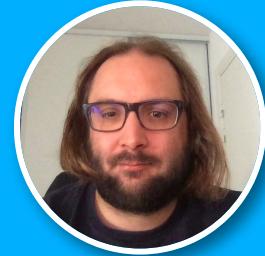




# Refactoring sans les **for**





# Igor Laborie

Expert Web & Java

@ilaborie <<https://twitter.com/ilaborie>>

igor@monkeypatch.io <<mailto:igor@monkeypatch.io>>

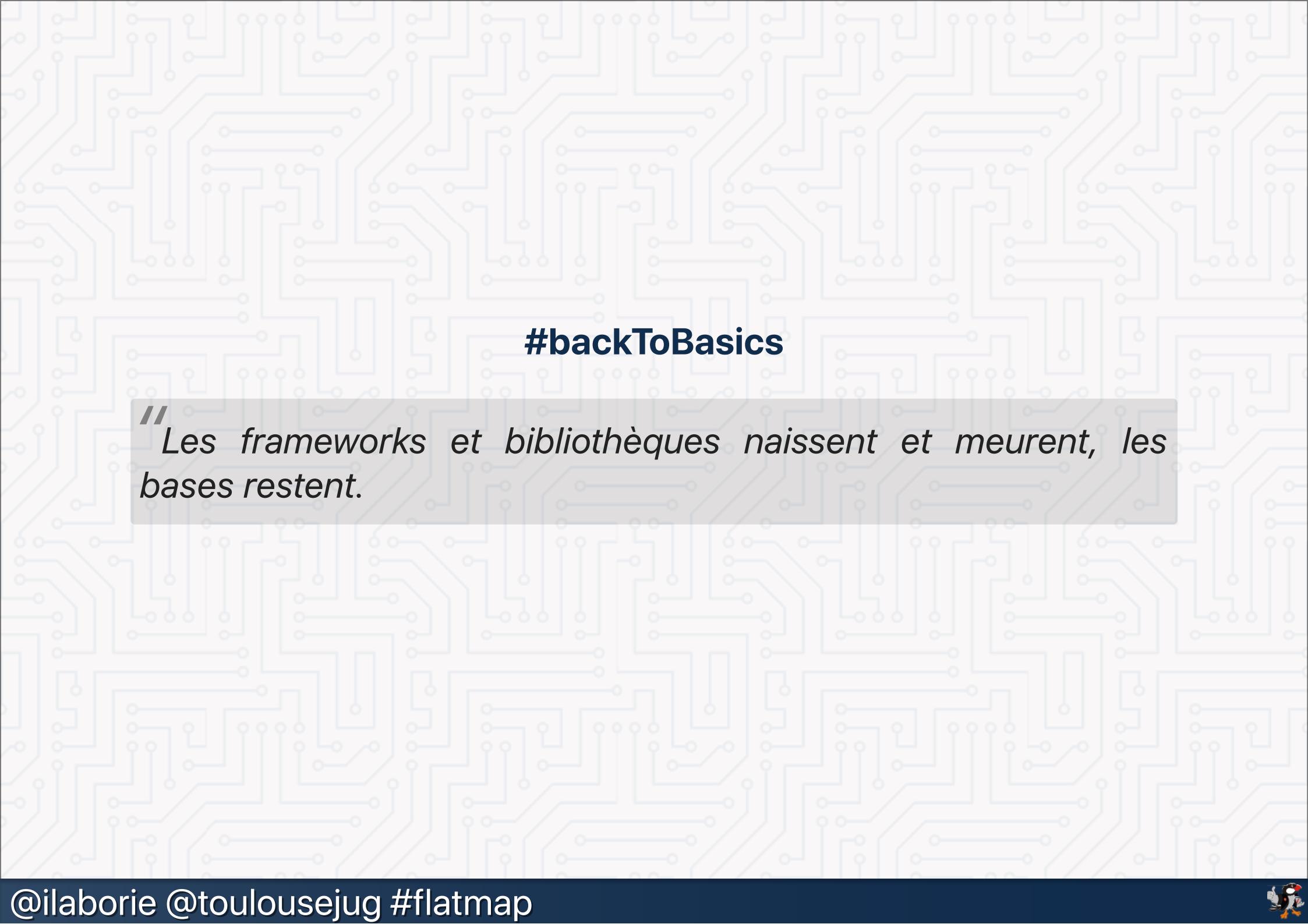
<Monkey Patch/>



# Plan

- I. **Introduction**
- II. **Anatomie d'une boucle** 
- III. **Récursion** 
- IV. **Stream** 
- V. **Qui est le meilleur ?** 
- VI. **Conclusion**



A light gray background featuring a dense grid of white circuit board traces and component pads, creating a technical and electronic theme.

## #backToBasics

“Les frameworks et bibliothèques naissent et meurent, les bases restent.





# Anatomie d'une boucle



# Transformation - Java 1.4

```
// Good old for style
public List transform(List input) {
    List results = new ArrayList();
    for (int i = 0; i < input.size(); i++) {
        Element element = (Element) input.get(i);
        Result res = transform(element);
        results.add(res);
    }
    return results;
}
```



# Transformation - Java 5

```
// Java 5 for Each
public List<Result> transform(List<Element> input) {
    List<Result> results = new ArrayList<Result>();
    for (Element element : input) {
        Result res = transform(element);
        results.add(res);
    }
    return results;
}
```



# Transformation - Java 8

```
// Java 8 forEach & lambda expression
public List<Result> transform(List<Element> input) {
    List<Result> results = new ArrayList<>();
    input.forEach(element -> {
        Result res = transform(element);
        results.add(res); // 😰
    });
    return results;
}
```



# Filtre



```
public List<Element> filter(List<Element> input) {  
    List<Element> results = new ArrayList<>();  
    for (Element element : input) {  
        if(isSomething(element)) {  
            results.add(element);  
        }  
    }  
    return results;  
}
```



# Accumulation



```
public Accumulator compute(List<Element> input) {  
    Accumulator accumulator = initialValue;  
    for (Element element : input) {  
        // accumulate :: (Accumulator, Element) → Accumulator  
        accumulator = accumulate(accumulator, element);  
    }  
    return accumulator;  
}
```



# Imbrication



```
for (Element element : input) {  
    for (Child child: element.getChildren()) {  
        // transform, filter, accumulate, ...  
    }  
}
```



# Et le reste

```
toto: for (Element element : input) {  
    tata: for (Child child : element.getChildren()) {  
        if(cond1) {  
            continue toto;  
        } else if (cond2) {  
            break tata;  
        }  
        // ...  
    }  
}
```





# Récursion



# Parcours - Java



```
public List<Result> transformR(List<Element> input) {  
    if (input.isEmpty()) { // end of recursion  
        return Collections.emptyList();  
    }  
    // Deconstruct  
    Element head = input.get(0);  
    List<Element> tail = input.subList(1, input.size() - 1);  
  
    // Handle head  
    Result transformed = transform(head);  
  
    // Recursion  
    List<Result> results = new ArrayList<>();  
    results.add(transformed);  
    results.addAll(transformR(tail));  
    return results;  
}
```



# Parcours - Kotlin



```
fun transformR(input: List<Element>): List<Result> =  
    if (input.isEmpty()) listOf()  
    else listOf(transform(input.first())) +  
        transformR(input.drop(1))
```



# Parcours - Scala



```
def transformR(input: List[Element]): List[Result] =  
  input match {  
    case Nil          => Nil  
    case head :: tail => transform(head) :: transformR(tail)  
  }
```



# Filtre & Sortie rapide - Java



```
public Element find(List<Element> input) {  
    if (input.isEmpty()) {  
        return null;  
    }  
    Element head = input.get(0);  
    if (isSomething(head)) {  
        return head;  
    }  
    List<Element> tail = input.subList(1, input.size());  
    return find(tail);  
}
```



# Récursion non terminale

$$x! = x \times (x - 1) \times \dots \times 2 \times 1$$
$$1! = 0! = 1$$

$fact(x)$

$x \times fact(x - 1)$

$x \times (x - 1) \times fact(x - 2)$

$x \times (x - 1) \times (x - 2) \times \dots$

$x \times (x - 1) \times (x - 2) \times \dots \times 2 \times 1$

$x$

$x - 1$

$x$

...

$x - 1$

$x$

1

2

...

$x - 1$

$x$



# Récursion terminale

$$fact(x) = fact(x, 1)$$
$$fact(x - 1, x \times 1)$$
$$fact(x - 2, x \times (x - 1))$$
$$fact(..., x \times (x - 1) \times (x - 2) \times ...)$$
$$fact(1, x \times (x - 1) \times (x - 2) \times ... \times 2)$$

⚠ Nécessite une optimisation par le compilateur



# Principe récursion terminale



```
tailRecFunc(scope, state) =  
  if (isFinish(scope)) computeResult(state)  
  else  
    (head, subScope) := scope  
    newState := reduce(state, head)  
    tailRecFunc(subScope, newState)
```



# Récursion terminale - Java

**GAME OVER**  
INSERT KOTLIN OR SCHLAF  
TO CONTINUE



# Récursion terminale - Kotlin

```
fun factorial(n: Int): Int {  
    tailrec fun aux(n: Int, acc: Int): Int =  
        if (n < 2) acc  
        else aux(n - 1, n * acc)  
  
    return aux(n, 1)  
}
```



# Récursion terminale - Scala

```
● ○ ■ ■  
def factorial(n: Int): Int = {  
    @tailrec  
    def aux(n: Int, acc: Int): Int =  
        if (n < 2) acc  
        else aux(n - 1, n * acc)  
  
    aux(n, 1)  
}
```





# Stream



# Création 1/2



```
// Create from a Collection
Stream<Element> s0 = col.stream();
Stream<Element> s1 = col.parallelStream();

// Create from array
Stream<Element> s2 = Arrays.stream(array);
Stream<Element> s3 = Stream.of(array);

// Manually
Stream<Element> s4 = Stream.of(elt1, elt2 /* , ... */);
Stream<Element> s5 = Stream.empty();
```



# Création 2/2

```
// Some IntStream
IntStream.range(0, 9); // 0,1,2,3,4,5,6,7,8
IntStream.rangeClosed(0, 9); // 0,1,2,3,4,5,6,7,8,9
IntStream.iterate(0, i -> i + 1); // 0,1,2,3, ...
IntStream ints = new Random().ints();

// With Path
Path path = Paths.get("plop.md");
Stream<String> lines = Files.lines(path);
Stream<Path> walk = Files.walk(path);

// With a Spliterator
Iterable<Element> elts = ...
StreamSupport.stream(eltsspliterator(), false);
```



# Transformation - map

```
// input: List<Element>
// transform : (Element) → Result
input.stream()
    .map(element -> transform(element))
    // : Stream<Result>
    // ...
;
```



# Filtre - filter

```
// input: List<Element>
// isSomething : (Element) → boolean

input.stream()
    .filter(element -> isSomething(element))
    // : Stream<Element>
    // ...
;
```

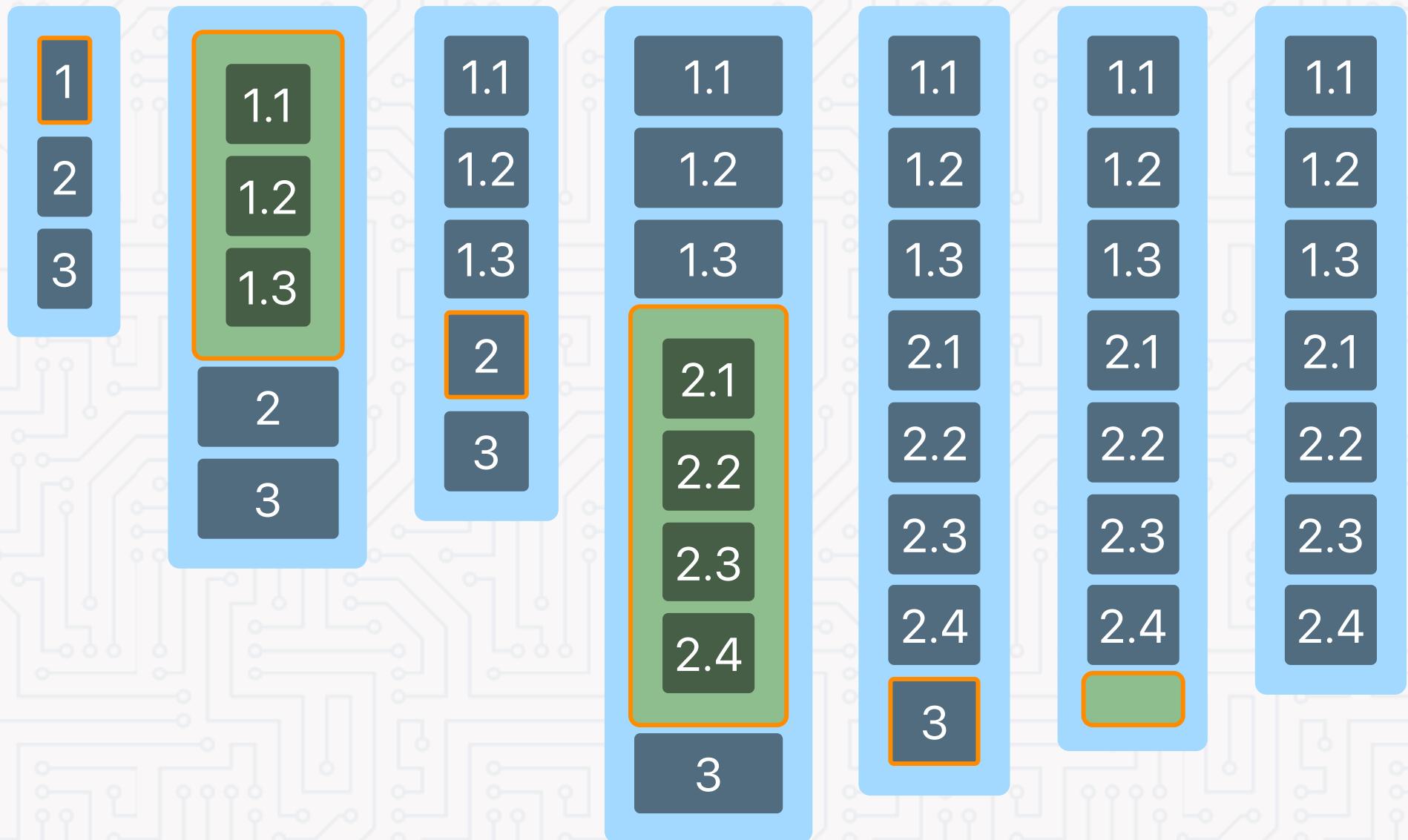


# Imbrication - flatMap 1/2

```
// input: List<Element>
// Element#getChildren : () → Collection<Child>

input.stream()
    .flatMap(element ->
        element.getChildren().stream())
    // : Stream<Child>
    // ...
;
```







```
String[] data = "lorem ... amet".split(" ");
Stream<String> stream = Arrays.stream(data)
    .map(s -> {
        System.out.print(s + ",");
        return s.toUpperCase();
    }).filter(s -> {
        System.out.print(s + ",");
        return s.startsWith("A");
    }).peek(s -> System.out.println(s));

System.out.println("Stream created");
```

lorem, ..., amet, ~~LOREM, ..., AMET, AMET~~  
Stream created

Stream created





@ilaborie @toulousejug #flatmap



# Opérations paresseuses & terminales

## Opérations paresseuses

`map`, `filter`, `flatMap`, `peek`, `distinct`, `limit` et `skip`,  
...

## Opérations terminales

`reduce` et `collect`, `findAny` et `findFirst`, `foreach`,  
`count`, ...



# Accumulation - Reduce 1/2

```
// input: List<Element>
// accumulate : (Accumulator, Element) → Accumulator
// Accumulator#merge : (Accumulator) → Accumulator

Accumulator result =
    input.stream()
        .reduce(
            new Accumulator(),
            (acc, elt) -> accumulate(acc, elt),
            (acc1, acc2) -> acc1.merge(acc2)
        );
```



# Accumulation - Reduce 2/2



## Cas particulier

Si `Element` = `Accumulator`, on peut utiliser un `reduce`  
Les `count`, `min`, `max`, `sum`, ... sont des réductions particulières

```
// input: List<Element>
// Element#merge : (Element) → Element

Element result =
    input.stream()
        .reduce(
            new Element(),
            (elt1, elt2) -> elt1.merge(elt2)
        );
```



# Accumulation - collect & Collectors

Les Stream#collect sont justes une généralisation du reduce

```
// T: Type du Stream  
// A: Type de l'accumulateur  
// R: Type du résultat  
interface Collector<T,A,R> {  
    Supplier<A> supplier()  
    BiConsumer<A,T> accumulator();  
    BinaryOperator<A> combiner();  
    Function<A,R> finisher();  
    // ...  
}
```



# Collectors classiques

Dans `java.util.stream.Collectors`

`toSet`, `toList` pour construire une collection

`toMap` pour construire un `Map`

`groupBy` pour grouper en une `Map<K, List<V>>`

`joining` pour construire une `String`

`summarizingInt`, `summarizingDouble`, `summarizingLong`

pour les statistiques

...



# Et l'index ?

Et si j'ai besoin de l'*index* ?

- 😢 pas faisable facilement et *proprement* en Java
- ✌️ mais il y a Kotlin et Scala...



# Nouveautés Java 9+

## Java 9

`Stream#takeWhile` et `Stream#dropWhile`

`Stream#iterate` avec un prédictat

`Stream#ofNullable`

`Optional#stream`

`Collectors#flatMap` et `Collectors#filtering`

`String#chars` et `String#codePoints`

## Java 10

`Collectors#toUnmodifiableXXX` pour les `List`, `Set` et `Map`

## Java 11

`String#lines`

## Java 12

`Collectors#teeing`



# Bilan Stream



## À proscrire

Les effets de bord ! (on tolère les *logs* dans le `peek`)

Les opérations non associatives dans des `Stream` parallèles

Les streams sur des `Integer`, `Double`, `Long`



Sans bonne raison, ne faites pas de `Stream` parallèle

La lisibilité est importante



On peut utiliser intelligemment les aspects paresseux

# Bilan Stream - Java



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

`Optional<T> reduce(BinaryOperator<T> accumulator)`  
`IntStream, DoubleStream, LongStream, avec mapToObj,`  
`mapToXXX, ...`

Le *boilerplate*, par exemple `.collect(Collectors.toList()),`  
`Collectors.groupBy, ...`

Pas de tuples en Java

Des 🐛 dans le *lazy* du `flatMap`, voir ➡ [Java Stream API was Broken](#)

Before JDK10 <<https://4comprehension.com/java-stream-api-was-broken-before-jdk10/>>



# Bilan Stream - Kotlin



API *lazy* avec les **Sequence** ou non directement sur les collections

API collection **immutable** ou mutable

🎭 utilise juste les classes de Java



# Bilan Stream - Scala



API *lazy* avec les Stream ou non directement sur les collections

API collection **immutable** ou mutable

Pas de réutilisation de Java

API de ➡ Stream <<https://www.scala-lang.org/api/2.12.3/scala/collection/immutable/Stream.html>> avec la possibilité de construction récursive



De gros changements arrivent dans la ➡ 2.13 <<https://www.scala-lang.org/blog/2018/06/13/scala-213-collections.html>>



# Scala for



```
val monkeyFood = (for {
    fruit ← fruits
    it = emojify(fruit)
    if "🍌" = it || "🥥" = it
} yield it)
```

Le `for` de Scala est du sucre syntaxique qui produit des `map`,  
`filter`, `flatMap`

Du coup on peut l'utiliser sur d'autre objects qui ont `map`, `filter`,  
`flatMap`





# Qui est le meilleur ?



# Relation d'ordre

Si on veut déterminer le meilleur, il nous faut une relation d'ordre, laquelle ?

- 🏎 le plus rapide ?
  - 💾 le moins couteux en mémoire ?
  - 📦 le plus maintenable ?
  - ⠇ le plus lisible ?
- ...



# Java & performances

faire des micro-benchmark en Java, c'est pas évident

🔥 la JVM à besoin de chauffer (JIT)

➡ JMH <<http://openjdk.java.net/projects/code-tools/jmh/>>

➡ Microbenchmarking in Java with JMH (5 articles)

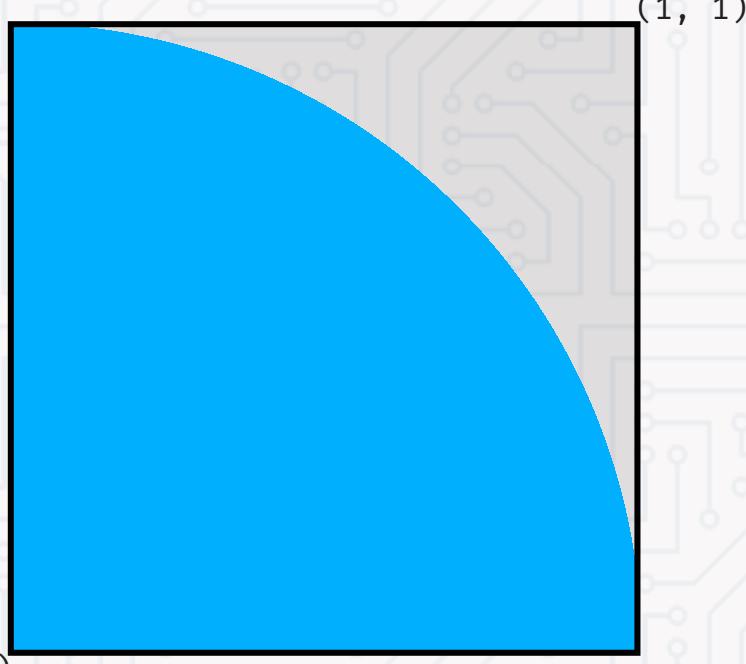
<<https://daniel.mitterdorfer.name/articles/2014/jmh-microbenchmarking-intro/>>



# Rewrite everything in Rust



# MonteCarlo $\pi$



$$\frac{\frac{\pi \cdot r^2}{4}}{r^2} = \frac{\pi}{4} \approx \frac{\text{nb. in}}{\text{nb. total}} \text{ avec } r = 1$$

10 points



# MonteCarlo - Point

```
● ● ●
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) { this.x = x; this.y = y; }

    public boolean inCircle() {
        return (x * x) + (y * y) <= 1;
    }

    private static final Random rnd = new Random();
    public static Point newPoint() {
        return new Point(rnd.nextDouble(), rnd.nextDouble());
    }

    public static double compute(int count, int inCircle) {
        return ((double) inCircle / count) * 4;
    }
}
```



# MonteCarlo - Java for

```
public static double monteCarloFor(int count) {  
    int inCircle = 0;  
    for (int i = 0; i < count; i++) {  
        Point p = newPoint();  
  
        if (p.inCircle()) {  
            inCircle++;  
        }  
    }  
    return compute(count, inCircle);  
}
```



# MonteCarlo - Java stream



```
public static double monteCarloStream(int count) {  
    int inCircle = (int) Stream.generate(Point::newPoint)  
        .limit(count)  
        .filter(Point::inCircle)  
        .count();  
  
    return compute(count, inCircle);  
}
```



# MonteCarlo - Java stream parallèle

```
public static double monteCarloStreamParallel(int count) {  
    int inCircle = (int) Stream.generate(Point::newPoint)  
        .unordered()  
        .parallel()  
        .limit(count)  
        .filter(Point::inCircle)  
        .count();  
  
    return compute(count, inCircle);  
}
```



# MonteCarlo - Kotlin for

```
fun monteCarloFor(count: Int): Double {  
    var inCircle = 0  
    for (i in 0 until count) {  
        val p = newPoint()  
  
        if (p.inCircle()) {  
            inCircle++  
        }  
    }  
    return compute(count, inCircle)  
}
```



# MonteCarlo - Kotlin tailrec

```
● ● ●   
fun monteCarloRecursion(count: Int): Double {  
    tailrec fun aux(count: Int, inCircle: Int): Int =  
        if (count == 0) inCircle  
        else {  
            val p = newPoint()  
            aux(count - 1, if (p.inCircle()) inCircle + 1 else inCircle)  
        }  
  
    val inCircle = aux(count, 0)  
    return compute(count, inCircle)  
}
```

# MonteCarlo - Kotlin collection

```
fun monteCarloCollection(count: Int): Double {  
    val inCircle = (1..count)  
        .map { newPoint() }  
        .count { it.inCircle() }  
  
    return compute(count, inCircle)  
}
```



# MonteCarlo - Kotlin séquence

```
fun monteCarloSequence(count: Int): Double {  
    val inCircle = generateSequence { newPoint() }  
        .take(count)  
        .count { it.inCircle() }  
  
    return compute(count, inCircle)  
}
```



# MonteCarlo - Kotlin séquence parallèle

```
fun monteCarloSequenceParallel(count: Int): Double {  
    val inCircle = sequence {  
        yieldAll(generateSequence { newPoint() })  
    }  
    .take(count)  
    .count { it.inCircle() }  
  
    return compute(count, inCircle)  
}
```



# MonteCarlo - Scala for

```
def monteCarloFor(count: Int): Double = {  
    var inCircle = 0  
    for (_ ← 1 to count) {  
        val p = newPoint()  
  
        if (p.inCircle()) {  
            inCircle += 1  
        }  
    }  
    compute(count, inCircle)  
}
```



# MonteCarlo - Scala tailrec

```
● ○ ●
def monteCarloRecursion(count: Int): Double = {
  @tailrec
  def aux(count: Int, inCircle: Int): Int =
    if (count == 0) inCircle
    else {
      val p = newPoint()
      aux(count - 1, inCircle + (if (p.inCircle()) 1 else 0))
    }

  val inCircle = aux(count, 0)
  compute(count, inCircle)
}
```



# MonteCarlo - Scala collection

```
def monteCarloCollection(count: Int): Double = {  
    val inCircle = (1 to count)  
        .map(_ => newPoint())  
        .count(_.inCircle())  
  
    compute(count, inCircle)  
}
```



# MonteCarlo - Scala stream

```
● ● ● ● ●  
def monteCarloStream(count: Int): Double = {  
    val inCircle = Stream.fill(count) {  
        newPoint()  
    }  
    .count(_.inCircle())  
  
    compute(count, inCircle)  
}
```



# MonteCarlo - Scala stream parallèle

```
def monteCarloStreamParallel(count: Int): Double = {  
    val inCircle = Stream.fill(count) {  
        newPoint()  
    }  
    .par  
    .count(_.inCircle())  
  
    compute(count, inCircle)  
}
```



# Disclaimer

“REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on why the numbers are the way they are. Use **profilers** (see `-prof`, `-lprof`), design factorial experiments, perform baseline and negative tests that provide experimental control, make sure the benchmarking environment is safe on JVM/OS/HW level, **ask for reviews** from the domain experts.

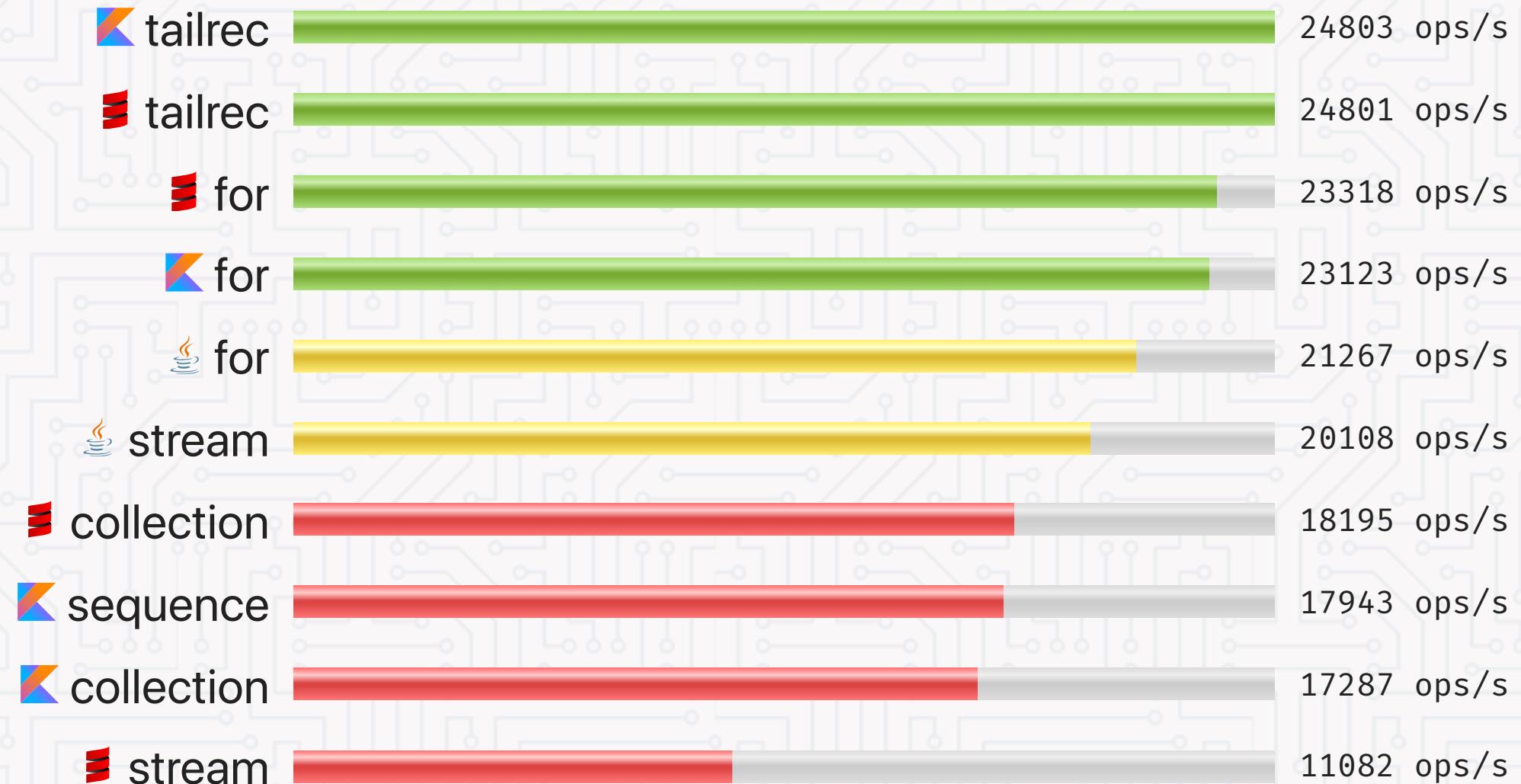
**Do not assume the numbers tell you what you want them to tell.**



venez lire, tester, critiquer, proposer des PR sur le  
➡ dépôt Github <<https://github.com/ilaborie/refactorLoops>>



# MonteCarlo - performance 1000 points

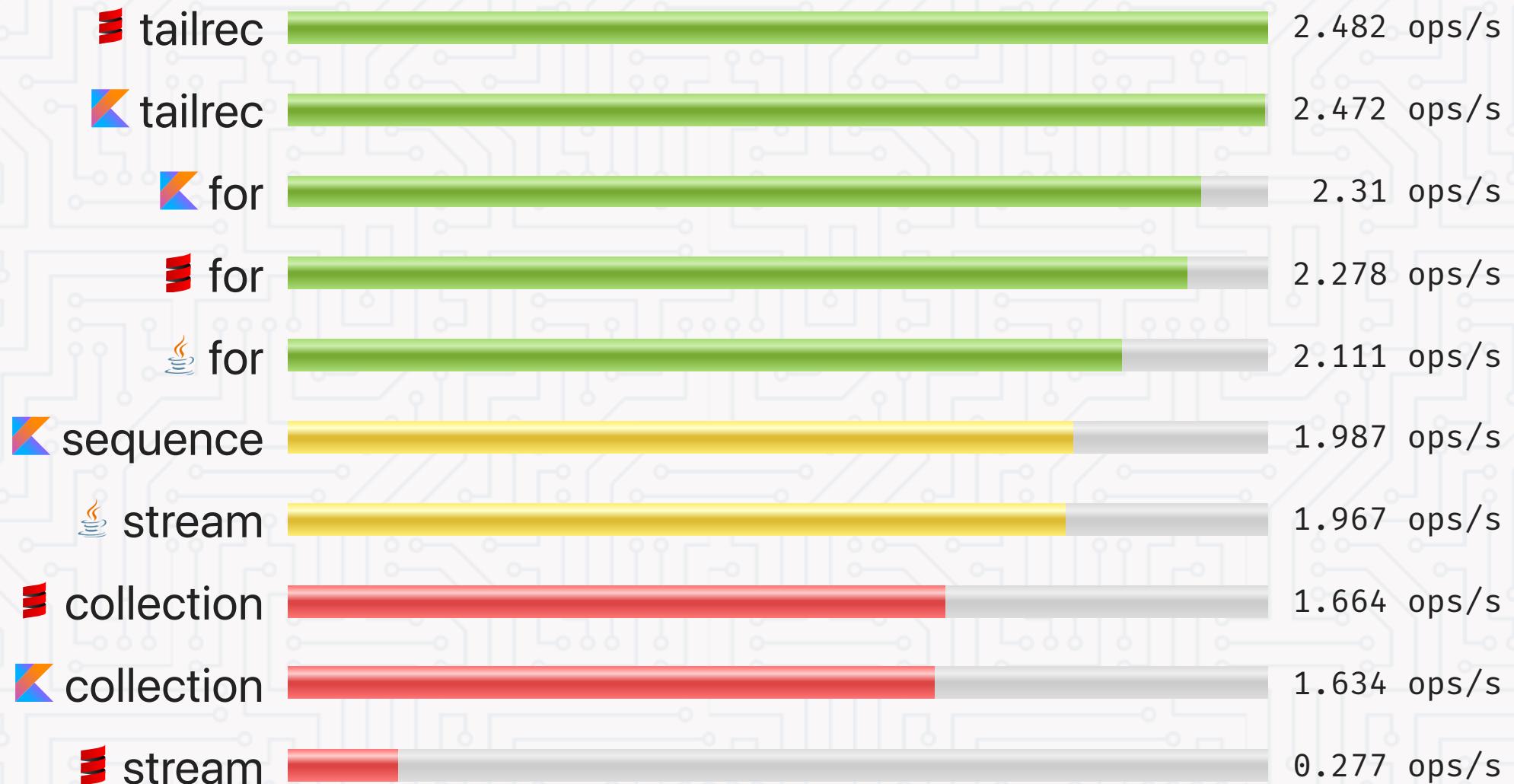


**1\_000 points sur OpenJDK (HotSpot) 8.0.202**

tailrec  $\approx 40.3\mu s$  > for  $\approx 47.0\mu s$  > stream  $\approx 49.7\mu s$  > collection  $\approx 55.0\mu s$



# MonteCarlo - performance 10M points

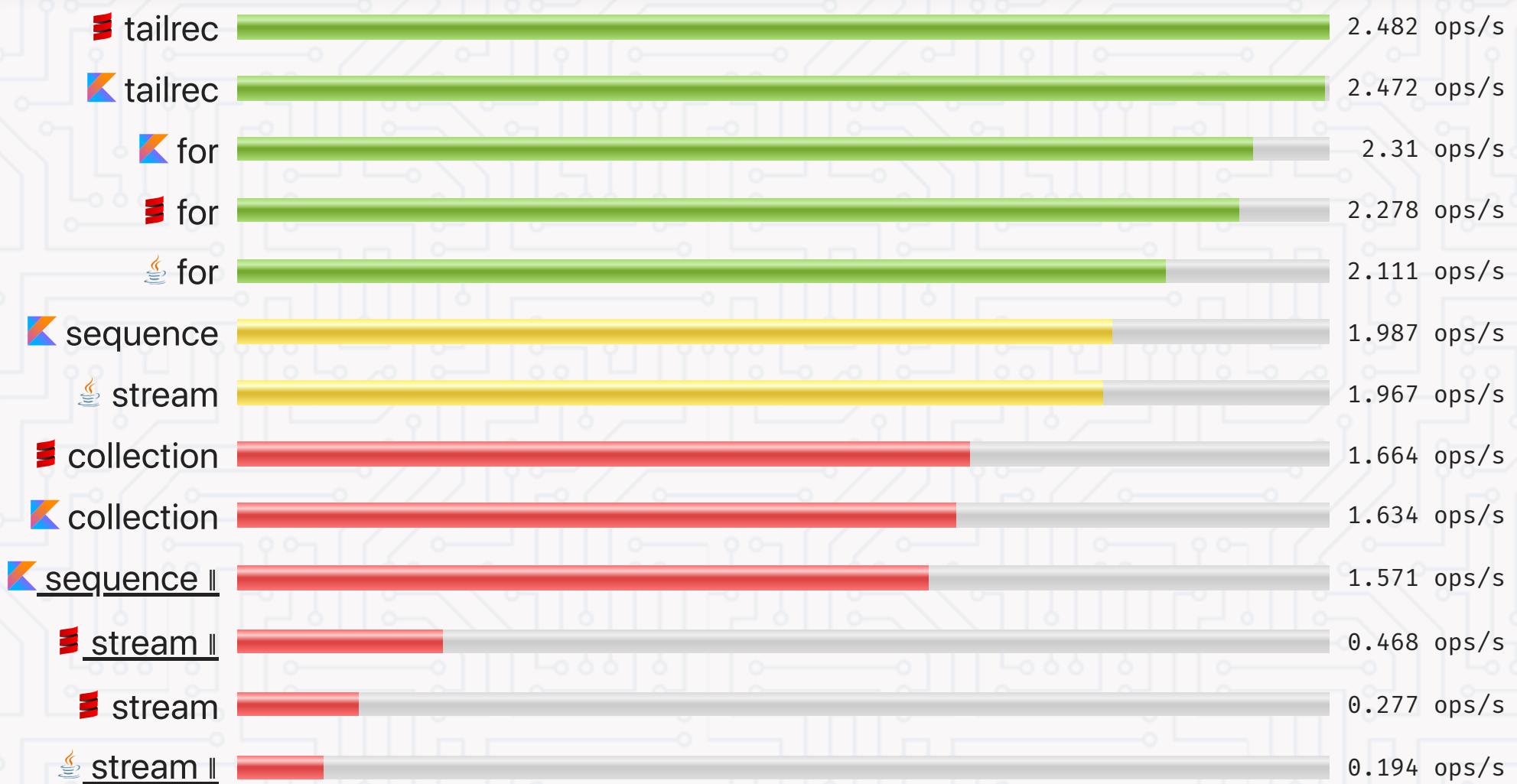


**10\_000\_000 points sur OpenJDK (HotSpot) 8.0.202**

tailrec  $\approx 403ms$  > for  $\approx 474ms$  > stream  $\approx 508ms$  > collection  $\approx 601ms$



# MonteCarlo - performance parallèle

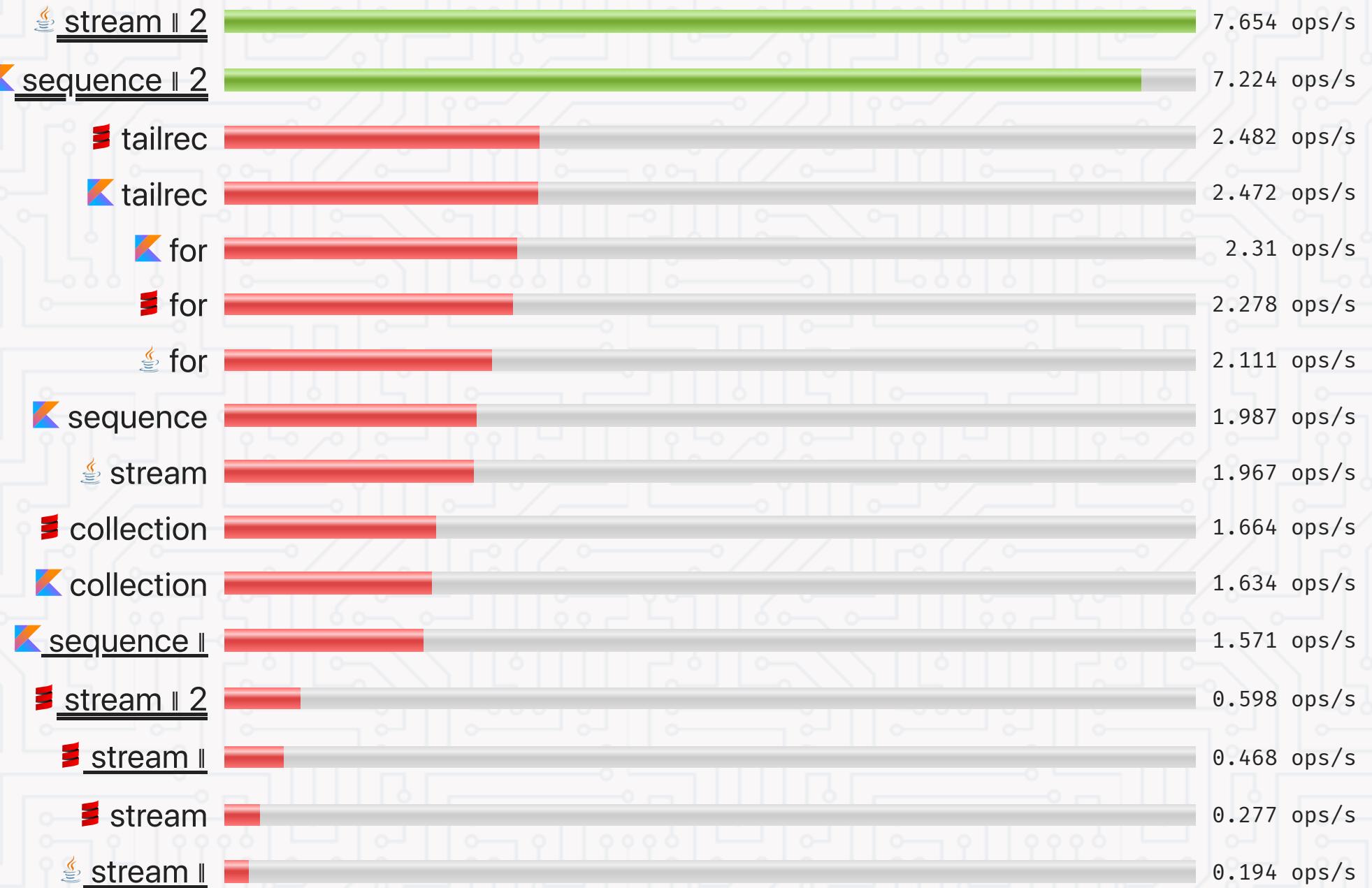


10\_000\_000 points sur OpenJDK (HotSpot) 8.0.202



le code parallèle





10\_000\_000 points sur OpenJDK (HotSpot) 8.0.202



# SplittableRandom



depuis Java 8 ➔ SplittableRandom

[<https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html>](https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html)



# Separation of Concerns

```
List<String> errors = new ArrayList<>();  
int errorCount = 0;  
File file = new File(fileName);  
String line = file.readLine();  
while(errorCount < 40 && line != null) {  
    if(line.startsWith("ERROR")) {  
        errors.add(line);  
        errorCount++;  
    }  
    line = file.readLine();  
}
```

```
List<String> errors =  
    Files.lines(Paths.get(fileName))  
        .filter(line -> line.startsWith("ERROR"))  
        .limit(40)  
        .collect(toList());
```

➤ @mariofusco <<https://twitter.com/mariofusco/status/571999216039542784>>

➤ Lazy Java par Mario Fusco <<https://www.youtube.com/watch?v=84MfG4tp30s>>



# Prédisposition aux



```
● ● ● // Good old for style
public List transform(List input) {
    List results = new ArrayList();
    for (int i = 0; i < input.size(); i++) {
        Element element = (Element) input.get(i);
        Result res = transform(element);
        results.add(res);
    }
    return results;
}
```



# Prédisposition aux 🐛 dangereux



```
// Java 8 forEach & lambda expression
public List<Result> transform(List<Element> input) {
    List<Result> results = new ArrayList<>();
    input.forEach(element -> {
        Result res = transform(element);
        results.add(res); // 😰
    });
    return results;
}
```



# Pas tous du même avis

➡ 3 Reasons why You Shouldn't Replace Your for-loops by Stream  
forEach <<https://www.javacodegeeks.com/2015/12/3-reasons-shouldnt-replace-loops-stream-foreach.html>>



# Exemple colonnes d'Excel - for

A, B, ..., Z, AA, AB, ..., ZZ, AAA, ...

```
fun excelColumn(n: Int): String {  
    var chars = ""  
    var current = n  
  
    while (true) {  
        current--  
        chars = ('A'.toInt() + current % 26).toChar() + chars  
        if (current < 26) break  
        current /= 26  
    }  
    return chars  
}
```



# Exemple colonnes d'Excel - récursif

```
● ● ●   
fun excelColumn(n: Int): String {  
    tailrec fun aux(n: Int, acc: String): String {  
        val current = n - 1  
        val rest = current % 26  
        val result = ('A'.toInt() + rest).toChar() + acc  
        return if (current < 26) result  
        else aux(current / 26, result)  
    }  
    return aux(n, "")  
}
```



# Exemple colonnes d'Excel - séquence

```
fun excelColumn(n: Int): String =  
    generateSequence<Pair<Int?, String>>(n to "") { (n, acc) →  
        if (n == null) null  
        else {  
            val current = n - 1  
            val result = ('A'.toInt() + current % 26).toChar() + a  
  
            if (current < 26) null to result  
            else (current / 26) to result  
        }  
    }.last().second
```



# Conclusion



# Bilan

goto firstQuote

“Les frameworks et bibliothèques naissent et meurent, les bases et les styles de programmation restent.

“Il suffit de choisir le langage, les frameworks, et les bibliothèques qui correspondent aux styles.  
Choisissez un style qui correspond à vos contraintes et vos goûts.

➡ Si vous avez du mal à choisir <<http://www.flatmapthatshit.com/>>

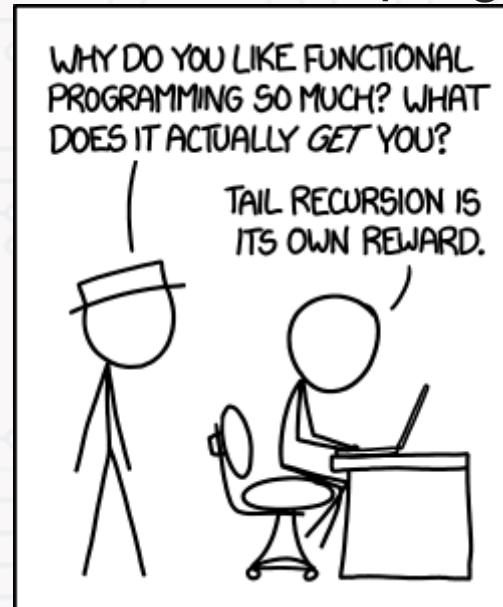


Lambda et fonctions d'ordre supérieur

Pas d'effet de bord

Immutabilité

⇒ Ceci est une présentation sur la programmation fonctionnelle



➡ <https://xkcd.com/1270/> <<https://xkcd.com/1270/>>





## ➤ Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire <<http://eprints.eemcs.utwente.nl/7281/01/db-utwente-40501F46.pdf>> - Erik Meijer and J. Hughes and M.M. Fokkinga and Ross Paterson" - 1991

*"We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all example functions in Bird and Wadler's "Introduction to Functional Programming" can be expressed using these operators.*

 **Robert Virding** @rvirding · Sep 28, 2013 

Replying to @C0deAttack  
@C0deAttack @viktorklang Anything you can write with a loop you write better with recursion, plus things which really suck with loops.

 **Erik Meijer** @headinthebox

@rvirding @C0deAttack @viktorklang Recursion is the GOTO of functional programming (from 1991 [citeseerx.ist.psu.edu/viewdoc/summary...](http://citeseerx.ist.psu.edu/viewdoc/summary...)). 

21 2:02 AM - Sep 29, 2013 

 See Erik Meijer's other Tweets 



# Crafters

- ? Quand vous codez, posez-vous des questions !
  - 🔪 Aiguissez votre esprit critique !
  - 🧠 Partagez vos questionnements, vos solutions, vos idées farfelues !





**ou ?**

