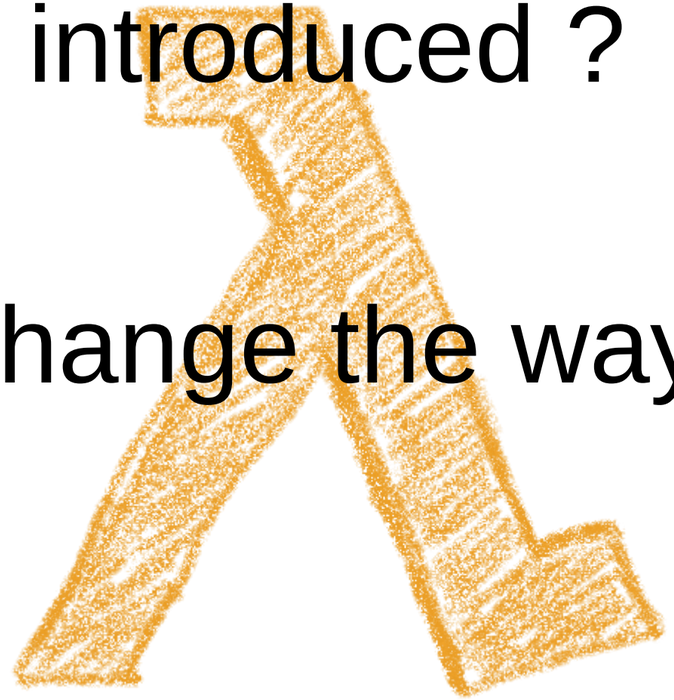
Still available hardover, paperback, multimedia CD

Why were they introduced ?

JavaOne™

Why were they introduced ?

How will they change the way we build applications ?



José PAUMARD

MCF Um. Paris 13

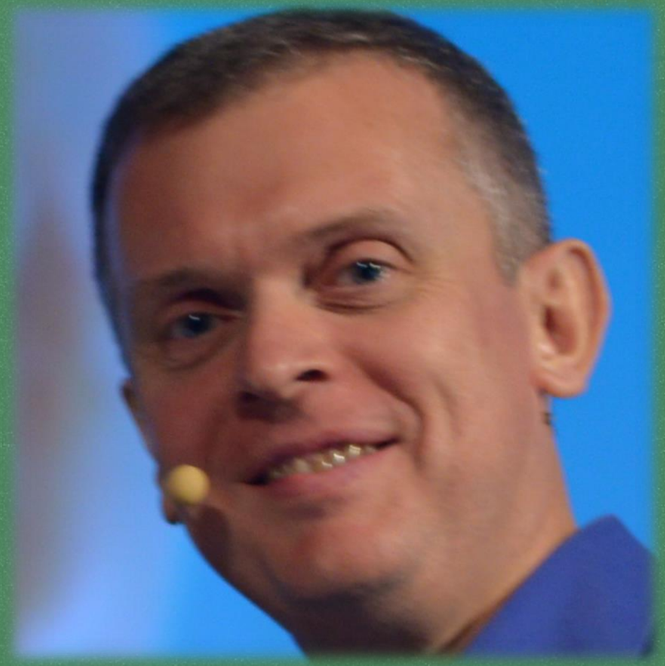
PhD Appl C.S.

Open source dev.

Independant

@JosePaumard

José PAUMARD



Open source dév.

Indépendant

@JosePaumard

Let’s introduce the lambdas

Let’s introduce the lambdas

on a simple example

A very simple example

```
public class Person {

   private String name ;
   private int age ;

   // constructors
   // getters / setters
}
```

A plain old bean…

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

… and a good old list

Average of the ages

```
int sum = 0 ;
// I need a default value in case
// the list is empty
int average = 0 ;
for (Person person : list) {
    sum += person.getAge() ;
}
if (!list.isEmpty()) {
    average = sum / list.size() ;
}
```

Trickier : average of the ages of people older than 20

```
int sum = 0 ;
int n = 0 ;
int average = 0 ;
for (Person person : list) {
    if (person.getAge() > 20) {
        n++ ;
        sum += person.getAge() ;
    }
}
if (n > 0) {
    average = sum / n ;
}
```

Trickier : average of the ages of people older than 20

```
int sum = 0 ;
int n = 0 ;
int average = 0 ;
for (Person person : list) {
    if (person.getAge() > 20) {
        n++ ;
        sum += person.getAge() ;
    }
}
if (n > 0) {
    average = sum / n ;
}
```

« imperative programming »

… it does not have to be like that !

```
select avg(age)
from Person
where age > 20
```

This is a description of the result

… it does not have to be like that !

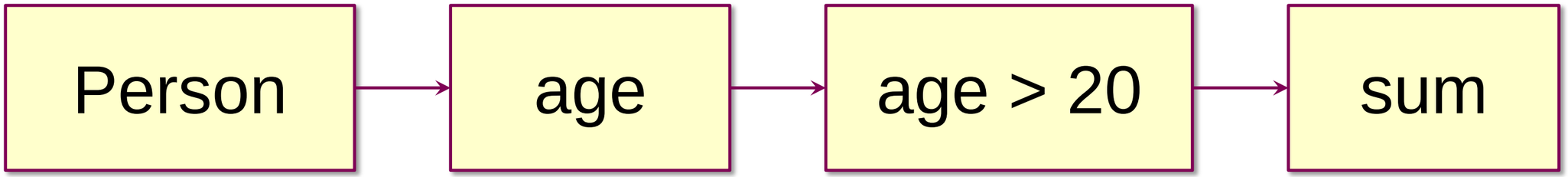
```
select avg(age)
from Person
where age > 20
```

© SQL language, 1974

This is a description of the result

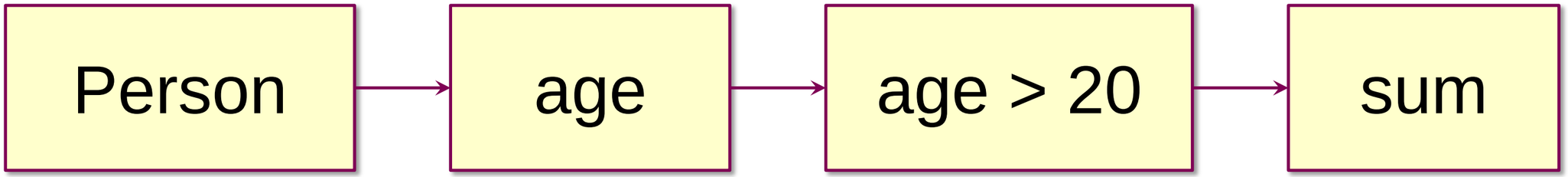
In that case, the DB server is free to compute the result the way it sees fit

map



1st step : mapping

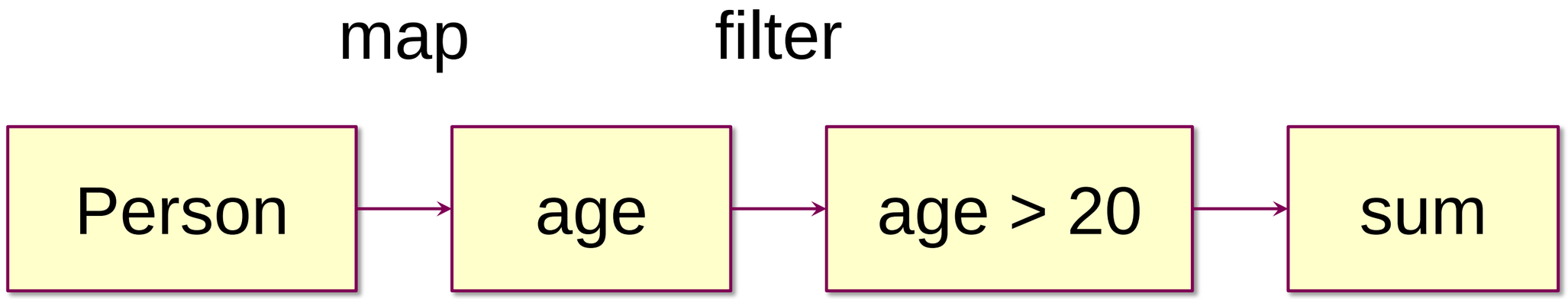
map



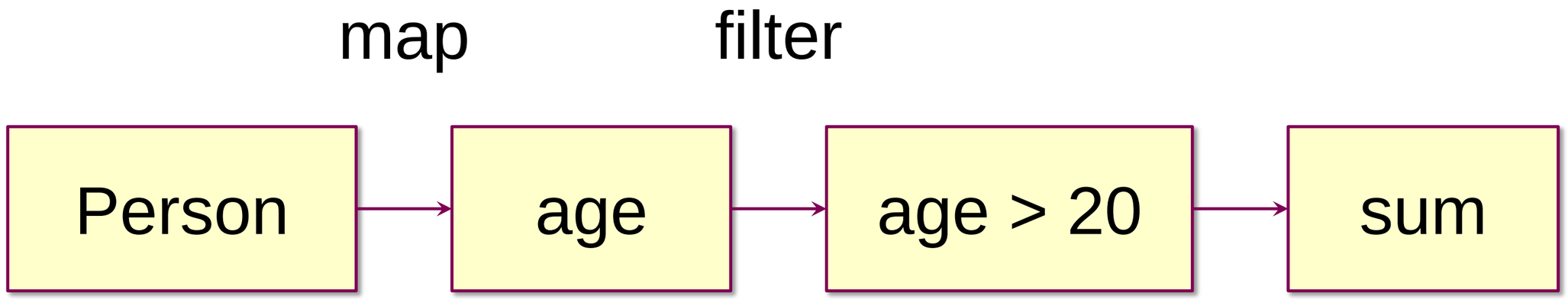
1st step : mapping

Mapping :

- takes a list of a given type
- gives another list of a different type
- same number of elements



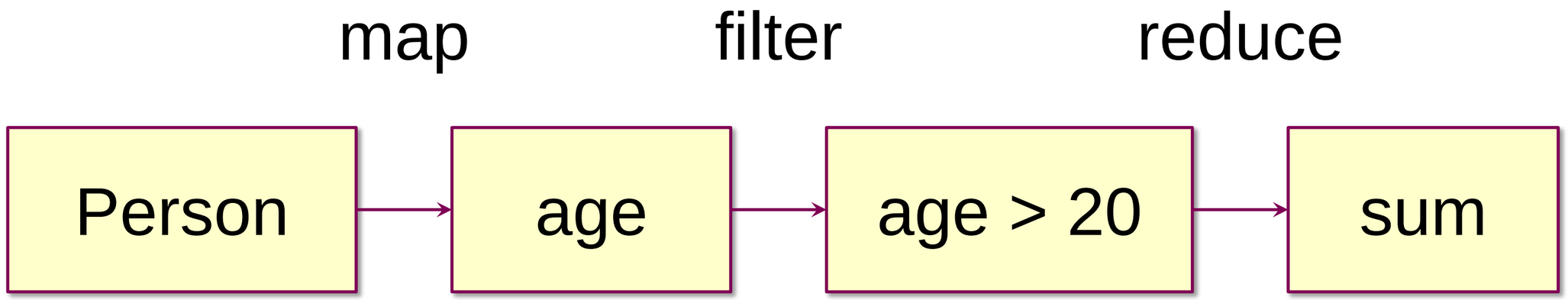
2nd step : filtering



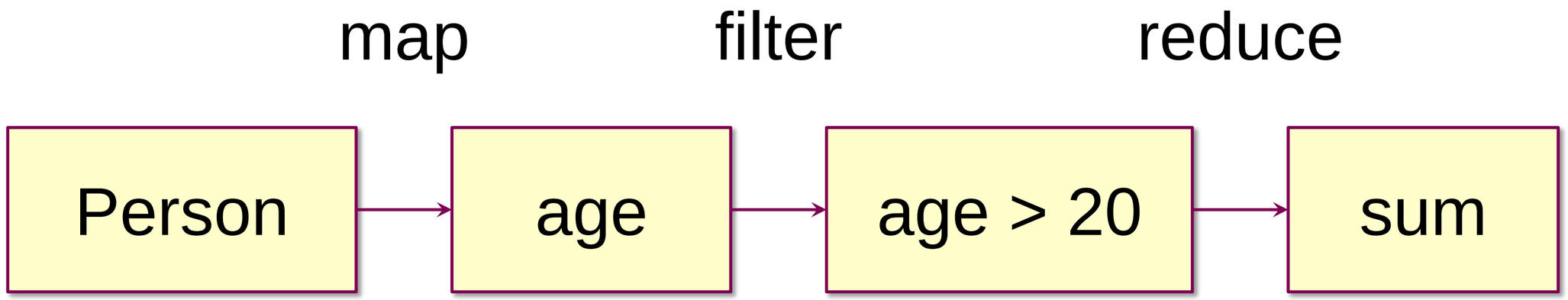
2nd step : filtering

Filtering :

- takes a list of a given type
- gives another list of a the same type
- less elements



3rd step : reduction



3rd step : reduction

Reduction : agregation of all the elements in a single one

Ex : average, sum, min, max, etc…

How can I model that ?

The JDK 7 way

Create an interface to model the mapper…

```
public interface Mapper<T, V> {
    public V map(T t) ;
}
```

The JDK 7 way

… and create an implementation

```
public interface Mapper<T, V> {
    public V map(T t) ;
}
```

```
Mapper<Person, Integer> mapper = new Mapper<Person, Integer>() {

    public Integer map(Person p) {
        return p.getAge() ;
    }
}
```

The JDK 7 way

We can do the same for filtering

```
public interface Predicate<T> {
   public boolean filter(T t) ;
}
```

```
Predicate<Integer> predicate = new Predicate<Integer>() {

   public boolean filter(Integer age) {
      return age > 20 ;
   }
}
```

The JDK 7 way

And for the reduction

```
public interface Reducer<T> {
   public T reduce(T t1, T t2) ;
}
```

```
Reducer<Integer> reduction = new Reducer<Integer>() {

   public Integer reduce(Integer i1, Integer i2) {
      return i1 + i2 ;
   }
}
```

The JDK 7 way

So the whole map / filter / reduce looks like this

1) Create 3 interfaces

```
public interface Mapper<T, V> {
    public V map(T t) ;
}
```

```
public interface Predicate<T> {
    public boolean filter(T t) ;
}
```

```
public interface Reducer<T> {
    public T reduce(T t1, T t2) ;
}
```

The JDK 7 way

So the whole map / filter / reduce looks like this

1) Create 3 interfaces
2) Apply the pattern

```
List<Person> persons = ... ;
int sum =
   persons.map(
      new Mapper<Person, Integer>() {

         public Integer map(Person p) {
            return p.getAge() ;
         }
      })
   .filter(
      new Filter<Integer>() {

         public boolean filter(Integer age) {
            return age > 20 ;
         }
      })
   .reduce(0,
      new Reducer<Integer>() {

         public Integer reduce(Integer i1, Integer i2) {
            return i1 + i2 ;
         }
      }
   }) ;
```

The JDK 7 way

So the whole map / filter / reduce looks like this

1) Create 3 interfaces
2) Apply the pattern

```
List<Person> persons = ... ;
int sum =
   persons.map(
      new Mapper<Person, Integer>() {

         public Integer map(Person p) {
            return p.getAge() ;
         }
      })
   .filter(
      new Filter<Integer>() {

         public boolean filter(Integer age) {
            return age > 20 ;
         }
      })
   .reduce(0,
      new Reducer<Integer>() {

         public Integer reduce(Integer i1, Integer i2) {
            return i1 + i2 ;
         }
      }
   }) ;
```

The JDK 8 way

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) {
        return person.getAge() ;
    }
}
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
    }
}
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
```

we take a person p

```
mapper = (Person person)
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
    }
}
```

and then …

```
mapper = (Person person) ->
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
    }
}
```

… return the age of person

```
mapper = (Person person) -> person.getAge() ;
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
    }
}
```

```
mapper = (Person person) -> person.getAge() ;
```

The JDK 8 way

Let’s rewrite our mapper

```
mapper = new Mapper<Person, Integer>() {
    public Integer map(Person person) { // 1 method
        return person.getAge() ;
    }
}
```

```
mapper = (Person person) -> person.getAge() ;
```

The compiler can recognize this as an implementation of Mapper

Some corner cases

What if …

… there is more than one statement ?

```
mapper = (Person person) -> {
    System.out.println("Mapping " + person) ;
    return person.getAge() ;
}
```

The return has to be explicit

What if …

…I have no return value ?

```
consumer = (Person person) -> p.setAge(p.getAge() + 1) ;
```

What if …

…I have more than one argument ?

```
reducer = (int i1, int i2) -> {
    return i1 + i2 ;
}
```

Or :

```
reducer = (int i1, int i2) -> i1 + i2 ;
```

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

1) mapper is of type Mapper

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

1) mapper is of type Mapper
2) There is only one method in Mapper

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

1) mapper is of type Mapper
2) There is only one method in Mapper
3) Both the parameters and the return types are compatible

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

1) mapper is of type Mapper
2) There is only one method in Mapper
3) Both the parameters and the return types are compatible
4) Thrown exceptions (if any are compatible)

The JDK 8 way

How can the compiler recognize the implementation of map() ?

```
mapper = (Person person) -> person.getAge() ;
```

1) mapper is of type Mapper
2) There is only one method in Mapper
3) Both the parameters and the return types are compatible
4) Thrown exceptions (if any are compatible)

More lambdas

Writing more lambdas becomes natural :

```
mapper  = (Person person) -> person.getAge() ; // mapper
filter  = (int age) -> age > 20 ;              // filter
reducer = (int i1, int i2) -> i1 + i2 ;        // reducer
```

More lambdas

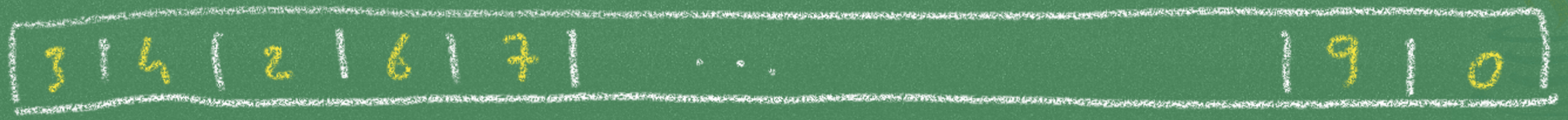
And most of the time, the compiler understands this :

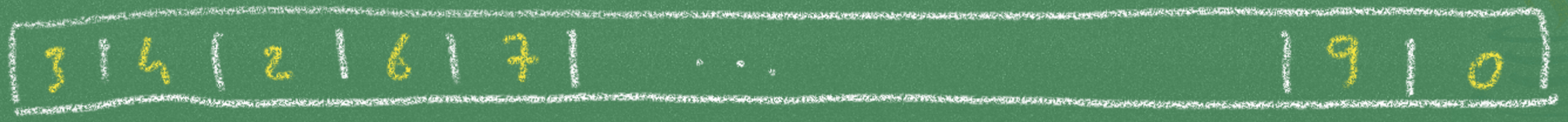
```
mapper  = person -> person.getAge() ;          // mapper
filter  = age -> age > 20 ;                    // filter
reducer = (i1, i2) -> i1 + i2 ;                // reducer
```

The « parameter types » can be omitted

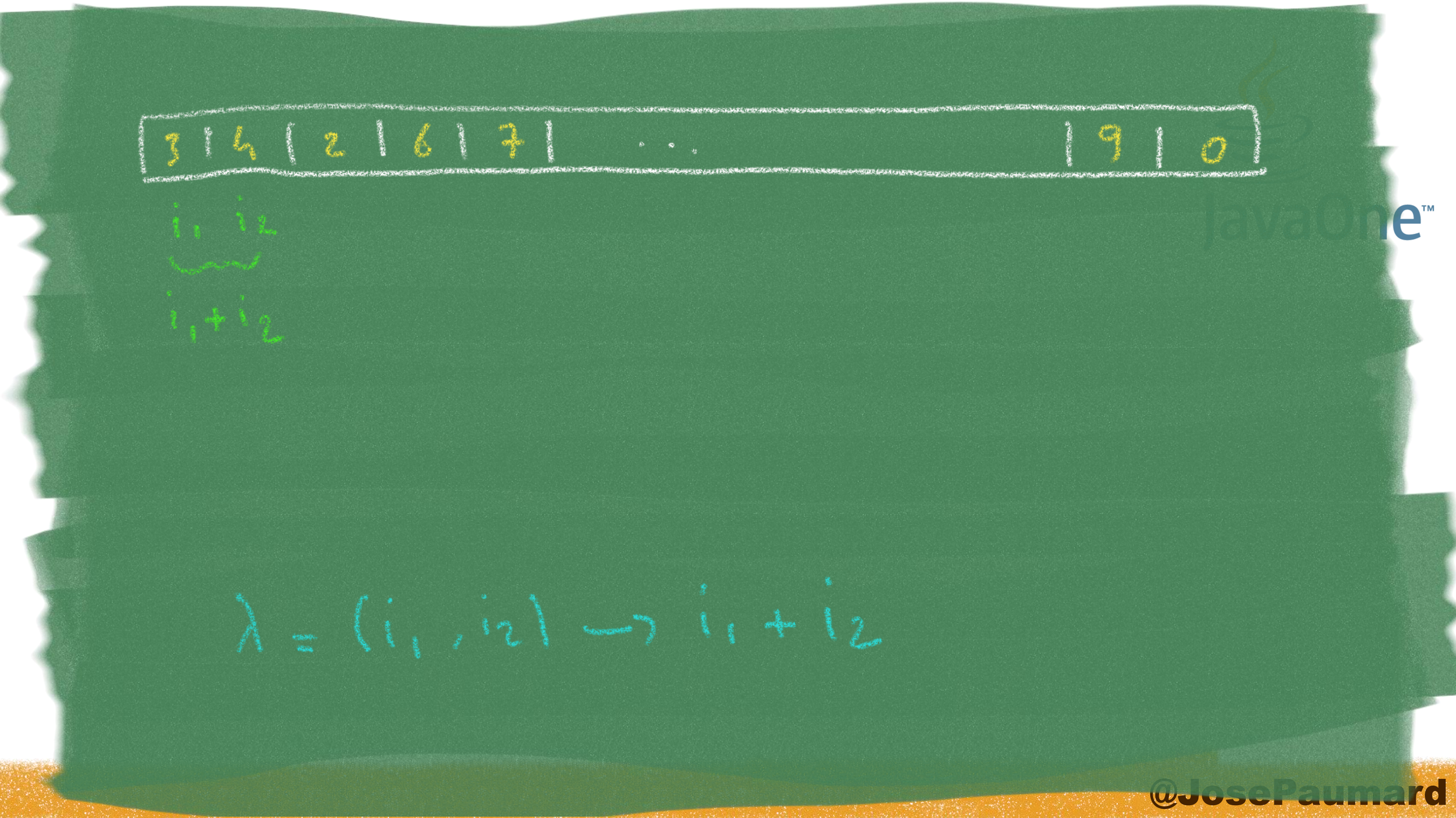
Just a remark on the reduction

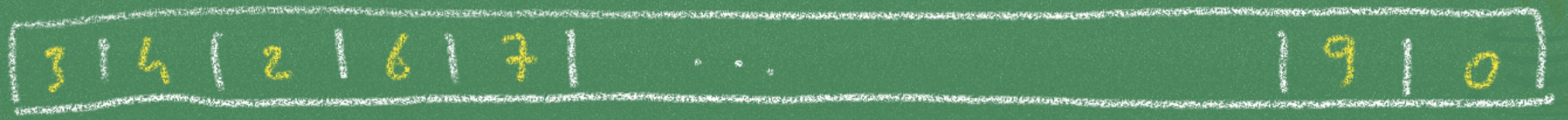
How does it really work ?





$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$


3 1 4 1 2 1 6 1 7 1 . . . 1 9 1 0
i1 i2
i1+i2
λ = (i1 , i2) → i1 + i2
JavaOne™
@JosePaumard



i_1 i_2

$i_1 + i_2$

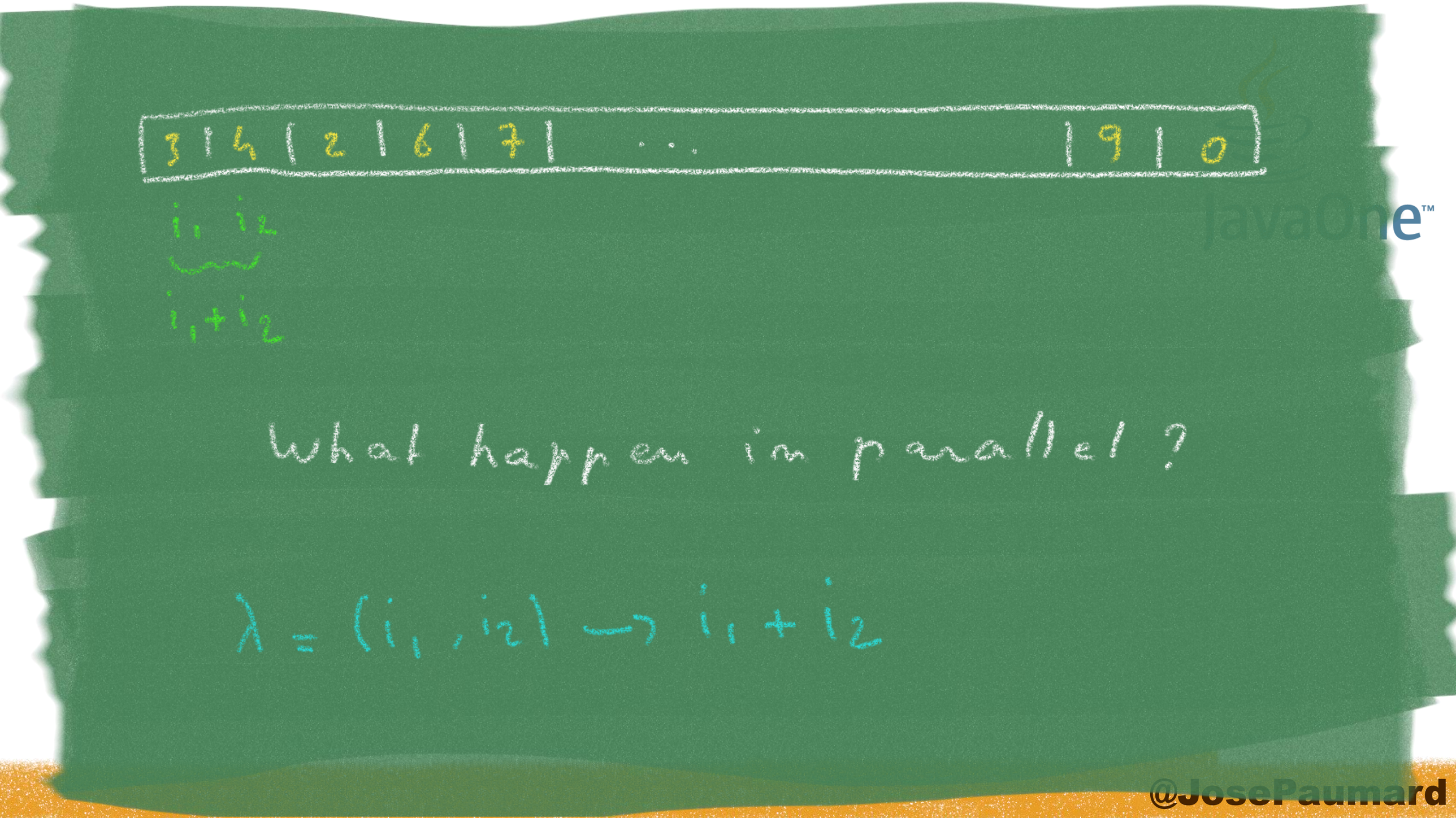
STEP 1

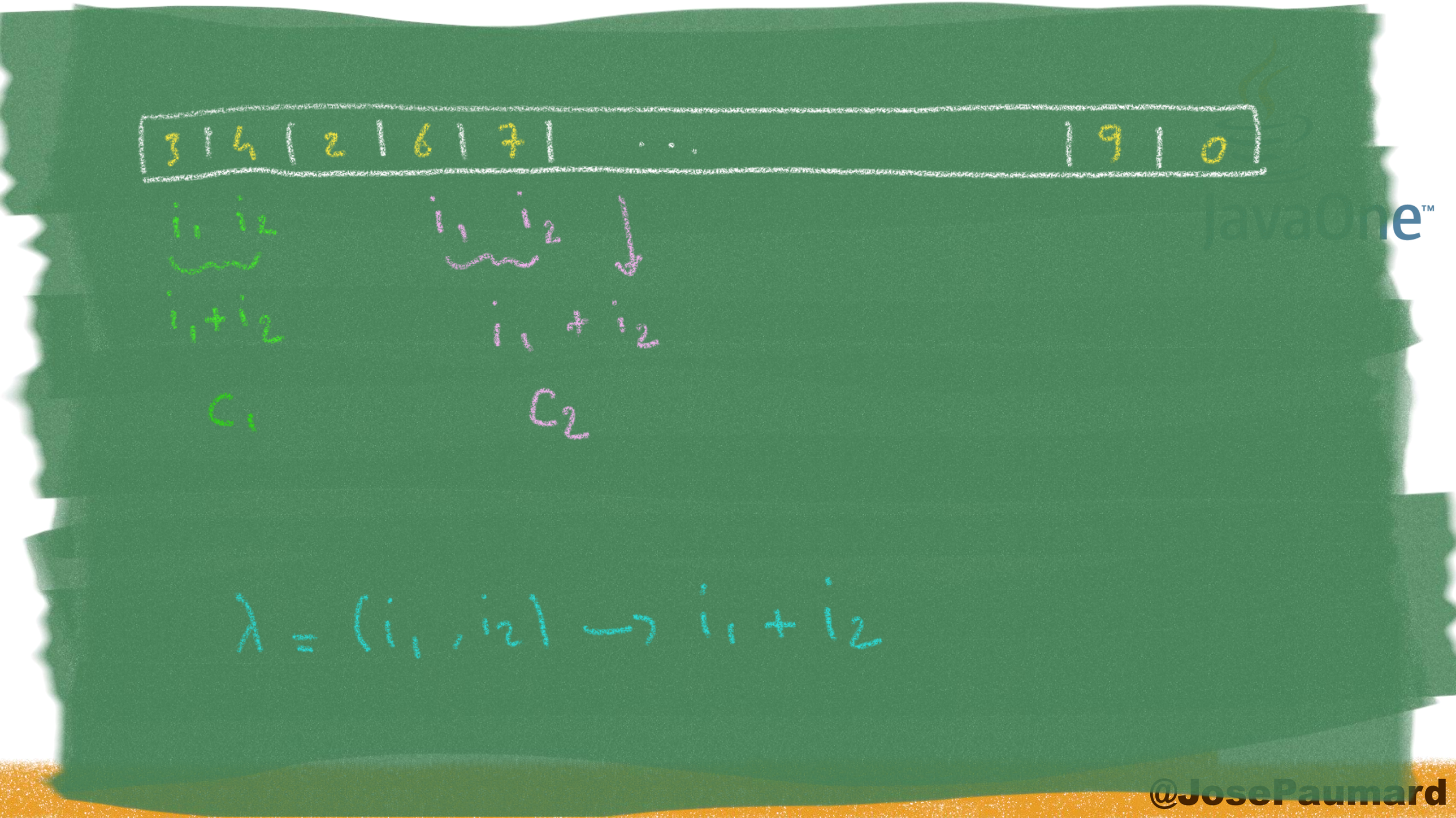
$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

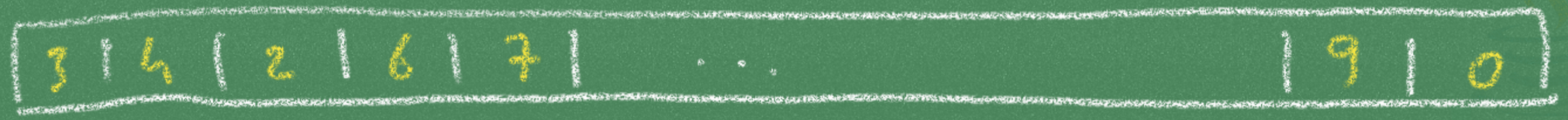
| 3 | 1 | 4 | 2 | 1 | 6 | 1 | 7 | ... | 9 | 0 |

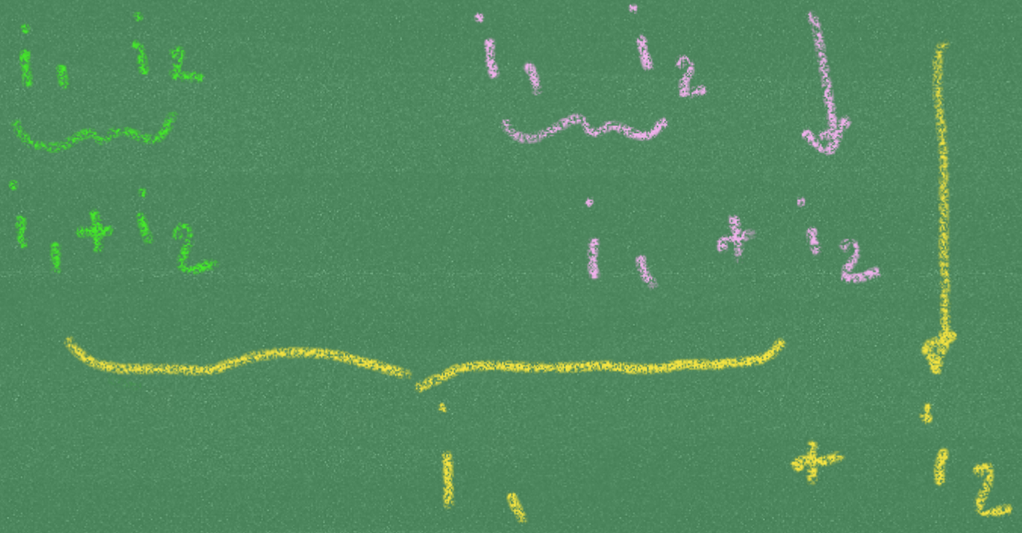
STEP 2

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

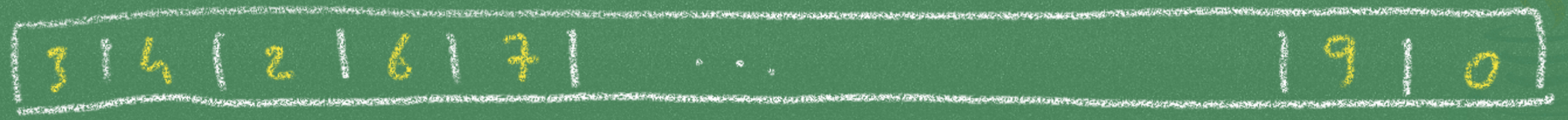

3 1 4 1 2 1 6 1 7 1 . . . 1 9 1 0
i1 i2
i1+i2
What happen in parallel ?
λ = (i1 , i2) → i1 + i2
JavaOne™
@JosePaumard







$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$



$i_1\ i_2$ $\quad$ $i_1\ i_2$

$i_1 + i_2$ $\quad$ $i_1 + i_2$

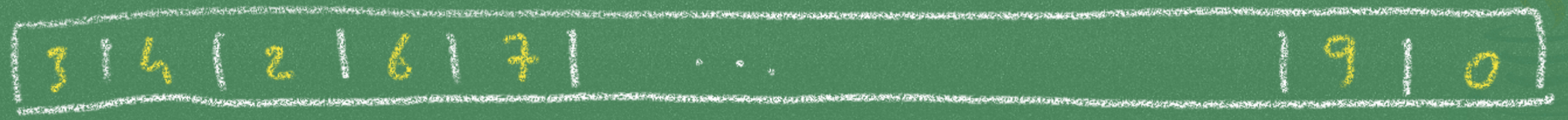
$i_1 \quad + \quad i_2$

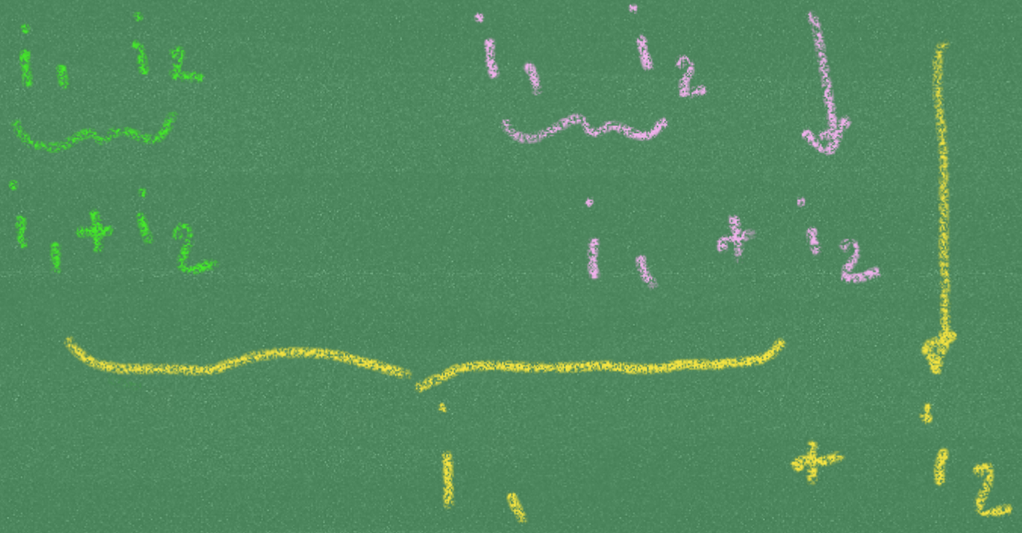
$\Rightarrow i_1 + (i_2 + i_3)$

$=$

$(i_1 + i_2) + i_3$

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$





$$\Rightarrow \begin{array}{l} Red(i_1, Red(i_2, i_3)) \\ = \\ Red(Red(i_1, i_2), i_3) \end{array}$$

$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$

Reduction

2 examples :

```
Reducer r1 = (i1, i2) -> i1 + i2 ;                  // Ok
Reducer r2 = (i1, i2) -> i1*i1 + i2*i2 ;            // Oooops
Reducer r3 = (i1, i2) -> (i1 + i2)*(i1 + i2) ;      // Aaaargh
```

Reduction

2 examples :

```
Reducer r1 = (i1, i2) -> i1 + i2 ;                  // Ok
Reducer r2 = (i1, i2) -> i1*i1 + i2*i2 ;            // Oooops
Reducer r3 = (i1, i2) -> (i1 + i2)*(i1 + i2) ;      // Aaaargh
Reducer r4 = (i1, i2) -> (i1 + i2) / 2 ;            // Ouch !!
```

Caveat :

the result is always reproductible in serial

it’s not in parallel

So far

A lambda expression is an alternative
to write instances of anonymous inner classes

Other syntaxes

Very often one writes

```
mapper = person -> person.getAge() ;
```

Other syntaxes

Very often one writes

```
mapper = person -> person.getAge() ;
```

But this syntax is also possible :

```
mapper = Person::getAge ; // non static method
```

« method reference »

Other syntaxes

Other example :

```
sum = (i1, i2) -> i1 + i2 ;
sum = Integer::sum ; // static method, new !
```

Or :

```
max = (i1, i2) -> i1 > i2 ? i1 : i2 ;
max = Integer::max ; // static method, new !
```

Other syntaxes

Other example :

```
toLower = String::toLowerCase ;
```

```
// !!!! NO NO NO !!!!
toLowerFR = String::toLowerCase(Locale.FRANCE) ;
```

More on the lambdas

3 Questions about lambdas :

What model for a lambda ?

3 Questions about lambdas :

What model for a lambda ?

Can I put a lambda in a variable ?

3 Questions about lambdas :

What model for a lambda ?

Can I put a lambda in a variable ?

Is a lambda an object ?

Modelization

A lambda is an instance of a « functional interface »

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

   public void accept(T t) ;
}
```

Modelization

A lambda is an instance of a « functional interface »

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

   public void accept(T t) ;
}
```

- only one abstract method (methods from Object dont count)
- can be annotated by @FunctionalInterface (optional)

Putting a lambda in a variable

Example of a consumer

```
Consumer<String> c = new Consumer<String>() {

    @Override
    public void accept(String s) {
        System.out.println(s) ;
    }
} ;
```

So :

```
Consumer<String> c = s -> System.out.println(s) ;
```

Putting a lambda in a variable

Example of a consumer

```
Consumer<String> c = new Consumer<String>() {

    @Override
    public void accept(String s) {
        System.out.println(s) ;
    }
} ;
```

So :

```
Consumer<String> c = System.out::println ;
```

Questions :

What model for a lambda ?

answer : with a functional interface

Can I put a lambda in a variable ?

answer : yes

Is a lambda an object ?

Is a lambda an objet ?

Let’s play the game of 7 errors (with only 1 error)

```
Consumer<String> c = new Consumer<String>() {

    @Override
    public void accept(String s) {
        System.out.println(s) ;
    }
} ;
```

```
Consumer<String> c = s -> System.out.println(s) ;
```

Is a lambda an objet ?

Let’s play the game of 7 errors (with only 1 error)

```
Consumer<String> c = new Consumer<String>() {

   @Override
   public void accept(String s) {
      System.out.println(s) ;
   }
} ;
```

```
Consumer<String> c = s -> System.out.println(s) ;
```

Questions :

What model for a lambda ?

answer : with a functionnal interface

Can I put a lambda in a variable ?

answer : yes

Is a lambda an object ?

answer : no

Plenty of lambdas : java.util.function

Java.util.functions

This new package holds the functionnal interfaces

There are 43 of them

Java.util.functions

Supplier : alone in its kind

Consumer / BiConsumer

Function / BiFunction (UnaryOperator / BinaryOperator)

Predicate / BiPredicate

Plus the primitive types versions

Supplier

A supplier supplies an object

```
public interface Supplier<T> {

    T get() ;
}
```

Consumer

A consumer just … accepts an object

```
public interface Consumer<T> {

    void accept(T t) ;
}
```

Function

A function takes an object and returns another one

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

    R apply(T t) ;
}
```

Can be chained and / or composed

Predicate

A Predicate takes an object and returns a boolean

```
public interface Predicate<T> {

    boolean test(T t) ;
}
```

Can be negated, composed with and / or

Back to the map / filter / reduce pattern

The JDK 7 way

How to apply map / filter / reduce to `List<Person>` ?

The JDK 7 way

How to apply map / filter / reduce
to `List<Person>` ?

The legal way is to iterate over the elements
and apply the pattern

The JDK 7 way

How to apply map / filter / reduce
to `List<Person>` ?

The legal way is to iterate over the elements
and apply the pattern

One can create a helper method

Applying map to a List

With a helper method, JDK 7

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

List<Integer> ages = // ages is a new list
   Lists.map(
      persons,
      new Mapper<Person, Integer>() {
         public Integer map(Person p) {
            return p.getAge() ;
         }
      }
   ) ;
```

Applying map to a List

With a helper method, JDK 8 & lambdas

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

List<Integer> ages =
   Lists.map(
      persons,
      person -> person.getAge()
   ) ;
```

Applying map to a List

With a helper method, JDK 8 & lambdas

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

List<Integer> ages =
   Lists.map(
      persons,
      Person::getAge
   ) ;
```

Putting things together

The map / filter / reduce pattern would look like this

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Putting things together

The map / filter / reduce pattern would look like this

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

The idea is to push lambdas to the API, and let it apply them on its content

Putting things together

The map / filter / reduce pattern would look like this

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Pro : the API can be optimized
without touching the code : great !

Putting things together

The map / filter / reduce pattern would look like this

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Pro : the API can be optimized
without touching the code

Cons …

Putting things together

1) Suppose persons is a really BIG list

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(    persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Putting things together

1) Suppose persons is a really BIG list

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(    persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

2 duplications : ages & agesGT20

Putting things together

1) Suppose persons is a really BIG list

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(    persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

2 duplications : ages & agesGT20

What do we do with them ? Send them to the GC

Putting things together

2) Suppose we have a allMatch() reducer

```
// applying the map / filter / reduce pattern
List<Person> persons = ... ;
List<String> names =
   Lists.map(persons, p -> p.getName()) ;
boolean allMatch =
   Lists.allMatch(names, n -> n.length() < 20) ;
```

allMatch is true if all the names are shorter than 20 chars

let’s see a possible implementation of allMatch()

How could one write allMatch ?

Here is a basic implementation of allMatch

```
public static <T> boolean allMatch(
    List<? extends T> list, Filter<T> filter) {

    for (T t : list) {
        if (!filter.filter(t)) {
            return false ;
        }
    }

    return true ;
}
```

How could one write allMatch ?

Here is a basic implementation of allMatch

```
public static <T> boolean allMatch(
    List<? extends T> list, Filter<T> filter) {

    for (T t : list) {
        if (!filter.filter(t)) {
            return false ;
        }
    }

    return true ;
}
```

No need to iterate over the whole list

Putting things together

When we apply the allMatch() reducer…

```
// applying the map / filter / reduce pattern
List<Person> persons = ... ;
List<String> names =
   Lists.map(persons, p -> p.getName()) ;
boolean allMatch =
   Lists.allMatch(names, n -> n.length() < 20) ;
```

… the list names has already been evaluated !

Putting things together

When we apply the allMatch() reducer…

```
// applying the map / filter / reduce pattern
List<Person> persons = ... ;
List<String> names =
   Lists.map(persons, p -> p.getName()) ;
boolean allMatch =
   Lists.allMatch(names, n -> n.length() < 20) ;
```

… the list names has already been evaluated !

We lost a big opportunity for optimization !

Putting things together

When we apply the allMatch() reducer…

```
// applying the map / filter / reduce pattern
List<Person> persons = ... ;
List<String> names =
   Lists.map(persons, p -> p.getName()) ;
boolean allMatch =
   Lists.allMatch(names, n -> n.length() < 20) ;
```

… we should have wait to apply the filter

A good way to do that : apply the filter step lazily !

Conclusion

Pro : 1

Cons : 3 (at least)

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Conclusion

Pro : 1

Cons : 3 (at least)

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

Conclusion

And the same goes for this one

```
// applying the map / filter / reduce pattern
List<Person>  persons  = ... ;
List<Integer> ages     = Lists.map(   persons,  p -> p.getAge()) ;
List<Integer> agesGT20 = Lists.filter(ages,     a -> a > 20) ;
int sum                = Lists.reduce(agesGT20, Integer::sum) ;
```

What about « in place » operations ?

Not possible for the map step, because the type of the list changes

Could be evaluated for the filter, with concurrency issues

Doesnt make sense for the reduction

Conclusion (again)

We need a new concept to handle BIG lists efficiently

Conclusion (again)

We need a new concept to handle BIG lists efficiently

The Collection framework is not the right one…

Conclusion (again)

We need a new concept to handle BIG lists efficiently

The Collection framework is not the right one

We need something else !

So what pattern
to choose ?

Introduction

Putting map / filter / reduce methods on Collection would have led to :

```
// map / filter / reduce pattern on collections
int sum = persons
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

And even if it doesnt lead to viable patterns,

it is still a pleasant way of writing things

Introduction

Let’s keep the same kind of pattern and add a stream()

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Introduction

Collection.stream() returns a Stream : new interface

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

New interface = our hands are free !

New Collection interface

So we need a new method on Collection

```
public interface Collection<E> {
    // our good old methods

    Stream<E> stream() ;
}
```

New Collection interface

Problem : ArrayList doesnt compile anymore…

Something has to be done !

```
public interface Collection<E> {
    // our good old methods

    Stream<E> stream() ;
}
```

Interfaces

Problem : ArrayList doesnt compile anymore…

Something has to be done !

Something that does not need to change or recompile all the existing implementations of Collection

New interfaces

Java 8 interfaces

Problem : ArrayList doesnt compile anymore…

Something has to be done !

Something that does not need to change or recompile all the existing implementations of Collection

Problem : how to add methods to an interface, without changing any of the implementations ?

Java 8 interfaces

Problem : how to add methods to an interface, without changing any of the implementations ?

Solution : change the way interfaces work in Java

Java 8 interfaces

ArrayList needs to know the implementation of stream()…

```
public interface Collection<E> {
    // our good old methods

    Stream<E> stream() ;
}
```

Java 8 interfaces

Let’s put it in the interface !

```
public interface Collection<E> {
    // our good old methods

    default Stream<E> stream() {
        return ... ;
    }
}
```

Java 8 interfaces

Let’s introduce *default methods* in Java

```
public interface Collection<E> {
    // our good old methods

    default Stream<E> stream() {
        return ... ;
    }
}
```

Default methods are about interface evolution

Allows the extension of old interfaces

Default methods

Does it bring multiple inheritance to Java ?

Default methods

Does it bring multiple inheritance to Java ?

Yes ! But we already have it

Default methods

Does it bring multiple inheritance to Java ?

Yes ! But we already have it

```
public class String
implements Serializable, Comparable<String>, CharSequence {

    // ...
}
```

Default methods

What we have so far is multiple inheritance of *type*

Default methods

What we have so far is multiple inheritance of *type*

What we get in Java 8 is multiple inheritance of *behavior*

Default methods

What we have so far is multiple inheritance of *type*

What we get in Java 8 is multiple inheritance of *behavior*

What we dont have is multiple inheritance of *state*

Default methods

What we have so far is multiple inheritance of *type*

What we get in Java 8 is multiple inheritance of *behavior*

What we dont have is multiple inheritance of *state*

… and this is where the trouble is !

Default methods

Will we have conflicts ?

Default methods

Will we have conflicts ?

Yes, so we need rules to handle them

Default methods

```
public class C implements A, B {
    // ...
}
```

Default methods

```
public class C implements A, B {
    // ...
}
```

Example #1

```
public interface A {
    default String a() { ... }
}
```

```
public interface B {
    default String a() { ... }
}
```

Default methods

```
public class C implements A, B {
    // ...
}
```

Example #1

```
public interface A {
    default String a() { ... }
}
```

```
public interface B {
    default String a() { ... }
}
```

Compile time error :

class C inherits unrelated defaults for a() from types A and B

Default methods

```
public class C implements A, B {
    public String a() { ... }
}
```

Rule #1

```
public interface A {
    default String a() { ... }
}
```

```
public interface B {
    default String a() { ... }
}
```

Class wins !

No default if an implementation is present

Default methods

```
public class C implements A, B {
    // ...
}
```

Rule #2

```
public interface A extends B {
    default String a() { ... }
}
```

```
public interface B {
    default String a() { ... }
}
```

A is more specific than B : A.a() is chosen

Default methods

```
public class C implements A, B {
    public String a() { return B.super(a) ; }
}
```

Back to rule #1

```
public interface A {
    default String a() { ... }
}
```

```
public interface B {
    default String a() { ... }
}
```

We can also make an explicit call

2 simple rules

To handle the conflict of multiple inheritance of *behavior*

2 simple rules

To handle the conflict of multiple inheritance of *behavior*

1) Class wins !

2 simple rules

To handle the conflict of multiple inheritance of *behavior*

1) Class wins !
2) More specific interfaces wins over less specific

2 simple rules

To handle the conflict of multiple inheritance of *behavior*

1) Class wins !
2) More specific interfaces wins over less specific

An example

Everybody who writes an implementation of Iterator writes this :

```
public class MyIterator implements Iterator<E> {
    // some critical business code

    public void remove() {
       throw new UnsupportedOperationException("Naah !!!!!!") ;
    } ;
}
```

An example

Thanks to Java 8 :

```
public interface Iterator<E> {

   default void remove() {
      throw new UnsupportedOperationException("remove") ;
   } ;
}
```

No silly implementation of remove() anymore !

Java 8 interfaces

So the concept of « interface » changed in Java 8

```
public class HelloWorld {
    // souvenir souvenir

    public static void main(String[] args) {
       System.out.println("Hello world!") ;
    } ;
}
```

Java 8 interfaces

So the concept of « interface » changed in Java 8

I mean, really

```
public interface HelloWorld {
    // souvenir souvenir

    public static void main(String[] args) {
       System.out.println("Hello world!") ;
    } ;
}
```

Java 8 interfaces

1) Default methods, multiple *behavior* inheritance

   Rules to handle conflicts, solved at compile time

2) Static methods in interfaces

Will bring new patterns and new ways to write APIs

Where are we ?

1) We have a new syntax
2) We have a new pattern to implement
3) We have new interfaces to keep the backward compatibily

The Stream API

What is a Stream

From a technical point of view : a typed interface

« a stream of String »

What is a Stream

From a technical point of view : a typed interface

« a stream of String »

Other classes for primitive types :

IntStream, LongStream, DoubleStream

It’s a new notion

It looks like a collection, but…

- It is built on a « source », that can be a collection
- No data needs to be provided at build time
- No limit on what the source can produce

It’s a new notion

A stream can be built on :

- A collection or an array
- An iterator
- An I/O source

Some of those can be « infinite »

It’s a new notion

1) A stream doesnt hold any data

It’s just an object on which one can declare operations

Streams are about « specifying computations »

It’s a new notion

1) A stream doesnt hold any data
2) A stream cant modify its source

Consequence : it can use parallel processing

It’s a new notion

1) A stream doesnt hold any data
2) A stream cant modify its source
3) A source can be infinite

So we need a way to guarantee that computations will be conducted in finite time

It’s a new notion

1) A stream doesnt hold any data
2) A stream cant modify its source
3) A source can be infinite
4) A stream works lazily

Then optimizations can be made among the different operations

We need a way / convention to trigger the computations

How can we build a stream ?

Many ways…

1) From a collection : method Collection.stream

```
Collection<String> collection = ... ;
Stream<String> stream = collection.stream() ;
```

How can we build a stream ?

Many ways…

1) From a collection : method Collection.stream
2) From an array : Arrays.stream(Object [])

```
Stream<String> stream2 =
   Arrays.stream(new String [] {"one", "two", "three"}) ;
```

How can we build a stream ?

Many ways…

1) From a collection : method Collection.stream
2) From an array : Arrays.stream(Object [])
3) From factory methods in Stream, IntStream, …

```
Stream<String> stream1 = Stream.of("one", "two", "three") ;
```

How can we build a stream ?

Some last patterns

```
Stream.empty() ; // returns an empty Stream
Stream.of(T t) ; // a stream with a single element

Stream.generate(Supplier<T> s) ;
Stream.iterate(T seed, UnaryOperator<T> f) ;
```

How can we build a stream ?

Other ways scattered in the JDK

```
string.chars() ;              // returns a IntStream
lineNumberReader.lines() ; // returns a Stream<String>
random.ints() ;               // return a IntStream
```

A 1st example

Back to our map / filter / reduce example

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Builds a stream on a List

A 1st example

Back to our map / filter / reduce example

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Declares operations

A 1st example

Back to our map / filter / reduce example

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Triggers the computation

A 1st example

Back to our map / filter / reduce example

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Default value, in case the stream is empty

3 things on Streams

1) Two types of operations

One can declare operations on a Stream

Two types of operations :

1) Intermediate operations : declarations, processed lazily

ex : map, filter

1) Two types of operations

One can declare operations on a Stream

Two types of operations :

1) Intermediate operations : declarations, processed lazily

   ex : map, filter

2) Terminal operations : trigger the computations

   ex : reduce

2) State of a stream

The implementation of Stream has a state :

- SIZED : the number of elements is known
- ORDERED : the order matters (List)
- DISTINCT : all elements are unique (Set)
- SORTED : all elements have been sorted (SortedSet)

2) State of a stream

Some operations change that :

- filtering removes the SIZED
- mapping removes DISTINCT and SORTED

2) State of a stream

It’s all about optimization!

```
Collection<Person> collection = ... ;
Stream<Person> stream = collection.stream() ;

stream.distinct()
      .map(Person::getAge)
      .filter(a -> a > 20)
      .reduce(0, Integer::sum) ;
```

2) State of a stream

It’s all about optimization!

```
Collection<Person> collection = new HashSet<>() ;
Stream<Person> stream = collection.stream() ; // DISTINCT

stream.distinct() // DISTINCT = will not do anything
      .map(Person::getAge)
      .filter(a -> a > 20)
      .reduce(0, Integer::sum) ;
```

2) State of a stream

It’s all about optimization!

```
Collection<Person> collection = ... ;
Stream<Person> stream = collection.stream() ;

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

2) State of a stream

It’s all about optimization!

```
Collection<Person> collection = new SortedSet<>() ;
Stream<Person> stream = collection.stream() ; // SORTED

stream.sorted() // SORTED = doesnt do anything
      .forEach(System.out::println) ;
```

3) Stateless / stateful operations

Stateless / stateful operations

Some operations are stateless :

```
persons.stream().map(p -> p.getAge()) ;
```

It means that they dont need more information than the one held by their parameters

3) Stateless / stateful operations

Stateless / stateful operations

Some operations are stateless :

```
persons.stream().map(p -> p.getAge()) ;
```

Some needs to retain information :

```
stream.limit(10_000_000) ; // select the 10M first elements
```

Example

Let’s sort an array of String

```
Random rand = new Random() ;

String [] strings = new String[10_000_000] ;
for (int i = 0 ; i < strings.length ; i++) {
   strings[i] = Long.toHexString(rand.nextLong()) ;
}
```

Example

Let’s sort an array of String

```
Random rand = new Random() ;

String [] strings = new String[10_000_000] ;
for (int i = 0 ; i < strings.length ; i++) {
   strings[i] = Long.toHexString(rand.nextLong()) ;
}
```

Soooo Java 7…

Example

Let’s sort an array of String

```
Random rand = new Random() ;

Stream<String> stream =
  Stream.generate(
    () ->
    Long.toHexString(rand.nextLong())
  ) ;
```

Better ?

Example

Let’s sort an array of String

```
// Random rand = new Random() ;

Stream<String> stream =
  Stream.generate(
    () ->
    Long.toHexString(ThreadLocalRandom.current().nextLong())
  ) ;
```

Better !

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
   ThreadLocalRandom
         .current()
         .longs() // returns a LongStream
         .mapToObj(l -> Long.toHexString(l)) ;
```

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
    ThreadLocalRandom
          .current()
          .longs()
          .mapToObj(Long::toHexString) ;
```

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
    ThreadLocalRandom
          .current()
          .longs()
          .mapToObj(Long::toHexString)
          .limit(10_000_000)
          .sorted() ;
```

T = 4 ms

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
    ThreadLocalRandom
          .current()
          .longs()
          .mapToObj(Long::toHexString)
          .limit(10_000_000)
          .sorted() ;

Object [] sorted = stream.toArray() ;
```

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
    ThreadLocalRandom
          .current()
          .longs()
          .mapToObj(Long::toHexString)
          .limit(10_000_000)
          .sorted() ;

Object [] sorted = stream.toArray() ;
```

T = 14 s

Example

Let’s sort an array of String

```
// other way
Stream<String> stream =
    ThreadLocalRandom
          .current()
          .longs()
          .mapToObj(Long::toHexString)
          .limit(10_000_000)
          .sorted() ;

Object [] sorted = stream.toArray() ;
```

Intermediate calls !

T = 14 s

More on stream operations

1) There are intermediate and terminal operations
2) A stream is processed on a terminal operation call

More on stream operations

1) There are intermediate and terminal operations
2) A stream is processed on a terminal operation call
3) Only one terminal operation is allowed
4) It cannot be processed again

If needed another stream should be built on the source

Parallel Streams

Optimization

The first optimization (well, laziness was first !) we need is parallelism

The fork / join enable parallel programming since JDK 7

But writing tasks is hard and does not always lead to better performances

Optimization

The first optimization (well, laziness was first !) we need is parallelism

The fork / join enable parallel programming since JDK 7

But writing tasks is hard and does not always lead to better performances

Using the parallel stream API is much more secure

How to build a parallel stream ?

Two patterns

1) Call parallelStream() instead of stream()

```
Stream<String> s = strings.parallelStream() ;
```

2) Call parallel() on an existing stream

```
Stream<String> s = strings.stream().parallel() ;
```

Can we decide the # of cores ?

Answer is yes, but it’s tricky

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

Can we decide the # of cores ?

Answer is yes, but it’s tricky

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

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

Parallelism

Parallelism implies overhead most of the time

Badly configured parallel operations can lead to unneeded computations that will slow down the overall process

Unnecessary ordering of a stream will lead to performance hits

Reductions

A simple reduction

Sum of ages

```
Stream stream3 = Stream.concat(stream1, stream2) ;
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

A simple reduction

It would be nice to be able to write :

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .sum() ;
```

But there’s no sum() on Stream<T>

What would be the sense of « adding persons » ?

A simple reduction

It would be nice to be able to write :

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .sum() ;
```

But there’s no sum() on Stream<T>

There’s one on IntStream !

A simple reduction

2nd version :

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(Person::getAge)
    .filter(a -> a > 20)
    .mapToInt(Integer::intValue)
    .sum() ;
```

Value if persons is an empty list : 0

A simple reduction

2nd version :

```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    // .mapToInt(Integer::intValue)
    .sum() ;
```

Value if persons is an empty list : 0

Default values

A simple reduction

What about max() and min() ?

```
// map / filter / reduce pattern on collections
....... = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    .max() ;
```

How can one chose a default value ?

Problem with the default value

The « default value » is a tricky notion

Problem with the default value

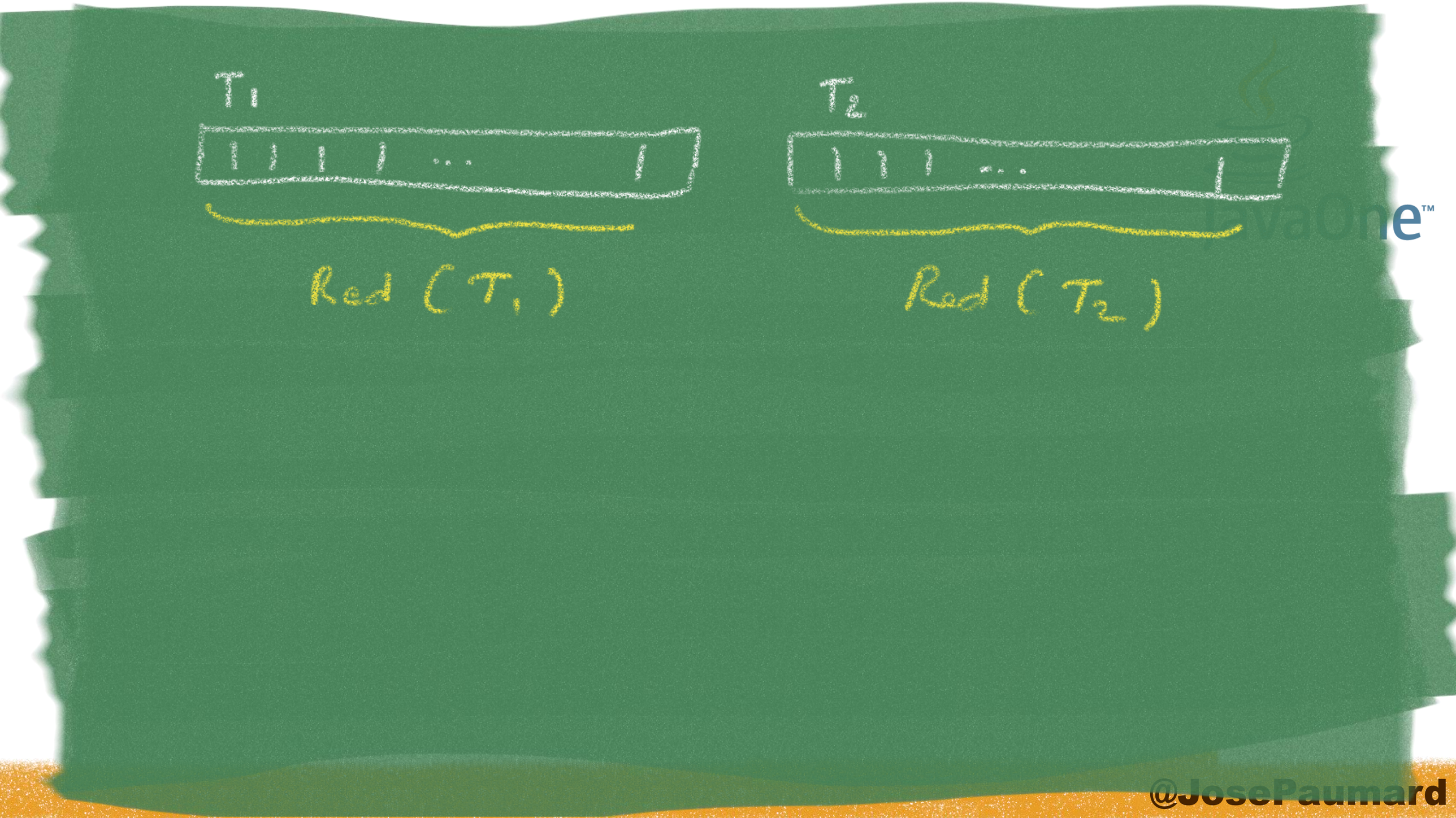
The « default value » is a tricky notion

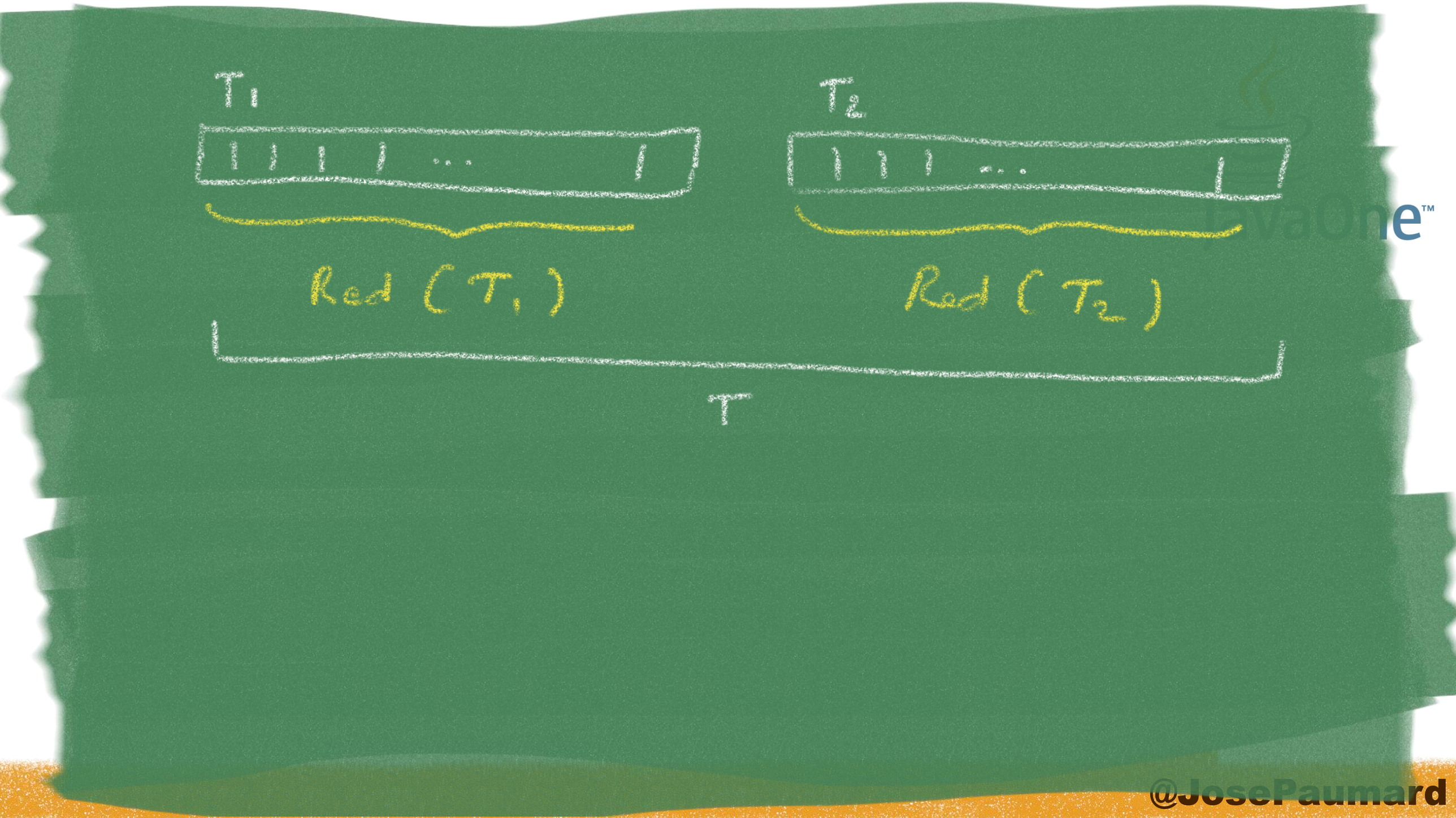
1) The « default value » is the reduction of the empty set

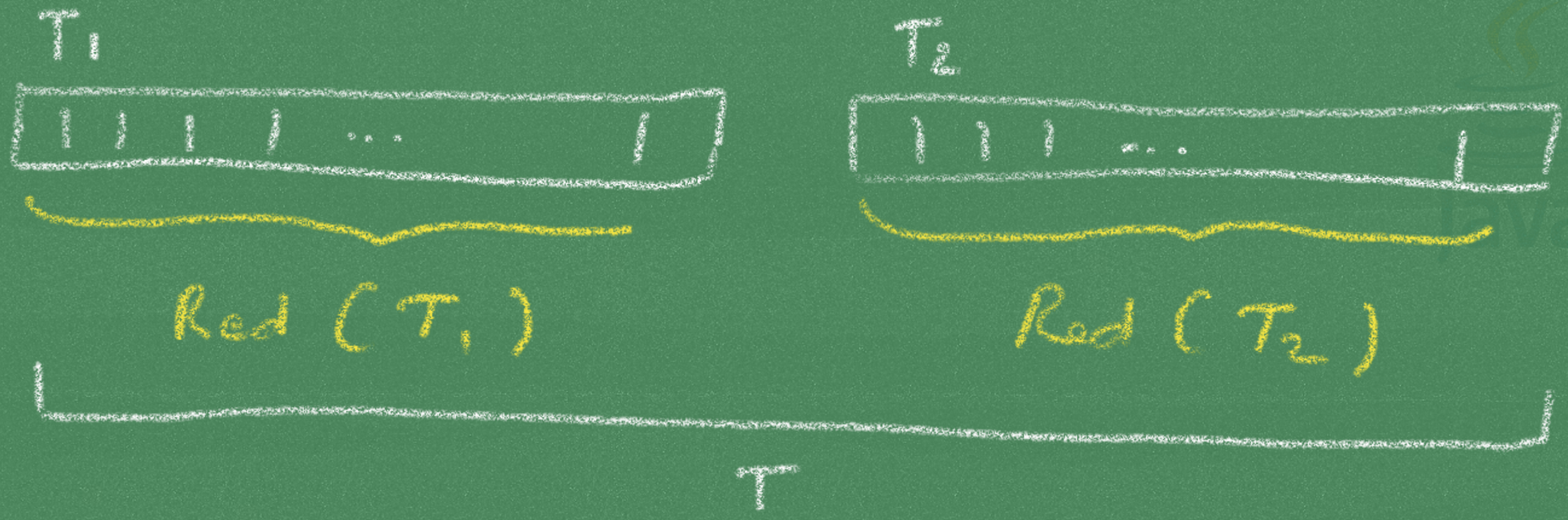
Problem with the default value

The « default value » is a tricky notion

1) The « default value » is the reduction of the empty set
2) But it’s also the identity element for the reduction



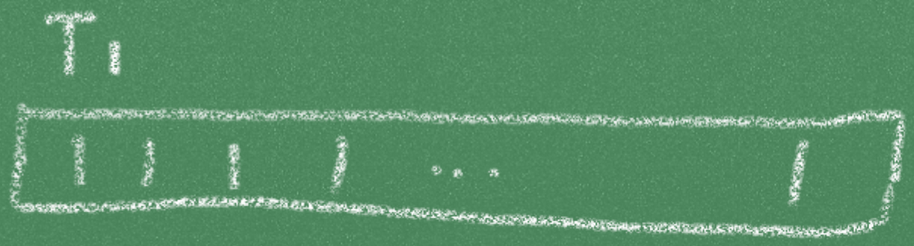


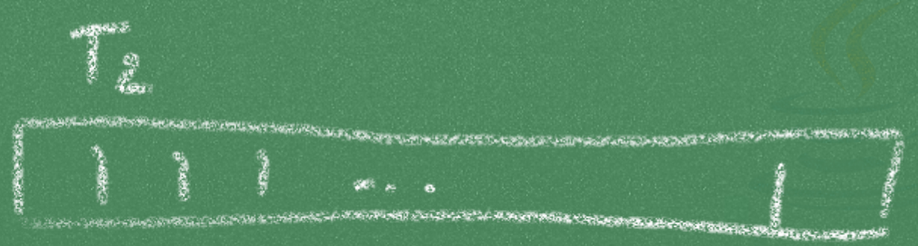


$$T = T_1 \cup T_2$$

$$Red(T) = Red(Red(T_1), Red(T_2))$$

Suppose T_1 is empty = ϕ





Suppose T_1 is empty $= \phi$

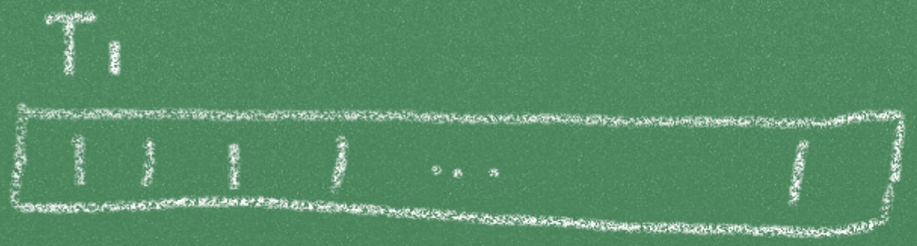
$T = T_2$

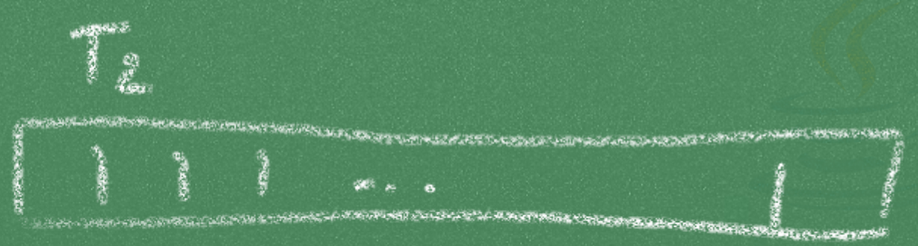
Suppose T_1 is empty $= \phi$

$T = T_2$

$Red(T) = Red(Red(T_1), Red(T_2))$

$Red(T) = Red(Red(\phi), Red(T))$





Suppose T_1 is empty $= \phi$

$T = T_2$

$Red(T) = Red(Red(T_1), Red(T_2))$

$Red(T) = Red(Red(\phi), Red(T))$

$Red(\phi) \rightarrow$ Identity element

Problem with the default value

The « default value » is a tricky notion

1) The « default value » is the reduction of the empty set
2) But it’s also the identity element for the reduction

Problem with the default value

Problem : max() and min() have no identity element

eg : an element *e* for which max(*e*, a) = a

Problem with the default value

Problem : max() and min() have no identity element

eg : an element *e* for which max(*e*, a) = a

0 cant be, max(0, -1) is not -1…

$-\infty$ is not an integer

Problem with the default value

Problem : max() and min() have no identity element

eg : an element *e* for which max(*e*, a) = a

0 cant be, max(0, -1) is not -1…

$-\infty$ is not an integer

Integer.*MAX_VALUE*

Problem with the default value

So what is the default value of max() and min() ?

Problem with the default value

So what is the default value of max() and min() ?

Answer : there is no default value for max() and min()

Problem with the default value

So what is the returned type of the max() reduction ?

```
// map / filter / reduce pattern on collections
..... = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    .max() ;
```

Problem with the default value

So what is the returned type of the max() reduction ?

```
// map / filter / reduce pattern on collections
..... = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    .max() ;
```

If it’s int, then the default value will be 0…

Problem with the default value

So what is the returned type of the max() reduction ?

```
// map / filter / reduce pattern on collections
........ = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    .max() ;
```

If it’s Integer, then the default value will be null…

Optionals

Since there’s none, we need another notion

```
// map / filter / reduce pattern on collections
OptionalInt optionalMax = persons.stream()
    .mapToInt(Person::getAge)
    .filter(a -> a > 20)
    .max() ;
```

« there might be no result »

Optionals

What can I do with OptionalInt ?

1st pattern : test if it holds a value

```
OptionalInt optionalMax = ... ;

int max ;
if (optionalMax.isPresent()) {
   max = optionalMax.get() ;
} else {
   max = ... ; // decide a « default value »
}
```

Optionals

What can I do with OptionalInt ?

2nd pattern : read the held value, can get an exception

```
OptionalInt optionalMax = ... ;

// throws NoSuchElementException if no held value
int max = optionalMax.getAsInt() ;
```

Optionals

What can I do with OptionalInt ?

3rd pattern : read the held value, giving a default value

```
OptionalInt optionalMax = ... ;

// get 0 if no held value
int max = optionalMax.orElse(0) ;
```

Optionals

What can I do with OptionalInt ?

4th pattern : read the held value, or throw an exception

```
OptionalInt optionalMax = ... ;

// exceptionSupplier will supply an exception, if no held value
int max = optionalMax.orElseThrow(exceptionSupplier) ;
```

Available optionals

Optional<T>

OptionalInt, OptionalLong, OptionalDouble

Available reductions

On Stream<T> :

- reduce(), with different parameters
- count(), min(), max()
- anyMatch(), allMatch(), noneMatch()
- findFirst(), findAny()

- toArray()
- forEach(), forEachOrdered()

Available reductions

On IntStream, LongStream, DoubleStream :

- average()
- summaryStatistics()

Available reductions

SummaryStatistics is an object that holds :

- Count, Min, Max, Sum, Average

It’s a consumer, so it can be easily extended to compute other statistics

Mutable reductions

Mutable reductions : example 1

Use of the helper class : Collectors

```
ArrayList<String> strings =
stream
    .map(Object::toString)
    .collect(Collectors.toList()) ;
```

Mutable reductions : example 2

Concatenating strings with a helper

```
String names = persons
    .stream()
    .map(Person::getName)
    .collect(Collectors.joining()) ;
```

Mutable reductions

Common things :

- have a container : Collection or StringBuilder
- have a way to add an element to the container
- have a way to merge two containers (for parallelization)

Mutable reductions : example 1

Putting things together :

```
ArrayList<String> strings =
stream
.map(Object::toString)
.collect(
   () -> new ArrayList<String>(),              // the supplier
   (suppList, s) -> suppList.add(s),           // the accumulator
   (suppL1, suppL2) -> suppL1.addAll(suppL2) // the combiner
) ;
```

Mutable reductions : example 1

Putting things together :

```
ArrayList<String> strings =
stream
    .map(Object::toString)
    .collect(
       ArrayList::new,    // the supplier
       ArrayList::add,    // the accumulator
       ArrayList::addAll // the combiner
    ) ;
```

Mutable reductions : example 1

Putting things together :

```
ArrayList<String> strings =
stream
    .map(Object::toString)
    .collect(Collectors.toList()) ;
```

Collectors

The Collectors class

A rich toolbox (37 methods) for various types of reductions

- counting, minBy, maxBy
- summing, averaging, summarizing
- joining
- toList, toSet

And

- mapping, groupingBy, partionningBy

The Collectors class

Average, Sum, Count

```
persons
    .stream()
    .collect(Collectors.averagingDouble(Person::getAge)) ;
```

```
persons
    .stream()
    . collect(Collectors.counting()) ;
```

The Collectors class

Concatenating the names in a String

```
String names = persons
    .stream()
    .map(Person::getName)
    .collect(Collectors.joining(", ")) ;
```

The Collectors class

Accumulating in a List, Set

```
Set<Person> setOfPersons = persons
    .stream()
    .collect(
       Collectors.toSet()) ;
```

The Collectors class

Accumulating in a custom collection

```
TreeSet<Person> treeSetOfPersons = persons
    .stream()
    .collect(
       Collectors.toCollection(TreeSet::new)) ;
```

The Collectors class

Getting the max according to a comparator

```
Optional<Person> optionalPerson = persons
    .stream()
    .collect(
       Collectors.maxBy(
          Comparator.comparing(Person::getAge)) ;
```

Bonus : new API to build comparators

Building comparators

New API to build comparators in a declarative way

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

Building comparators

New API to build comparators in a declarative way

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

```
Comparator<Person> comp =
    Comparator.comparing(Person::getLastName)
              .reversed() ;
```

Building comparators

New API to build comparators in a declarative way

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

```
Comparator<Person> comp =
    Comparator.nullsFirst(
        Comparator.comparing(Person::getLastName)
    ) ;
```

Other examples

Interface Predicate

```
Predicate<Long> p1 = i -> i > 20 ;
Predicate<Long> p2 = i -> i < 50 ;

Predicate<Long> p3 = p1.and(p2) ;

Person brian = ... ;
Predicate<Person> p = Predicat.isEqual(brian) ;
```

The Collector API : groupingBy

« Grouping by » builds hash maps

- must explain how the keys are built
- by default the values are put in a list
- may specify a downstream (ie a collector)

The Collector API : groupingBy

Grouping a list of persons by their age

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

The Collector API : groupingBy

Grouping a list of persons by their age

```
Map<Integer, Set<Person>> map =
persons
    .stream()
    .collect(
       Collectors.groupingBy(
          Person::getAge,
          Collectors.toSet() // the downstream
       ) ;
```

The Collector API : groupingBy

Grouping a list of persons names by their age

```
Map<Integer, Set<String>> map =
persons
    .stream()
    .collect(
       Collectors.groupingBy(
          Person::getAge,
          Collectors.mapping(      //
             Person::getLastName, // the downstream
             Collectors.toSet()   //
          )
       ) ;
```

The Collector API : groupingBy

Grouping a list of persons names by their age

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

The Collector API : groupingBy

Grouping a list of blah blah blah

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

The Collector API : groupingBy

Example : creating an age histogram

```
Map<Integer, Long> map =
persons
    .stream()
    .collect(
       Collectors.groupingBy(Person::getAge, Collectors.counting())
    ) ;
```

Gives the # of persons by age

The Collector API : partitioningBy

Creates a Map<Boolean, …> on a predicate

- the map has 2 keys : TRUE and FALSE
- may specify a downstream

The Collector API : partioningBy

Creates a Map<Boolean, …> on a predicate

```
Map<Boolean, List<Person>> map =
    persons
        .stream()
        .collect(
           Collectors.partitioningBy(p -> p.getAge() > 20)
        ) ;
```

map.get(TRUE) returns the list people older than 20

The Collector API : partioningBy

Can further process the list of persons

```
Map<Boolean, TreeSet<String>> map =
    persons
        .stream()
        .collect(
           Collectors.partitioningBy(
              p -> p.getAge() > 20,
              Collectors.mapping(
                 Person::getLastName,
                 Collectors.toCollection(TreeSet::new))
              )
           )
        ) ;
```

The Collector API : collectingAndThen

Collect data with a downstream

Then apply a function called a « finisher »

Useful for putting the result in a immutable collection

The Collector API : collectingAndThen

```
Set<Map.Entry<Integer, List<Person>>> set =
persons
    .stream()
    .collect(
       Collectors.collectingAndThen(
          Collectors.groupingBy(
             Person::getAge),     // downstream, builds a map
          Map::entrySet           // finisher, applied on the map
       ) ;
```

In this case « Map::entrySet » is a *finisher*

The Collector API : collectingAndThen

```
Map<Integer, List<Person>> map =  // immutable
persons
    .stream()
    .collect(
       Collectors.collectingAndThen(
          Collectors.groupingBy(
             Person::getAge),           // the downstream builds a map
          Collections::unmodifiableMap // finisher is applied to the
                                       // map
       ) ;
```

Some real examples

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

Conclusion

Conclusion

Why are lambdas introduced in Java 8 ?

Conclusion

Why are lambdas introduced in Java 8 ?

Because it’s in the mood !

Conclusion

Why are lambdas introduced in Java 8 ?

~~Because it’s in the mood !~~

Conclusion

Why are lambdas introduced in Java 8 ?

~~Because it’s in the mood !~~

Because it allows to write more compact code !

Compact code is better !

An example of compact code (in C)

```
#include "stdio.h"
main()  {
   int b=0,c=0,q=60,_=q;for(float i=-20,o,O=0,l=0,j,p;j=O*O,p=l*l,
   (!_--|(j+p>4)?fputc(b?q+(_/3):10,(i+=!b,p=j=O=l=0,c++,stdout)),
   _=q:l=2*O*l+i/20,O=j-p+o),b=c%q,c<2400;o=-2+b*.05) ;
}
```

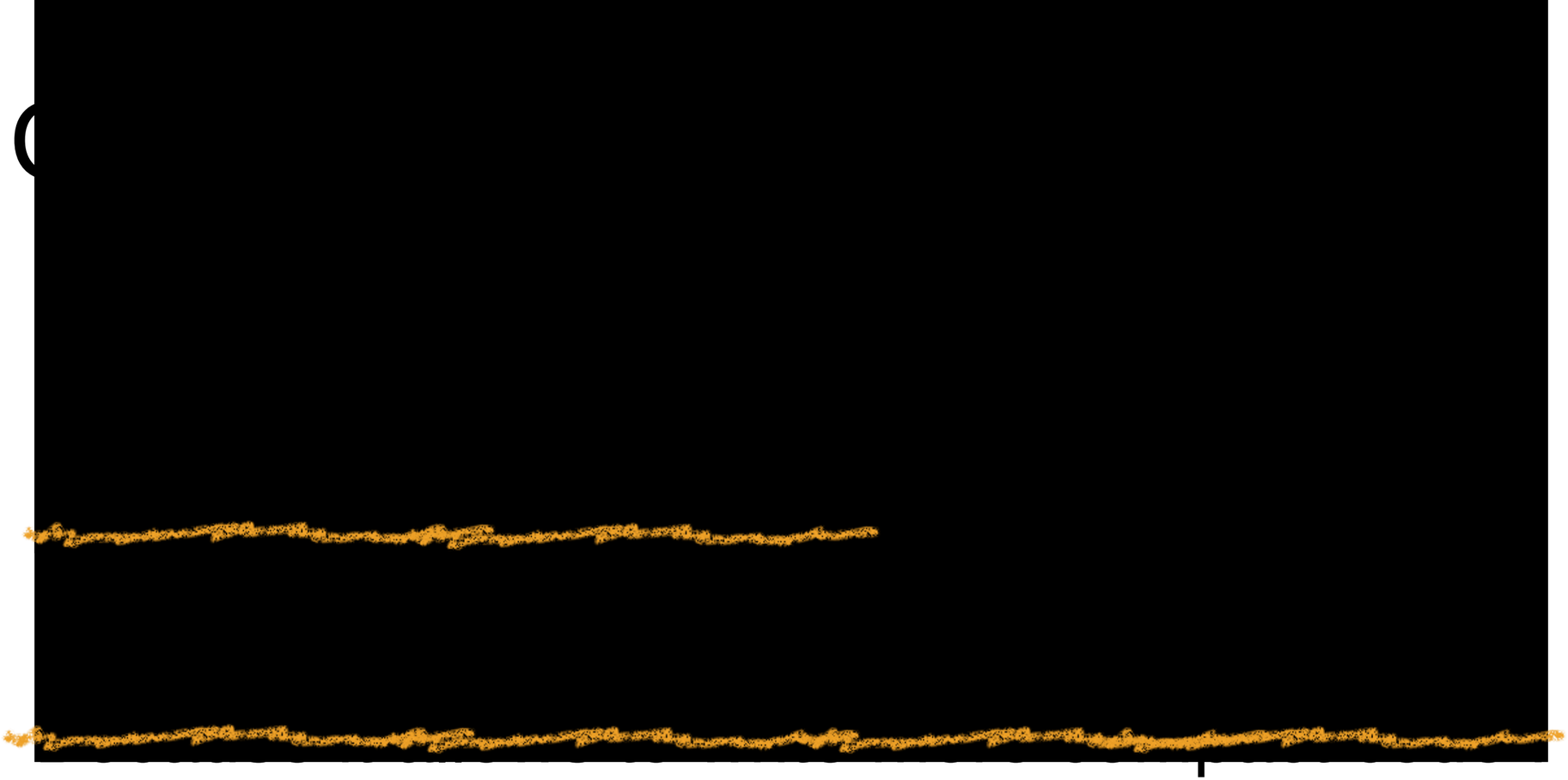
http://www.codeproject.com/Articles/2228/Obfuscating-your-Mandelbrot-code

Compact code is better !

An example of compact code (in C)

```
#include "stdio.h"
main()  {
    int b=0,c=0,q=60,_=q;for(float i=-20,o,O=0,l=
    (!_--|(j+p>4)?fputc(b?q+(_/3):10,(i+=!b,p=j=(
    _=q:l=2*O*l+i/20,O=j-p+o),b=c%q,c<2400;o=-2+|
}
```

http://www.codeproject.com/Articles/2228/Obfuscating-your-Mandelbrot-code



Conclusion

Java 8 is there, it’s the biggest update in 15 years

Conclusion

Java 8 is there, it’s the biggest update in 15 years

Moving to Java 8 means a lot of work for us developers !

- Self training
- Changing our habits

Conclusion

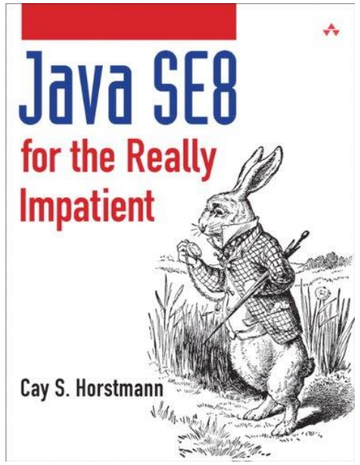
Java 8 is coming, it’s the biggest update in 15 years

Moving to Java 8 means a lot of work for us developers !

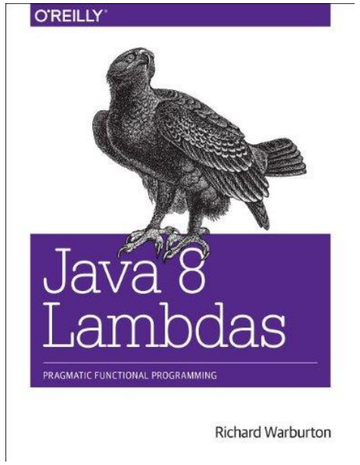
- Self training
- Changing our habits
- Convincing our bosses (sigh…)

Conclusion

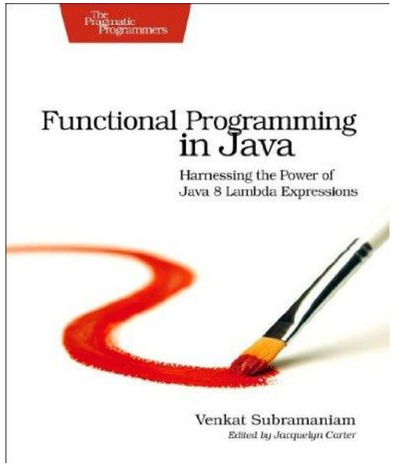
Books are coming !



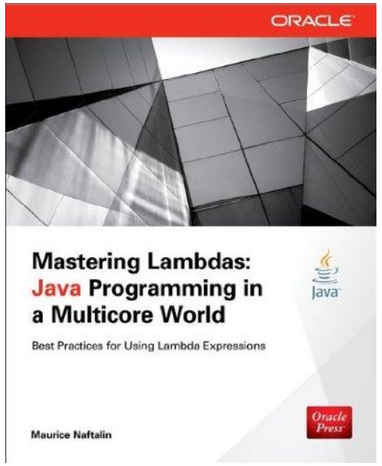
available



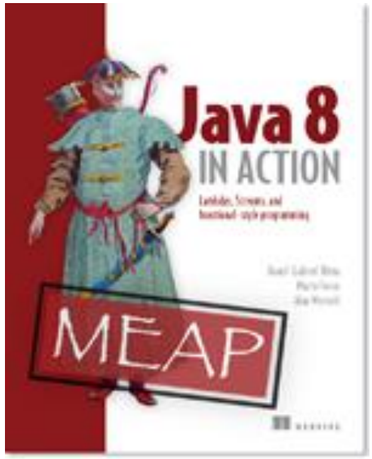
available



available



coming soon



coming soon

Conclusion

Java 8 is the biggest update in 15 years

« Java is a blue collar language. It’s not PhD thesis material but a language for a job » – James Gosling, 1997

Conclusion

Java 8 is the biggest update in 15 years

« Language features are not a goal unto themselves; language features are enablers, encouraging or discouraging certain styles and idioms » – Brian Goetz, 2013

Conclusion

Java 8 is the biggest update in 15 years

The good news is :
Java is still Java !


Thank you!


Q/A