Implementing Map Filter Reduce Using Lambdas and Collections

José Paumard

@JosePaumard | blog.paumard.org

Agenda

Introduction to the map / filter / reduce

A focus on the reduction step

How to implement it in the JDK

A Simple Example

Map / filter / reduce on a classical case

A Simple Example

- Let us compute the average of the age of people older than 20

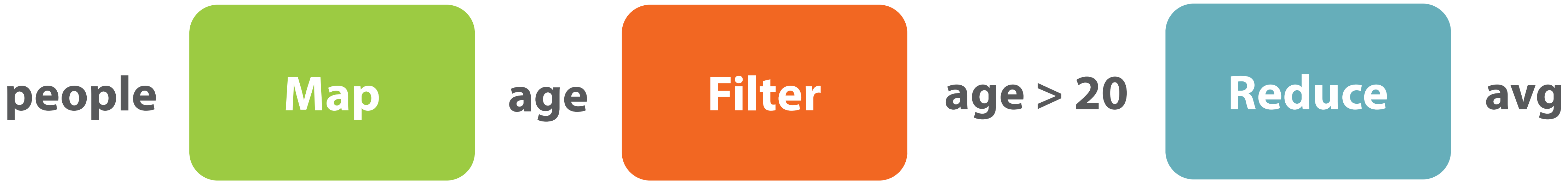
```
List<Person> people = ... ;

int sum = 0;
int count = 0;
for (Person p : people) {
   if (p.getAge() > 20) {
      sum += p.getAge();
      count++ ;
   }
}
int average = 0;
if (count > 0)
   average = sum / count;
```

This is the
Java 7 way
of writing things

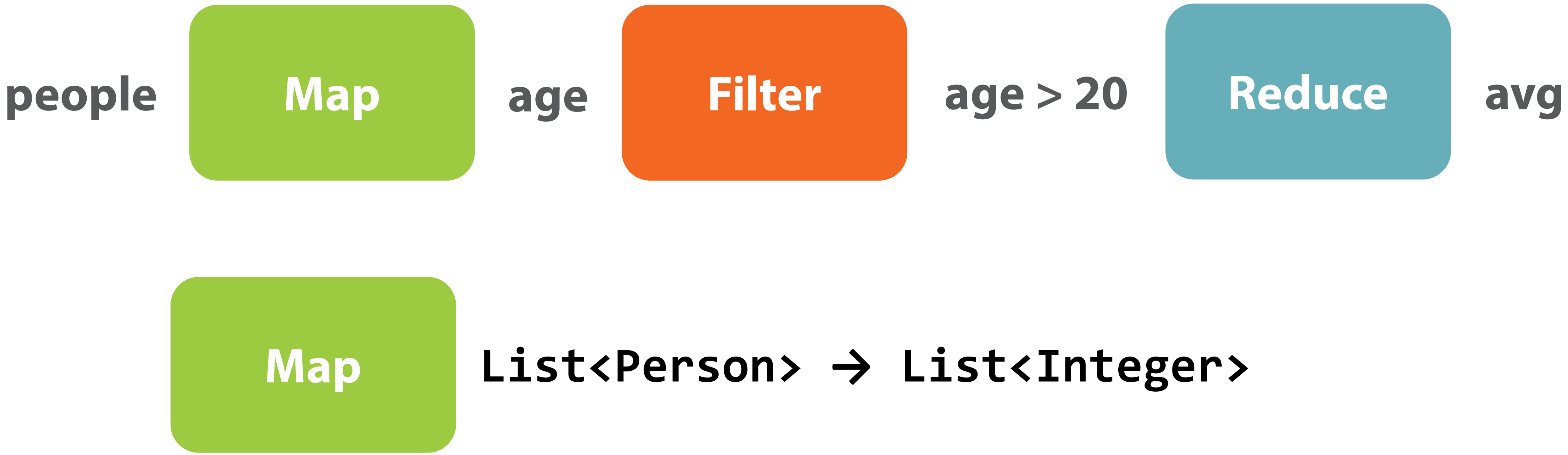
A Simple Example

- Let us compute the average of the age of people older than 20



A Simple Example

- Let us compute the average of the age of people older than 20



people

Map

age

Filter

age > 20

Reduce

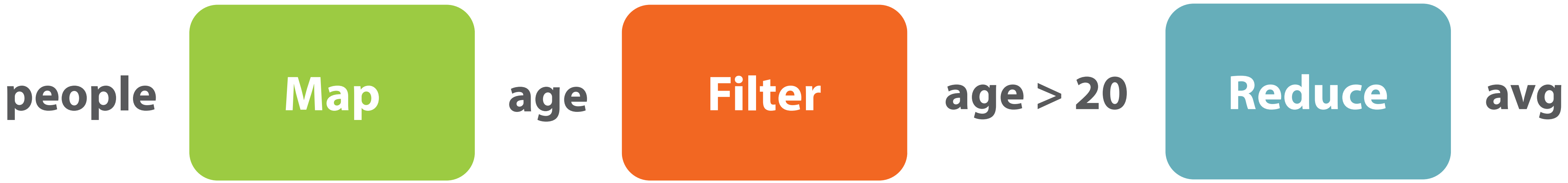
avg

Map

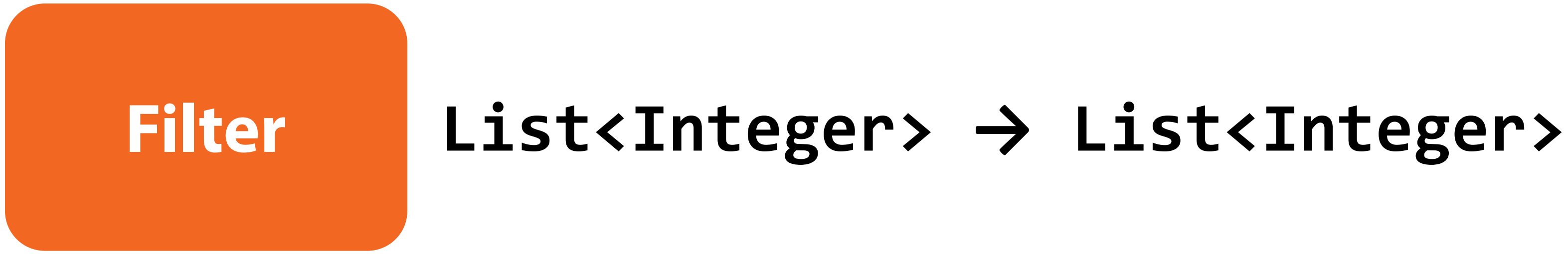
`List<Person> → List<Integer>`

A Simple Example

- Let us compute the average of the age of people older than 20



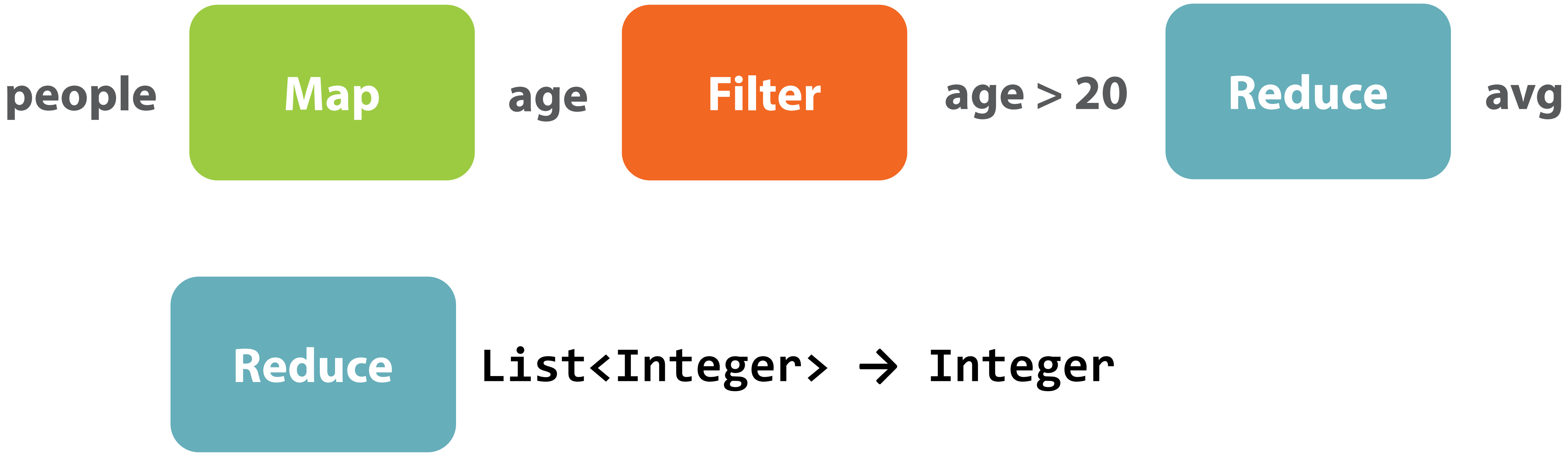
people **Map** age **Filter** age > 20 **Reduce** avg



Filter `List<Integer> → List<Integer>`

A Simple Example

- Let us compute the average of the age of people older than 20



people **Map** age **Filter** age > 20 **Reduce** avg

Reduce `List<Integer> → Integer`

How Can We Design an API?

- Java 7 is fond of helper classes, let us create a Lists class

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int sum                = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

- It does what we want: push the data + lambdas to the API, and let it handle everything

A Focus on the Reduction Step

Tips and pitfalls on the reduction step

A Focus on the Reduction Step

- The reduction step is written in this way:

```
int sum = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

- How can we reduce lists with this lambda?

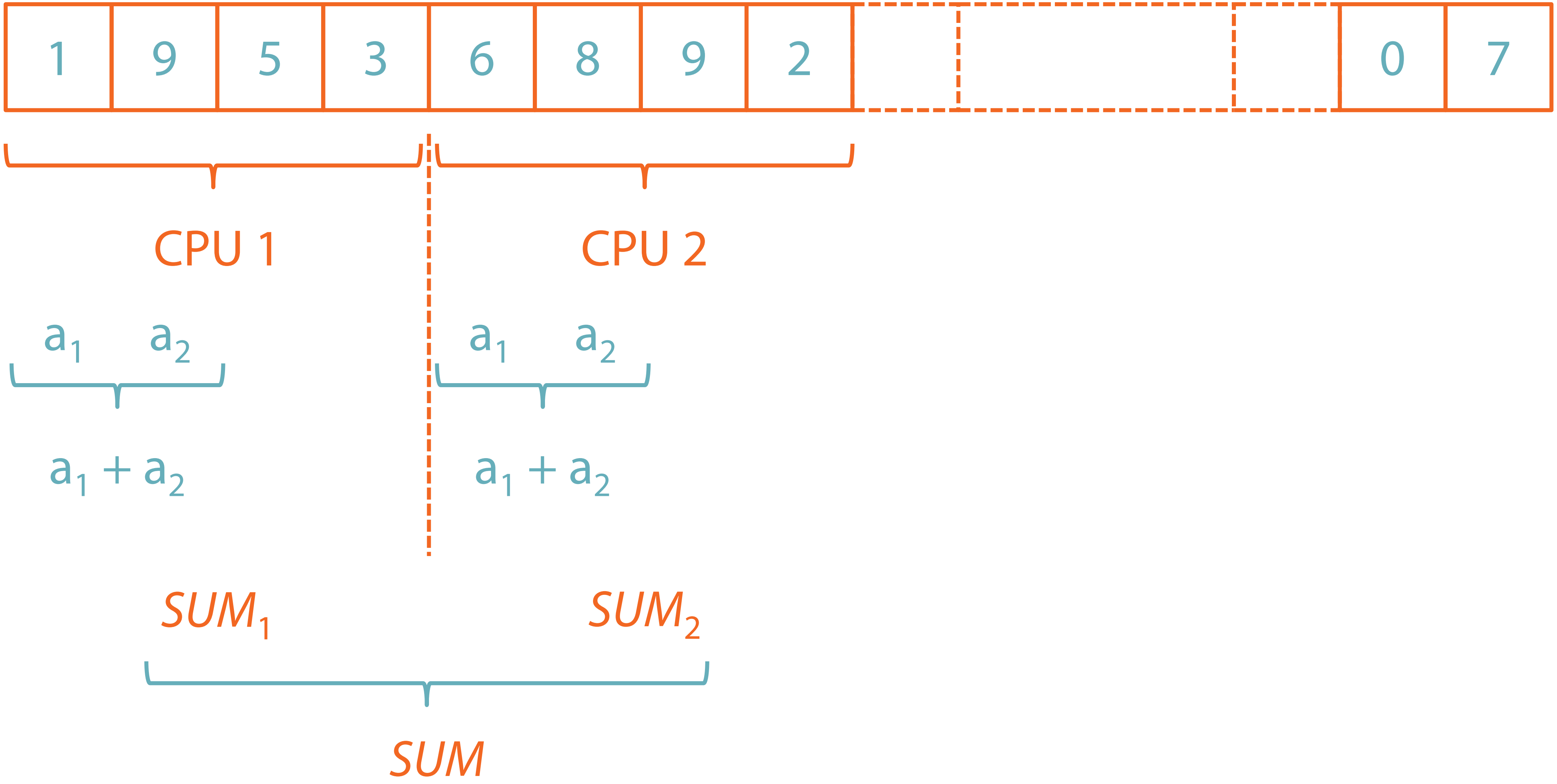
1	9	5	3				0	7

1st step: $\underbrace{a_1 \quad a_2}_{a_1 + a_2}$

2nd step: $\underbrace{a_1 \qquad a_2}_{a_1 + a_2}$

As a Bonus: Parallelization

- This algorithm is easily computed in parallel



1	9	5	3	6	8	9	2			0	7

CPU 1

a_1 a_2

$a_1 + a_2$

CPU 2

a_1 a_2

$a_1 + a_2$

SUM_1 *SUM_2*

SUM

As a Bonus: Parallelization

- This algorithm is easily computed in parallel
- But there is a condition:

$$Red(a, Red(b, c)) = Red(Red(a, b), c)$$

- It is called « associativity »

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;
```

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;

BinaryOperator<Integer> op2 = (i1, i2) -> Integer.max(i1, i2);
```

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;

BinaryOperator<Integer> op2 = (i1, i2) -> Integer.max(i1, i2);

BinaryOperator<Integer> op3 = (i1, i2) -> i1*i1 + i2*i2;
```

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;

BinaryOperator<Integer> op2 = (i1, i2) -> Integer.max(i1, i2);

BinaryOperator<Integer> op3 = (i1, i2) -> i1*i1 + i2*i2;

BinaryOperator<Integer> op4 = (i1, i2) -> i1;
```

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;

BinaryOperator<Integer> op2 = (i1, i2) -> Integer.max(i1, i2);

BinaryOperator<Integer> op3 = (i1, i2) -> i1*i1 + i2*i2;

BinaryOperator<Integer> op4 = (i1, i2) -> i1;

BinaryOperator<Integer> op5 = (i1, i2) -> (i1 + i2)/2;
```

Associativity

- Associative? Not associative?

```
BinaryOperator<Integer> op1 = (i1, i2) -> i1 + i2;

BinaryOperator<Integer> op2 = (i1, i2) -> Integer.max(i1, i2);

BinaryOperator<Integer> op3 = (i1, i2) -> i1*i1 + i2*i2;

BinaryOperator<Integer> op4 = (i1, i2) -> i1;

BinaryOperator<Integer> op5 = (i1, i2) -> (i1 + i2)/2;
```

Associativity

- If the lambda passed as a parameter is not associative, what is going to happen?
- In fact: nothing!

1) The code will compile properly
2) It will execute properly
3) A result will be returned
4) But it will be false!

- We need to be extra careful here!

A Focus on the Reduction Step

- Implementation of the reduction step:

```
List<Integer> ints = new ArrayList<>();
int sum = 0;
BinaryOperator<Integer> op = (i1, i2) -> i1 + i2;
for (int i : ints) {
   sum = op.apply(sum, i);
}
```

A Focus on the Reduction Step

- Implementation of the reduction step:

```
List<Integer> ints = new ArrayList<>();
int sum = 0;
BinaryOperator<Integer> op = (i1, i2) -> i1 + i2;
for (int i : ints) {
   sum = op.apply(sum, i);
}
```

- What is the meaning of this 0?

Reduction of Singletons

- Suppose we have only one element in our list:

```
List<Integer> ints = new Arrays.asList(1); // special case
int sum = 0;
BinaryOperator<Integer> op = (i1, i2) -> i1 + i2;
for (int i : ints) {
   sum = op.apply(sum, i);
}
```

- We expect the result to be 1

- There are cases that do not work like that!

Reduction of Singletons

- Suppose the reduction is a *max*

```
BinaryOperator<Integer> op = (i1, i2) -> Integer.max(i1, i2);
```

```
BinaryOperator<Integer> op = Integer::max;
```

```
List<Integer> ints = new ArrayList<>(); // special case
int max = 0;
BinaryOperator<Integer> op = Integer::max;
for (int i : ints) {
   max = op.apply(max, i);
}
```

Reduction of Singletons

- Suppose now that we compute the following

```
List<Integer> list1 = Arrays.asList(-1);
```

```
int max = 0;
BinaryOperator<Integer> op = Integer::max;
for (int i : ints) {
   max = op.apply(max, i);
}
```

- The max of list1 is … 0
- It should be -1

Reduction of Singletons

- Suppose now that we compute the following

```
List<Integer> list1 = Arrays.asList(-1, -2, -3);
```

```
int max = 0;
BinaryOperator<Integer> op = Integer::max;
for (int i : ints) {
   max = op.apply(max, i);
}
```

- The max of list1 is … 0
- It should be -1

Reduction of Singletons

- Why is the result not -1?
- Because 0 is not the identity element of the *max* operation

- The reduction should have an identity element
- Not all operations have one (*max*)

Live Coding

Caveats using reduction

Using non-associative reduction

Using reduction that has no identity element

Live Coding Summary

- We saw the main cases that do not work
- It is very easy to mess up things
- And nothing is here to prevent us from messing things up!

How Things Have Been Handled?

- The JDK introduces a new concept: Optional
- An Optional is a wrapper type that may be empty (*eg* ≠ Integer)

Conclusion on the Reduction Step

- The reduction is critical
- It is very easy to write a non-associative reduction
- It is very easy to write a reduction with no identity element

Conclusion: be extra careful
when designing the reduction step!

Implementation in the JDK

Map / filter / reduce put in the right way

How Can We Design an API?

- How to design a new JDK API to implement the map / filter / reduce?

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

How Can We Design an API?

- Any caveat on this approach?

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

How Can We Design an API?

- Any caveat on this approach?

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

How Can We Design an API?

- Any caveat on this approach?

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

- 2 duplications : ages and agesGT20
- High memory footpring, CPU load!

How Can We Design an API?

- It can be even worse…
- Suppose the reduction step is a « all match »

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

List<Integer> names = Lists.map(people, person -> person.getName());

boolean namesLT20    = Lists.allMatch(names, name -> name.length() < 20);
```

How Can We Design an API?

- What could be the code for the « all match »?

```java
public boolean allMatch() {
    for (String name : names) {
        if (name.length() < 20) {
            return false;
        }
    }
    return true;
}
```

How Can We Design an API?

- What could be the code for the « all match »?

```
public boolean allMatch() {
    for (String name : names) {
        if (name.length() < 20) {
            return false;
        }
    }
    return true;
}
```

- No need to scan all the elements to get the result…

How Can We Design an API?

- That is too bad, because…

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

List<Integer> names = Lists.map(people, person -> person.getName());

boolean namesLT20   = Lists.allMatch(names, name -> name.length() < 20);
```

- The list *names* has already been computed!

How Can We Design an API?

- So this way…

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

- Is not the right one to design such an API!

How Can We Design an API?

- So this way…

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

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

List<Integer> agesGT20 = Lists.filter(ages, age -> age > 20);

int average            = Lists.reduce(agesGT20, (a1, a2) -> a1 + a2);
```

- Is not the right one to design such an API!

How Can We Design an API?

- So this way…

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

List<Integer> ages     = people.map(person -> person.getAge());

List<Integer> agesGT20 = ages.filter(age -> age > 20);

int average            = agesGT20.reduce((a1, a2) -> a1 + a2);
```

- Is not the right one to design such an API!

How Can We Design an API?

- So how has it been done?
- The fact is that this way of writing things is nice

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

int average = people
                 .map(p -> p.getAge())
                 .filter(age -> age > 20)
                 .average();
```

How Can We Design an API?

- The choice has been made to add an intermediate call

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

int average = people.stream()
                    .map(p -> p.getAge())
                    .filter(age -> age > 20)
                    .average();
```

- The call to stream() returns a Stream, a new interface in Java 8

Summary

- What is the map / filter / reduce pattern
- Focus on the reduction step, which is the tricky one
- A quick hint about optionals
- How not to implement it on the Collection framework