PROGRAMMATION FONCTIONNELLE

Fabrice Legond-Aubry et Pascal Poizat

Juillet-Août 2024

- Introduction
- Pureté
- Immutabilité
- Ordre supérieur
- Programmes séquentiels
- Absence de valeur et erreurs
- Types de données algébriques
- Effets de bord
- Flux et évaluation paresseuse
- Concurrence
- Test
- Synthèse des notations

ORDRE SUPÉRIEUR

PROBLÈME : WORD SCORE

- score d'un mot : nb caractères != a
- tri d'une liste de mots par score descendant

on dispose de :

Java
```java
static int score(String word) {
    return word.replaceAll("a", "").length();
}
```

idée : utiliser un comparateur

Java
```java
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

`xs.sort(comp)` avec `xs` liste et `comp` comparateur

SOLUTION 1

Java

```java
static Comparator<String> scoreComparator =  new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(score(o2), score(o1));
    }
};
static List<String> rankedWords(List<String> words) {
    words.sort(scoreComparator);
    return words;
}
public static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    System.out.println(rankedWords(words));
}
```

- mutation de `words`
- dépendance à `scoreComparator` non paramètre
- verbosité de `scoreComparator`

INTERFACES ET LAMBDAS

- **interface fonctionnelle** (IF) = 1 méthode abstraite
  - il peut y avoir des méthodes non abstraites
- une **lambda** (fonction) est une fonction anonyme
  - généralement : `(T1 x, T2 y) -> { ...; return r;}`
  - sous conditions : `(x, y) -> r`
- une lambda peut instancier une IF
  - `@FunctionalInterface Itf<T> o = new Itf<T>() { @Override public U method(T1 x, T2 y) {...} };`
  - `Itf<T> o = (x, y) -> {...};`

SOLUTION 2

- évitement de la mutabilité
- comparateur en paramètre
- utilisation d'une lambda fonction

Java

```java
static Comparator<String> scoreComparator =
    (String o1, String o2) -> Integer.compare(score(o2), score(o1));

static List<String> rankedWords(Comparator<String> comp, List<String> words) {
    List<String> rankedWords = new ArrayList<>(words);
    rankedWords.sort(comp);
    return rankedWords;
}
public static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    System.out.println(rankedWords(scoreComparator, words));
}
```

PROBLÈME : WORD SCORE V2

- score **de base** d'un mot : nb caractères != a
- tri d'une liste de mots par score descendant
- bonus : +5 si le mot contient un c
- l'ancienne façon de calculer doit rester supportée

SOLUTION

Java

```java
static int score(String word) { ... }
static int scoreWithBonus(String word) {
    int base = score(word);
    return (word.contains("c")) ? base + 5 : base;
}
static Comparator<String> scoreComparator =
    (o1, o2) -> Integer.compare(score(o2), score(o1));

static Comparator<String> scoreWithBonusComparator =
    (o1, o2) -> Integer.compare(scoreWithBonus(o2), scoreWithBonus(o1));

public static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    System.out.println(rankedWords(scoreComparator, words));
    System.out.println(rankedWords(scoreWithBonusComparator, words));
}
```

aussi (si lambda ad-hoc)

Java

```java
System.out.println(rankedWords((o1, o2) -> Integer.compare(score(o2), score(o1)),
```

FONCTIONS COMME OBJETS

en Java on dispose du type `Function<T,U>`
pour représenter une fonction $f : T \to U$

Java

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ...
}
```

on peut ensuite les créer comme suit :

Java

```java
static Function<String, Integer> score = w -> w.replaceAll("a", "").length();
```

principe : les fonctions sont stockées comme des valeurs

API FONCTION EN JAVA

`@FunctionalInterface`

- `Function<T, U>`: *T* → *U* avec `apply`
- `BiFunction<T, U, V>`: (*T*,*U*) → *V* avec `apply`
- `UnaryOperator<T>`: *T* → *T* avec `apply`
- `BinaryOperator<T>`: (*T*,*T*) → *T* avec `apply`
- `Predicate<T>`: *T* → *Boolean* avec `test`
- `Supplier<T>`: () → *T* avec `get`
- `Consumer<T>`: *T* → () avec `accept`

on note aussi ici

- `Comparator<T>`: (*T*,*T*) → *Int* avec `compare`

FONCTIONS VS MÉTHODES

méthodes : appel direct

score : _ • String → Int

Java

```java
static int score(String w) { return w.replaceAll("a", "").length(); }
score("scala");
```

fonctions : utilisation de `apply`

score : String → Int

Java

```java
static Function<String, Integer> score = w -> w.replaceAll("a", "").length();
score.apply("scala");
```

FONCTIONS VS MÉTHODES

une méthode peut être utilisée comme fonction
par utilisation du référencement (`::`)

soit `work(Function<T,U> f)`, on peut utiliser :

- `work(f)`
  - si `f` est de type `Function<T,U>` ($f : T \to U$)
- `work(C::m)`
  - si `U m(T t)` est une méthode de classe de `C` ($m : C \bullet T \to U$)
- `work(c::m)`
  - si `U m(T t)` est une méthode d'instance de `C` ($m : C \hookrightarrow T \to U$) et `c: C`

(modulo le sous-typage aussi)

FONCTIONS PARAMÈTRES ET RETOURS

les fonctions peuvent être passées en **paramètres** (ordre supérieur) et retournées comme **résultats**

$$scoreWithBonus : Classe \bullet String \rightarrow Int$$

$$scoreWithBonusFunction : String \rightarrow Int$$

$$genComparator : \_ \bullet (String \rightarrow Int) \rightarrow ((String,String) \rightarrow Int)$$

$$rankedWords : \_ \bullet (((String,String) \rightarrow Int), List[String]) \rightarrow List[String]$$

Java

```java
static scoreWithBonus(String word) { ... }
static Function<String, Integer> scoreWithBonusFunction = ...
static Comparator<String> genComparator(Function<String, Integer> scoring) {
    return (o1, o2) -> Integer.compare(scoring.apply(o2), scoring.apply(o1));
}

System.out.println(rankedWords(genComparator(scoreWithBonusFunction), words));
System.out.println(rankedWords(genComparator(w -> scoreWithBonus(w)), words));
System.out.println(rankedWords(genComparator(Classe::scoreWithBonus), words));
```

FONCTIONS PARAMÈTRES ET RETOURS

les fonctions peuvent être passées en **paramètres** (ordre supérieur) et retournées comme **résultats**

$$scoreWithBonus : Classe \bullet String \rightarrow Int$$

$$scoreWithBonusFunction : String \rightarrow Int$$

$$rankedWords : \_ \bullet (String \rightarrow Int, List[String]) \rightarrow List[String]$$

Java

```java
static scoreWithBonus(String word) { ... }
static Function<String, Integer> scoreWithBonusFunction = ...
static List<String> rankedWords(Function<String, Integer> scoring, List<String> w
    List<String> rankedWords = new ArrayList<>(words);
    rankedWords.sort((o1, o2) -> Integer.compare(scoring.apply(o2), scoring.apply
    return rankedWords;
}

System.out.println(rankedWords(scoreWithBonusFunction, words));
System.out.println(rankedWords(w -> scoreWithBonus(w), words));
System.out.println(rankedWords(Classe::scoreWithBonus, words));
```

EXERCICE

Java

```java
class Classe {
    private int base;
    public Classe(int base) { this.base = base; }
    // méthode d'intance
    public int m1(String w) { return w.length() + base; }
    // méthode de classe
    static int m2(String w) { return w.length(); }
    // fonction
    static Function<String, Integer> m3 = w -> w.length();

    static Function<Function<String, Integer>, Function<String, String>> work =
        f -> w -> String.format("%s", f.apply(w));
}
```

typez et déchiffrez les appels suivants

Java

```java
Classe objet = new Classe(3);
System.out.println(Classe.work.apply(objet::m1).apply("scala"));
System.out.println(Classe.work.apply(Classe::m2).apply("scala"));
System.out.println(Classe.work.apply(Classe.m3).apply("scala"));
```

EXERCICE : WORD SCORE V3

- bonus : -7 si le mot contient un s
- les anciennes façons de calculer doivent rester supportées

pistes

- définition d'une nouvelle fonction

  String → Int
- utilisation de trois fonctions pour score / bonus / pénalité

PROBLÈMES AVEC JAVA

- Java permet la définition de lambdas
- Java a une API Fonction

mais

- Java est verbeux, par exemple :
  - `Function<T, U>` pour $T \to U$
  - `scoring.apply(o1)` pour $scoring(o1)$
- Java a par défaut une API liste mutable

(on verra des solutions à cela plus tard)

WORD SCORE EN SCALA

- utilisation de `sortBy`

  `List[A].sortBy[B](f: A => B)(implicit ord: Ordering[B]): List[A]`

  List[*A*] ↪ (*A*→*B*) → *List*[*A*]

  ! tri par ordre croissant

Scala

```scala
def score(word: String): Int = word.replaceAll("a", "").length

def rankedWords(scoring: String => Int, words: List[String]): List[String] = {
    def negativeScoring(word: String): Int = -scoring(word)
    words.sortBy(negativeScoring)
}
```

WORD SCORE EN SCALA (AMÉLIORÉ)

- style plus déclaratif
  trier **puis** inverser

Scala

```scala
def score(word: String): Int = word.replaceAll("a", "").length

def rankedWords(scoring: String => Int, words: List[String]): List[String] = {
    words.sortBy(scoring).reverse
}
```

à comparer à

Java

```java
static score(String word) { return word.replaceAll("a", "").length(); }

static List<String> rankedWords(Function<String, Integer> scoring, List<String> w
    List<String> rankedWords = new ArrayList<>(words);
    rankedWords.sort((o1, o2) -> Integer.compare(scoring.apply(o2), scoring.apply
    return rankedWords;
}
```

SOLUTION WORD SCORE V3

Scala

```scala
object WordScore {
  def score(word: String): Int = word.replaceAll("a", "").length
  def bonus(word: String): Int = if (word.contains("c")) 5 else 0
  def malus(word: String): Int = if (word.contains("s")) 7 else 0

  def rankedWords(scoring: String => Int, words: List[String]): List[String] = {
    words.sortBy(scoring).reverse
  }

  def test(): Unit = {
    val words: List[String] = List("ada", "haskell", "scala", "java", "rust")
    println(rankedWords(w => score(w) + bonus(w) - malus(w), words))
  }
}
```

WORD SCORE V4

- on souhaite connaître le score de chaque mot
- aucun changement de fonction existante

idée

- utiliser `map`

  `List[A].map(f: A => B): List[B]`

  List[*A*] ↪ (*A*→*B*) → *List*[*B*]

Scala

```scala
def wordScores(scoring: String => Int, words: List[String]): List[Int] = {
    words.map(scoring)
}

def test(): Unit = {
    val words: List[String] = List("ada", "haskell", "scala", "java", "rust")
    println(wordScores(w => score(w) + bonus(w) - malus(w), words))
}
```

EN JAVA ?

Java

```java
static List<Integer> wordScores(Function<String, Integer> scoring, List<String> w
    List<Integer> wordScores = new ArrayList<>(words.size());
    for (String word: words) {
        wordScores.add(scoring.apply(word));
    }
    return wordScores;
}

static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    System.out.println(wordScores(w -> score(w) + bonus(w) - malus(w), words))
}
```

(on verra un `map` plus tard en Java)

EXERCICE

utiliser `map` pour réaliser les fonctions suivantes sur les éléments d'une liste

- longueurs ("scala" donne 5)
- nombre de lettres 's' ("scala" donne 1)
- inversion du signe (1 donne -1)
- doublement (1 donne 2)

WORD SCORE V5

- liste de mots dont le score est > 1
- aucun changement de fonction existante

idée

- utiliser `filter`

  `List[A].filter(p: A => Boolean): List[A]`

  List[*A*] ↪ (*A*→*Boolean*) → *List*[*A*]

Scala

```scala
def highScoringWords(scoring: String => Int, words: List[String]): List[String] =
    words.filter(w => scoring(w) > 1)
}

def test(): Unit = {
    val words: List[String] = List("ada", "haskell", "scala", "java", "rust")
    println(highScoringWords(w => score(w) + bonus(w) - malus(w), words))
}
```

EN JAVA ?

Java

```java
static List<String> highScoreWords(Function<String, Integer> scoring, List<String
    List<String> highScoreWords = new ArrayList<>(words.size());
    for (String word: words) {
        if (scoring.apply(word) > 1) {
            highScoreWords.add(word);
        }
    }
    return highScoreWords;
}

static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    System.out.println(highScoreWords(w -> score(w) + bonus(w) - malus(w), words)
}
```

(on verra un `filter` en Java plus tard)

EXERCICE

utiliser `filter` pour réaliser les fonctions suivantes sur les éléments d'une liste

- taille < 5 ("scala" KO, "java" OK)
- plus d'un 'l' ("scala" KO, "haskell" OK)
- nombres pairs
- nombres > 4

WORD SCORE V6

- liste de mots dont le score est > n
- aucun changement de fonction existante

idée

- rajouter un paramètre

Scala
```
def highScoringWords(scoring: String => Int, words: List[String], n: Int): List[S
    words.filter(w => scoring(w) > n)
}
```

problème ?

Scala
```
def test(): Unit = {
    val words: List[String] = List("ada", "haskell", "scala", "java", "rust")
    println(highScoringWords(w => score(w) + bonus(w) - malus(w), words, 1))
    println(highScoringWords(w => score(w) + bonus(w) - malus(w), words, 0))
    println(highScoringWords(w => score(w) + bonus(w) - malus(w), words, 5))
```

WORD SCORE V6 (AMÉLIORATION)

- utiliser une fonction annexe

$$(String \to Int) \to ((List[String], Int) \to List[String])$$

Scala

```scala
def highScoringWords2(scoring: String => Int): (List[String], Int) => List[String
    (words: List[String], n: Int) => words.filter(w  => scoring(w) > n)
}

def test(): Unit = {
    val words: List[String] = List("ada", "haskell", "scala", "java", "rust")
    val highScoringSBM = highScoringWords2(w => score(w) + bonus(w) - malus(w))
    println(highScoringSBM(words, 1))
    println(highScoringSBM(words, 0))
    println(highScoringSBM(words, 5))
}
```

EN JAVA ?

Java

```java
static BiFunction<List<String>, Integer, List<String>> highScoringWords2(Function
    return (words, n) -> {
        List<String> highScoreWords = new ArrayList<>(words.size());
        for (String word: words) {
            if (scoring.apply(word) > n) { highScoreWords.add(word); }
        }
        return highScoreWords;
    };
}
public static void main(String[] args) {
    List<String> words = Arrays.asList("ada", "haskell", "scala", "java", "rust")
    BiFunction<List<String>, Integer, List<String>> highScoringSBM = highScoringW
    System.out.println(highScoringSBM.apply(words, 1));
    System.out.println(highScoringSBM.apply(words, 0));
    System.out.println(highScoringSBM.apply(words, 5));
}
```

EXERCICE

réalisez les fonctions suivantes sur une liste

- nombres > 4, puis > 1
- nombres divisibles par 5, puis par 2
- mots de taille < 4, puis < 7
- mots contenant plus de deux ‘s’, puis un

exemple

```scala
List(5, 1, 2, 4, 0).filter(i => i > 4) // List(5)
```

puis

```scala
def largerThan(n: Int): Int => Boolean = i => i > n
List(5, 1, 2, 4, 0).filter(largerThan(1))  // List(5, 2, 4)
```

CURRYFICATION

- nous sommes passés de

  highScoringWords : ((*String*→*Int*),*List*[*String*],*Int*) → *List*[*String*]

  à

  highScoringWords2 : (*String*→*Int*) → (*List*[*String*],*Int*) → *List*[*String*]

  et si on voulait aussi factoriser `words` ?

- *highScoringWords3* : (*String*→*Int*) → *List*[*String*] → *Int* → *List*[*String*]

c'est le principe de la **curryfication**

attention : l'ordre des paramètres est important

CURRYFICATION EN SCALA (MÉTHODES)

non currifié _ • (*Int*,*Int*) → *Int*

Scala

```scala
def addition(a: Int, b: Int): Int = a + b
val v1 = addition(2, 3)  // 5
```

currifié _ • *Int* → *Int* → *Int*

Scala

```scala
def additionC(a: Int): Int => Int = b => a + b
val add2 = additionC(2)
val v2 = add2(3)           // 5
val v3 = additionC(2)(3)   // 5
```

currifié _ • *Int* → *Int* → *Int*

Scala

```scala
def additionCv2(a: Int)(b: Int): Int = a + b
val add2v2 = additionCv2(2)
val v4 = add2v2(3)         // 5
val v5 = additionCv2(2)(3) // 5
```

CURRYFICATION EN SCALA (FONCTIONS)

non currifié *(Int,Int) → Int*

Scala

```scala
val addition : (Int, Int) => Int = (a, b) => a + b
val v1 = addition(2, 3)
```

currifié *Int → Int → Int*

Scala

```scala
val additionC : Int => Int => Int = a => b => a + b
val add2 = additionC(2)
val v2 = add2(3)
val v3 = additionC(2)(3)
```

CURRYFICATION EN JAVA (MÉTHODES)

non currifié $c \hookrightarrow (Int,Int) \rightarrow Int$

Java

```java
static int addition(int a, int b) { return a + b; }
System.out.println(addition(2, 3));
```

currifié $c \hookrightarrow Int \rightarrow Int \rightarrow Int$

Java

```java
static Function<Integer, Integer> additionC(int a) { return b -> a + b; }
Function<Integer, Integer> add2 = additionC(2);
System.out.println(add2.apply(3));
System.out.println(additionC(2).apply(3));
```

CURRYFICATION EN JAVA (FONCTIONS)

non currifié *(Int,Int) → Int*

Java

```java
static BiFunction<Integer, Integer, Integer> addition = (a, b) -> a + b;
System.out.println(addition.apply(2, 3));
```

currifié *Int → Int → Int*

Java

```java
static Function<Integer, Function<Integer, Integer>> additionC2 = a -> b -> a + b
Function<Integer, Integer> add2v2 = additionC2.apply(2);
System.out.println(add2v2.apply(3));
System.out.println(additionC2.apply(2).apply(3));
```

(DÉ)CURRYFICATION EN SCALA (AUTOMATISATION)

$$((T,U)\to V) \to (T\to U\to V)$$

Scala

```scala
def curried[T, U, V](f: (T, U) => V): T => U => V = {
    t => u => f(t, u)
}
```

$$(T\to U\to V) \to ((T,U)\to V)$$

Scala

```scala
def uncurried[T, U, V](f: T => U => V): (T, U) => V = {
    (t, u) => f(t)(u)
}
```

(DÉ)CURRYFICATION EN JAVA (AUTOMATISATION)

$$((T,U)\to V) \to (T\to U\to V)$$

Java

```
static <T, U, V> Function<T, Function<U, V>> curried(BiFunction<T, U, V> f) {
    return t -> u -> f.apply(t, u);
}
```

$$(T\to U\to V) \to ((T,U)\to V)$$

Java

```
static <T, U, V> BiFunction<T, U, V> uncurried(Function<T, Function<U, V>> f) {
    return (t, u) -> f.apply(t).apply(u);
}
```

PROBLÈME : WORD SCORE V7

- obtenir le score cumulé de tous les mots

idée

- utiliser `foldLeft`

  `List[A].foldLeft(z: B)(f: (B, A) => B): B`

  List[*A*] ↪ *B* → ((*B*,*A*)→*B*) → *B*

Scala

```scala
def cumulativeScore(scoring: String => Int, words: List[String]): Int = {
    words.foldLeft(0)((acc, word) => acc + scoring(word))
}
```

EN JAVA ?

Java

```java
static int cumulativeScore(Function<String, Integer> scoring, List<String> words)
    int result = 0;
    for (String word: words) {
        result += scoring.apply(word);
    }
    return result;
}
```

(on verra un proche de `foldLeft` en Java plus tard)

EXERCICE

écrire des fonctions qui retournent :

- la somme des entiers d'une liste
- la longueur totale des mots d'une liste
- le nombre total de 's' dans les mots d'une liste
- le maximum des entiers d'une liste

SYNTHÈSE INTERMÉDIAIRE

- `sortBy`: *List*[*A*] ↪ (*A*→*B*) → *List*[*A*]

  ```
  List[A].sortBy[B](f: A => B)(implicit ord:
  Ordering[B]): List[A]
  ```

- `map`: *List*[*A*] ↪ (*A*→*B*) → *List*[*B*]

  ```
  List[A].map(f: A => B): List[B]
  ```

- `filter`: *List*[*A*] ↪ (*A*→*Boolean*) → *List*[*A*]

  ```
  List[A].filter(p: A => Boolean): List[A]
  ```

- `foldLeft`: *List*[*A*] ↪ *B* → ((*B*,*A*)→*B*) → *B*

  ```
  List[A].foldLeft(z: B)(f: (B, A) => B): B
  ```

TYPE PRODUIT

- $T = \{a : T_1, b : T_2, ...\}$
- immuable
- gratuits : constructeur, accesseurs, comparaison

Scala

```scala
case class ProgrammingLanguage(name: String, year: Int)
val java = ProgrammingLanguage("Java", 1995)
val scala = ProgrammingLanguage("Scala", 2004)
```

Java

```java
record ProgrammingLanguage(String name, int year) {}
ProgrammingLanguage java = new ProgrammingLanguage("Java", 1995);
ProgrammingLanguage java = new ProgrammingLanguage("Scala", 2004);
```

notation _ utilisable

la fonction `lang => lang.name` devient `_.name`

EXERCICE

```Scala
case class ProgrammingLanguage(name: String, year: Int)
val java = ProgrammingLanguage("Java", 1995)
val scala = ProgrammingLanguage("Scala", 2004)
val languages = List(java, scala)
```

écrire des fonctions qui retournent :

- la liste des noms de langages
- la listes des langages récents (post. à 2000)

STREAMS JAVA

- Java dispose d'une API de collection particulière : `Stream<T>`
- les streams sont des objets immuables, ils ne peuvent pas être modifiés
- ils supportent aussi une évaluation paresseuse
- 3 types d'opérations :
  - création : `empty`, `of`, `iterate` et `generate`, `toStream`, ...
  - transformation : `filter`, `map`, `flatMap`, ...
  - terminales : `forEach`, `collect`, `reduce`, `allMatch`, `anyMatch`, `count`, `findFirst`, `findAny`, ...
- on a des streams spécialisés
  - opérations associées, ex `DoubleStream::sum`

EVALUATION PARESSEUSE

- exécution au moment du besoin (opération terminale)

Java

```
public static void pasBoum() {
    List<Integer> nombres = List.of(2,4,0,8);
    IntBinaryOperator divise = (x, y) -> x / y;
    nombres.stream()
            .map(n -> divise.applyAsInt(42, n))
            .forEach(System.out::println);
}
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Java

```
public static void pasBoum() {
    List<Integer> nombres = List.of(2,4,0,8);
    IntBinaryOperator divise = (x, y) -> x / y;
    nombres.stream()
            .map(n -> divise.applyAsInt(42, n))
            .limit(2)
            .forEach(System.out::println);
}
```

SCALA VS JAVA : MAP

Scala

```scala
def wordScores(scoring: String => Int, words: List[String]): List[Int] = {
    words.map(scoring)
}
```

Java

```java
static List<Integer> wordScores(Function<String, Integer> scoring, List<String> w
    return words.stream().map(scoring).toList();
}
```

SCALA VS JAVA : FILTER

Scala

```scala
def highScoringWords(scoring: String => Int, words: List[String]): List[String] =
    words.filter(w => scoring(w) > 1)
}
```

Java

```java
static List<String> highScoringWords(Function<String, Integer> scoring, List<Stri
  return words.stream().filter(w -> scoring.apply(w) > 1).toList();
}
```

SCALA VS JAVA : FOLDLEFT

Scala

```scala
def cumulativeScore(scoring: String => Int, words: List[String]): Int = {
    words.foldLeft(0)((acc, word) => acc + scoring(word))
}
```

Java

```java
static int cumulativeScore(Function<String, Integer> scoring, List<String> words)
    return words.stream().map(scoring).reduce(0, (acc, score) -> acc + score);
}
```

noter le `map` en Java

- `foldLeft`: *List*[*A*] ↪ *B* → ((*B*,*A*)→*B*) → *B*

  `List[A].foldLeft(z: B)(f: (B, A) => B): B`
- `reduce`: *Stream*[*A*] ↪ (*A*,(*A*,*A*)→*A*) → *A*

  `A reduce(A identity, BinaryOperator<A> accumulator);`

SCALA VS JAVA : FOLDLEFT

mais on a rarement une solution unique !

Java

```java
static int cumulativeScore(Function<String, Integer> scoring, List<String> words)
    return words.stream().map(scoring).reduce(0, (acc, score) -> acc + score);
}

static int cumulativeScore2(ToIntFunction<String> scoring, List<String> words) {
    return words.stream().mapToInt(scoring).sum();
}

static int cumulativeScore3(Function<String, Integer> scoring, List<String> words
    return words.stream().map(scoring)
                .collect(Collectors.reducing(0, (acc, score) -> acc + score));
}
```

RÉSUMÉ

- fonctions stockées comme des valeurs et lambdas
- fonctions passées en paramètres / retournées
- (dé)curryfication
- application partielle
- types produit
- Scala
  - `sortBy`, `map`, `filter`, `foldLeft`
  - notation _
- Java
  - interfaces fonctionnelles et lien aux lambdas
  - API fonctionnelle
  - API Stream