PF
Programmation Fonctionnelle

Legond-Aubry Fabrice

fabrice.legond-aubry@parisnanterre.fr

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Collections & Tables & Flux

Compléments

Définitions (Hors Langage)

Exemple

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Collections & Tables & Flux

Compléments

Définitions (Hors Langage)

Exemple

Collections & Tables & Flux

Les listes (« Collection ») et les tables (« Map »)

- Souvent abusivement nommées les Collections.
  - ✓ Regroupe les listes et ensembles qui héritent de l'interface « `Collection` »
  - ✓ Regroupe les tables qui héritent de l'interface « `Map <K,V>` »

- Manipulations des listes/ensembles (« `Collection` »)
  - ✓ Toutes ces classes dérivent de « `Collection` »
  - ✓ `Iterable` est la superinterface de `Collection` »
  - ✓ Attention : « `Collection` » est la « Top Level Interface» des listes
  - ✓ Attention : « `Collections` » est une classe d'outils de manipulation des Collections
  - ✓ Voir : « `Array` »/« `Arrays` », …

- Manipulations des tables (« `Map <K,V>` »)
  - ✓ Toutes les classes dérivent de « `Map<K,V>` »
  - ✓ Attention : « `Map <K,V>` » est la « Top Level Interface» des tables

Collections & Tables & Flux

Les listes (« Collection ») et les tables (« Map »)

- Rappel :
  - ✓ Attention à la concurrence et aux différentes implémentations
  - ✓ List : liste ordonnée ou non d'éléments
    - ➢ `ArrayList`, `Stack`, `Vector`
  - ✓ Set : liste ordonnée ou non d'éléments distincts
    - ➢ `SortedSet`, `TreeSet`
  - ✓ Map : Table de hachage d'éléments (Key,Value) ordonnés ou non
    - ➢ `HashMap`, `TreeMap`

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Toutes les sous classes de « Collection » héritent de l'interface « `Iterable` »
  - ✓ Offre « `iterator`() », « `spliterator`() », « `forEach`() »

- Toutes les collections possèdent la méthode « `iterator()` »
  - ✓ Permet de passer en revue un par un les éléments
  - ✓ triés dans une collection triée
  - ✓ Il existe des itérateurs (classes) pour les types primitifs

- Toutes les collections possèdent la méthode « `spliterator()` »
  - ✓ Permet de créer automatiquement une partition et un itérateur pour chacune d'elle
  - ✓ Utile pour les traitements //
  - ✓ On peut implémenter un « `Spliterator` » pour différentes structures.

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Un « `Iterator` » a les méthodes « `hasNext()` » / « `next()` » habituel
  - ✓ « `forEachRemaining`() » utile pour faire un cas particulier avec le premier élément puis traiter tous les autres
  - ✓ « `next`() » et « `forEachRemaining`() » ne détruit pas les éléments
  - ✓ « `remove`() » détruit l'éléments mais aucune garantie en cas de concurrence

- « `ForEach` » et « `ForEachRemaining` » prennent des « `Consumer` »
  - ✓ « `default void forEachRemaining(Consumer<? super E> action)` »
  - ✓ « `andThen` » du « `Consumer` » peut être utilisé

- Les objets passés au « `Consumer` » peuvent être mutés s'ils sont mutables.
  - ➢ Pas du tout esprit P.F.
  - ➢ Impossible pour les Integer, String, Double, Boolean, …
  - ➢ Possible pour StringBuffer, AtomicInteger, …

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Exemple (interface iterable) : forEach, et iterator (forEachRemaining)

```
List<String> ordis =
      Arrays.asList("Dell", "HP", "Lenovo", "IBM");

ordis.forEach ( x -> System.out.println (x) );

Iterator<String> itV = ordis.iterator();
Consumer<String> cS = x -> System.out.println (x);
itV.forEachRemaining( cS.andThen (
      x -> System.err.println("Debug:"+x)
) ;
```

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Exemple (interface Iterable) : spliterator (forEachRemaining)

```
List<String> ordis =
      Arrays.asList("Dell", "HP", "Lenovo", "IBM");

// Coupe une source en 2 listes
Spliterator<String> ordisSplit = ordis.spliterator();
// 1ere partie de la partition
Spliterator<String> partition1erePartie =
      ordisSplit.trySplit();
partition1erePartie.forEachRemaining( cS ) ;
// 2eme partie de la partition
ordiSplit.forEachRemaining ( cS ) ;
```

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Exemple non valide: Calcul d'un max, fonction impure
  - ✓ **int est un type primitif NON final (il change de valeur) !!!**

```
List<String> ordis =
      Arrays.asList("Dell", "HP", "Lenovo", "IBM");
Iterator<String> itV = ordis.iterator();
int maxLen= 0;
  if (itV.hasNext())
      maxLen = itV.next().length();
  Consumer<String> cS = x -> {
      int lg = x.length();
      if (lg > maxLen) { maxLen=lg; }
  };
  itV.forEachRemaining( cS ) ;
  System.out.println (maxLen);
```

Collections & Tables & Flux

« Collection » (listes/ens) : itérateurs

- Exemple valide: Calcul d'un max, fonction impure
  - ✓ **AtomicInteger est un objet mutable mais dont la ref est final !!!**

```
List<String> ordis =
      Arrays.asList("Dell", "HP", "Lenovo", "IBM");
Iterator<String> itV = ordis.iterator();
AtomicInteger maxLen= new AtomicInteger();
  if (itV.hasNext())
      maxLen.set(itV.next().length());
  Consumer<String> cS = x -> {
      int lg = x.length();
      if (lg> maxLen.get()) { maxLen.set(lg); }
  };
  itV.forEachRemaining( cS ) ;
  System.out.println (maxLen);
```

Collections & Tables & Flux

« Map » (tables) : itérateurs

- Itérateur sur une table (K,V)
  - ✓ Même logique qu'avec une liste. Mais n'hérite pas de « `Iterable` » !!!
- Juste la méthode « `forEach` ».
  - ✓ Utilise « `BiConsumer` », 2 paramètres : un pour K, un pour V

```
default void forEach(BiConsumer<? super K,? super V> action)
```

  - ✓ Possibilité d'utiliser « `andThen` » de « `BiConsumer` »

- Sinon, on peut extraire les collections de K et de V.
  - ✓ Chacune sous forme de collections

Collections & Tables & Flux

« Map » (tables) : itérateurs

- Exemple forEach() sur une table

```
HashMap <String, Double> salaireEmployes = new HashMap();
salaireEmployes.put ("Simon Patarin", 30884.4));
salaireEmployes.put ("Paul Ferriere", 90884.4));

salaireEmployes.forEach (
    (k,v) ->
       System.out.println ("Le salarié "+k+" a pour salaire "+v+"€.")
);
```

Collections & Tables & Flux

« Collection » (listes/ens) : méthodes de modification

- Supprimer des éléments (sous condition) d'une « `Collection` » (liste/ensemble)
  - ✓ Méthode « `removeIf` ». Utilisation d'un « `Predicat` ».
- Trier les éléments d'une « `Collection` » (liste/ensemble)
  - ✓ Méthode « `Sort` »
  - ✓ Utilisation d'un « `Comparator` »
  - ✓ https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
  - ✓ Interface Fonctionnelle permettant de comparer deux éléments
- Remplacer (ou muter) des objets d'une « `Collection` » (liste)
  - ✓ Méthode « `replaceAll` »
  - ✓ Utilisation d'un « `UnaryOperator` » pour une liste

Collections & Tables & Flux

« Collection » (listes/ens) : méthodes de modification

- Trier une liste
  - ✓ Méthode « `Sort` »
  - ✓ Utilisation d'un « `Comparator` »
  - ✓ https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
  - ✓ Interface Fonctionnelle permettant de comparer deux éléments
- Le « `Comparator` » a de nombreuses méthodes
  - ✓ Permet de configurer précisément le tri
  - ✓ Static : « `naturalOrder` » et « `reverseOrder` » renvoie deux comparateurs par défault
  - ✓ « `reversed` » renvoie un « `Comparator` » inverse du comparateur actuel
  - ✓ « `thenComparing` » permet d'ajouter un nouveau critère en cas d'égalité du premier critère

Collections & Tables & Flux

« Map » (tables) : méthodes de modification

- Remplacer (ou muter) des objets d'une « `Map` » (table)
  - ✓ Méthode « `replaceAll` »
  - ✓ Utilisation d'une « `BiFunction <? super K,? super V,? extends V>` » pour une table (K,V)
  - ✓ Rappel : BiFunction < type param1 , type param2 , type retour >

Collections & Tables & Flux

« Map » (tables) : méthodes de modification

```
default V compute
(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
```

- Faire des calculs sur des tables: Méthode « `compute` »
  - ✓ Travaille sur un élément de la table référencé par le paramètre K
  - ✓ « `compute` » : génère une valeur pour une clef avec une « `BiFunction` »
  - ✓ Si la « `BiFunction` » renvoie null cela supprimera l'entrée dans la table
  - ✓ MAJ atomique de l'élément suivant la « `BiFunction` »
  - ✓ Si la méthode « `remappingFunction` » génère une exception, l'élément est non touché et l'exception est transmise.
  - ✓ La méthode « `remappingFunction` » ne doit pas modifier la table (K).
    - ➢ « `ConcurrentModificationException` » est généré en cas de tentative
  - ✓ La méthode « `remappingFunction` » peut modifier l'état de la valeur.

Collections & Tables & Flux

« Map » (tables) : méthodes de modification

```
default V computeIfAbsent
  (K key, Function<? super K,? extends V> mappingFunction)
```

- Faire des calculs sur des tables: Méthode « `computeIfAbsent` »
  - ✓ Travaille sur un élément de la table référencé par le paramètre K
  - ✓ Si la clef n'existe pas ou renvoie "`null`", tente de calculer une valeur et d'en insérer la valeur dans la table.
    - ➢ Si la « `mappingFunction` » renvoie `null`, il n'y a aucun ajout.
  - ✓ Si la méthode « `mappingFunction` » génère une exception, l'élément est non touché et l'exception est transmise.
  - ✓ La méthode « `mappingFunction` » ne doit pas modifier la table.
    - ➢ « `ConcurrentModificationException` » est généré en cas de tentative

Collections & Tables & Flux

« Map » (tables) : méthodes de modification

```
default V computeIfPresent
  (K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
```

- Faire des calculs sur des tables: Méthode « `computeIfPresent` »
  - ✓ Travaille sur un élément de la table référencé par le paramètre K
  - ✓ Si la clef renvoie une valeur non null, tente de calculer une nouvelle valeur.
    - ➢ Si la « `remappingFunction` » renvoie null, cela supprimera l'entrée dans la table
  - ✓ Si la méthode « `remappingFunction` » génère une exception, l'élément est non touché et l'exception est transmise.
  - ✓ La méthode « `remappingFunction` » ne doit pas modifier la table.
    - ➢ « `ConcurrentModificationException` » est généré en cas de tentative

Collections & Tables & Flux

« Map » (tables) : méthodes de modification

```
default V merge
   (K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
```

- Faire des calculs sur des tables : Méthode « `merge` »
  - ✓ Travaille sur un élément de la table référencé par le paramètre K
  - ✓ Si la clef est non présente, ajoute simplement la nouvelle entrée
  - ✓ Si la clef est présente, on utilise la « `BiFunction` » pour mettre à jour la valeur à partir de la nouvelle valeur et de l'ancienne
  - ✓ Si la clef est null, merge sur la clef « 0 »

Collections & Tables & Flux
« Stream » (les flux)

- Collections et tables : Pour permettre de traiter efficacement les données, elles sont souvent transformés/organisées en flux de données
  - ✓ Ce n'est cependant pas limité aux collections et/ou au tables
- Caractéristiques d'un flux :
  - ✓ Un flux ne stocke pas de données.
  - ✓ Il se contente de les transférer d'une source vers une suite d'opérations.
  - ✓ Le chargement des données pour des opérations sur un flux s'effectue de façon « lazy » (chargement à l'utilisation).
  - ✓ Un flux peut ne pas être borné, contrairement aux collections.
  - ✓ Un flux n'est pas réutilisable. **Il est utilisable 1 et 1 seul fois !**

Collections & Tables & Flux
« Stream » (les flux)

- Caractéristiques d'un flux (suite) :
  - ✓ Un flux ne modifie pas les données de la source sur laquelle il est construit.
    - **Attention, il reste possible de le faire par des moyens détournés.**
    - **ex: mutation de l'état des instances d'objets**
    - **Dangereux. → Effets de bords (concurrence). Vision Non P.F. !**
  - ✓ Si une opération doit « modifier » des données pour les réutiliser :
    - Elle va construire un nouveau flux (sortant) à partir du flux initial (entrant).
    - Ce sont des opération dites intermédiaires.
    - Cela change potentiellement le type des éléments entre l'entrée et la sortie de l'opération.
    - Vision P.F.

Collections & Tables & Flux
« Stream » (les flux)

- Méthodes de manipulations de flux :
  - ✓ Il existe des méthodes **INTERMEDIAIRES** qui renvoient des flux et peuvent donc être composées (ici au sens applications successives de transformation du flux).
    - ➢ Changement potentielle de type de chaque élément (ENTRE entrée et sortie)
  - ✓ Il existe des méthodes **TERMINALES** qui ne produisent pas de nouveau flux (le retour n'est pas un flux) et arrête la composition.

Collections & Tables & Flux
« Stream » (les flux) : générateurs

- La classe « `Stream` » permet de manipuler les flux
  - ✓ On trouve dans beaucoup de classe une méthode permettant de générer des « `Stream` »
- Par ex. pour générer un flux à partir des collections :
  - ✓ Méthode « `stream`() » et « `parallelStream`() » sur les listes ou via Arrays
- Il existe une méthode
  - ✓ « `of()` » qui prend une suite d'éléments et qui en génère un flux.
  - ✓ « `empty()` » qui génère une flux vide
- Il existe une méthode « `concat`() » dans la classe « `Stream` » qui agrège des flux

Collections & Tables & Flux
« Stream » (les flux) : générateurs

- Il existe une méthode « `generate`() » dans la classe « `Stream` » pour créer un flux à partir d'un code
  - ✓ Utilisation d'une I.F. « `Supplier` »
- Il existe une méthode « `iterate`() » dans la classe « `Stream` » pour créer un flux à partir d'une fonction
  - ✓ Utilisation d'une I.F. « `UnitaryOperator` »
- Il existe une classe interne « `Stream.Builder` » qui permet de fabriquer des flux complexes.

Collections & Tables & Flux

« Stream » (les flux) : analyse & test

- Les flux ont des opérations de tests sur la totalité du flux
  - ✓ « `anyMatch` » renvoie VRAI si au moins un des éléments du flux vérifie le prédicat passé en paramètre
    ```
    boolean anyMatch(Predicate<? super T> predicate)
    ```
  - ✓ « `allMatch` » renvoie VRAI si tous les éléments du flux vérifie le prédicat passé en paramètre
    ```
    boolean allMatch(Predicate<? super T> predicate)
    ```
  - ✓ « `noneMatch` » renvoie VRAI si aucun des éléments du flux vérifie le prédicat passé en paramètre
    ```
    boolean noneMatch(Predicate<? super T> predicate)
    ```
- Ce sont des opérations terminales.
- Rappel : On peut aussi manipuler les prédicats (pour les inverser par ex)

Collections & Tables & Flux

« Stream » (les flux) : parcours

- Les flux peuvent être parcourus comme les collections
  - ✓ Méthode « `forEach`() » avec une « `Function` ». C'est une opération TERMINALE.
- On peut aisément exclure certains éléments du flux avec un prédicat
  - ✓ Méthode « `filter`() » avec un « `Predicate` ». C'est une opération intermédiaire.
- On peut trier les éléments comme les collections.
  - ✓ Attention perte de lazyness. On doit consommer tout le flux entrant pour produire le sortant.
  - ✓ Méthode « `sort`() » avec un « `Comparator` ». C'est une opération intermédiaire.
- On peut dupliquer le flux et appliquer sur l'une des branches un consommateur. Utile pour faire du debug lors de la composition de flux.
  - ✓ Méthode « `peek`() » avec un « `Consumer` ». C'est une opération intermédiaire.
- On peut appliquer de nombreuses fonctions de calculs bases
  - ✓ « `count` », « `max` », « `min` ». Ce sont des opérations TERMINALES.
  - ✓ « `distinct` ». C'est une opération intermédiaire.

Collections & Tables & Flux

« Stream » (les flux) : Manipulations – map() VS flatmap()

- Les fonctions de « `map`() » et de « `flatmap`() »
  - ✓ Elles appliquent une transformation sur l'ensemble des éléments du flux pour générer en sortie un flux d'un nouveau type.
  - ✓ Ce sont des opérations intermédiaires.

- Différence
  - ✓ « `map`() » travaille sur un flux d'éléments
  - ✓ « `flatmap`() » travaille avec des flux de flux d'éléments pour produire un flux

- Un peu semblable à la méthode « `replaceAll`() » des collections et des tables mais :
  - ➢ Elles ne font pas des mutations de la liste
  - ➢ Elles travaille sur un flux
  - ➢ Elles prennent un élément d'entrée et produisent une sortie d'un nouveau type.
  - ➢ Vision plus P.F. que « `replaceAll`() »

Collections & Tables & Flux

« Stream » (les flux) : Manipulations – map() VS flatmap()

Map	FlatMap
Travail sur un flux de valeurs (1 niveau)	Travail sur un flux DE FLUX de valeurs (2 niveaux)
Elle applique une fonction de mapping (transformation sur chaque élément du flux)	Elle applique une fonction de mapping (transformation sur chaque élément du flux). Puis une fonction d'aplanissement (« flatten ») (agrégation des flux de 2e niveau en un seul flux de sortie)
La fonction « mapper » produit un élément de sortie pour chaque élément d'entrée	La fonction « mapper » peut produire plusieurs éléments (encapsulé dans un flux)pour chaque éléments d'entrée
Conversion « One-To-One »	Conversion « One-to-Many »
Mapper : Function <T,R> Stream<T> →« mapper »→ Stream<R>	Mapper : Function < T, Stream<R> > (note : T peut être une liste de liste par ex) Stream<T> →« mapper »→ Stream<Stream<R>> →« flat »→ Stream<R>

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de map()

```
<R> Stream<R> map (Function<? super T,? extends R> mapper)
```

- La fonction « `map`() » prend une « `Function` » nommée « `mapper` »
  - ✓ Elle s'applique sur un flux qui produit des éléments de type T
  - ✓ Elle applique une fonction « `mapper` » qui produit un élément de type R à partir d'un élément de T
  - ✓ Elle produit donc un flux sortant d'éléments de type R qui est le résultat de l'application de la fonction « `mapper` » sur chacun des éléments de type T du flux entrant.

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de map()

- Exemple de map() avec un typage explicite pour illustration :

```
Stream<Integer> sOfIntegers = Stream.of (23,45,65,76,-1,34);

sOfIntegers.map(
      (Integer x) -> { return (String) "str="+x.toString(); }
).forEach(
      (String s) -> System.out.println (s.getClass()+":"+s)
);
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de flatmap()

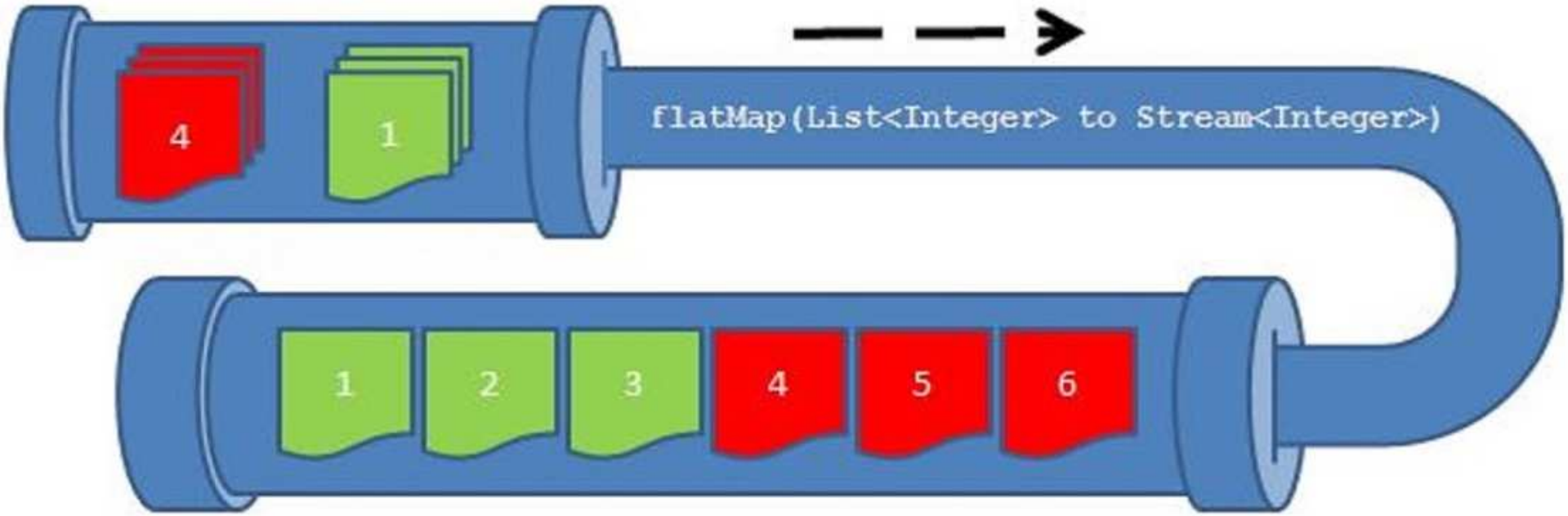
```
<R> Stream<R>  flatMap
    (Function<? super T,? extends Stream<? extends R>> mapper)
```

- La fonction « `flatmap`() » prend une « `Function` » nommée « `mapper` »
  - ✓ Elle prend un flux entrant qui produit un type T
  - ✓ Elle utilise une fonction « `mapper` » qui produit un **flux** de type R à partir de T
  - ✓ Elle produit donc un flux de flux d'éléments de type R qui sont le résultat de l'application de la fonction « `mapper` » sur chacun des éléments du flux original. Opération de « **mapping** »
  - ✓ Ensuite l'ensemble des flux de R produits sont agrégés en un seul flux concaténé (flux mis bout à bout). Opération de « **flatten** »
- Souvent `flatmap`() est utilisée pour l'aplanissement sans véritable opération de `map`().

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de flatmap()

- Utilisation simple du flatmap() → on fait juste un aplanissement



Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de flatmap()

- Utilisation simple du flatmap() → on fait juste un aplanissement (flatten)
- Exemple de flatmap() :

```
List<List<Integer>> lOfLOfInts =
    Arrays.asList(
        Array.asList(2,4,6),
        Array.asList(1,3,5),
        Array.asList(10,20,30));
lOfLOfInts.flatMap
      (lst -> lst.stream())
      .forEach
          (System.out::println);
```

```
Stream sOfS =
      Stream.of (
          Stream.of (2,4,6),
          Stream.of(1,3,5),
          Stream.of(10,20,30));
sOfs.flatMap
      ( x -> x) // F° IDENTITE
      .forEach
          (System.out::println);
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de flatmap()

- flatmap() par forcément pour flatten() → que fait ce code ?

```
List<Integer> lOfInts = Arrays.asList( 1, 4, 2, 6, 8 );
final Random random = new Random();
lOfInts.stream()
      .flatMap ( (Integer v) -> {
            ArrayList<String> alS = new ArrayList<>();
            char rc=(char) ('a'+random.nextInt(26) );
            String rs = new String(new char[] { rc } );
            for (int j=0; j<v; j++) alS.add(rs);
            return (Stream<String>) alS.stream();
      })
      .forEach ( x -> System.out.println (x) );
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de reduce()

- Opération de « `reduce` »
  - ✓ Réduire un flux à une seule valeur résultat du même type que les éléments du flux
  - ✓ Combinaison d'éléments (par ex: min, max, somme, moyenne pour des int)
- On applique un « `BinaryOperator` » sur chacun des éléments.
  - ✓ Une version modifié du « `reduce()` » utilise un élément initiale pour initialiser l'opération. Par ex, pour une somme, la valeur initiale est 0.
    `T reduce (T identity, BinaryOperator<T> accumulator)`
- C'est une opération TERMINALE. Elle ne produit pas un nouveau flux !
- L'opération de « `reduce`() » produit un « `Optional` » qui sera discuté plus tard dans ce module.

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de reduce()

- Il existe une version parallélisable et plus générique

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

- Elle nécessite la définition d'une opération de fusion des partitions nommé « `BinaryOperator<U> combiner` »
- Elle est proche du « fold » de P.F.
- Cette opération de « `reduce`() » produit type U différent du type d'entrée T
  - ➢ U peut donc être un type complexe comme une liste d'éléments
- C'est aussi une opération TERMINALE. Elle ne produit pas un nouveau flux !

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de reduce()

- Exemple de reduce() :

```
List<String> mots = Arrays.asList
  ("TNT",  "Geek", "bonjour", "Quiz", "moralement");
String motLePlusLong = mots.stream().reduce(
  (String mot1, String mot2) ->
  mot1.length() > mot2.length() ? mot1 : mot2
);

List<Integer> valeurs = Arrays.asList (1,2,3,4,5,6);
Integer somme = valeurs.stream().reduce (0, (a, b) -> a+b);
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- « `collect`() » est une opération complexe de réduction du flux vers un ensemble dit de « réduction mutable » (mutable reduction)
  - ✓ C'est donc une opération qui **accumule** les éléments d'entrée dans un **conteneur** (« `container` ») mutable, qui peut évoluer au fur et à mesure que le flux est traité.
  - ✓ Le type de la réduction mutable peut être différent de celui des éléments d'entrée
- Réduction mutable (« `Mutable reduction` ») :
  - ✓ C'est un conteneur dont l'état peut évoluer au cours du temps
  - ✓ exemple de conteneur : un StringBluider ou une Collection
  - ✓ https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#MutableReduction

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

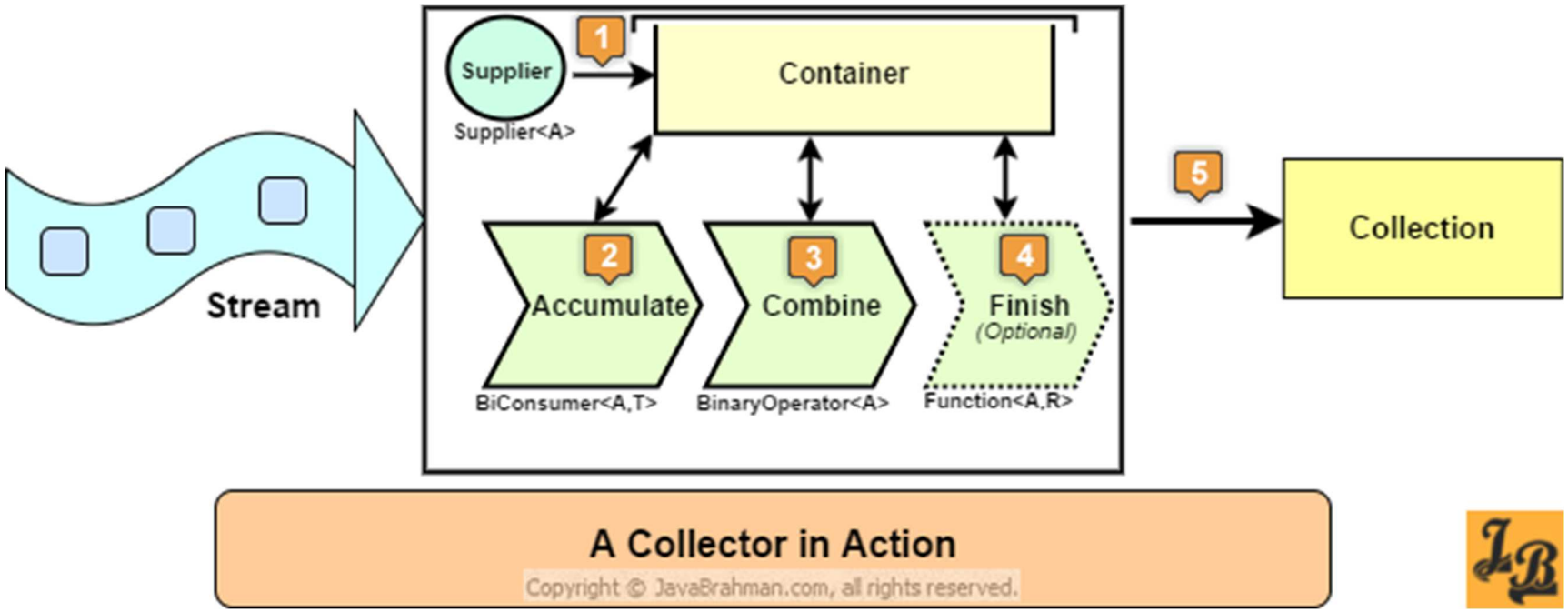
- Propriétés:
  - ✓ « `collect`() » est une opération plus générique que « `reduce`() »
  - ✓ « `collect`() » utilise soit un « `Collector` » soit un tuple de « `Function` » représentant un « `Collector` » (en version simplifiée)
  - ✓ C'est une opération TERMINALE. Elle ne produit pas un nouveau flux !
  - ✓ Rien n'interdit l'utilisation d'un « `map`() » avant le « `collect`() ».

```
// collect() avec un Collector à définir
<R,A> R collect (Collector<? super T,A,R> collector)

// collect()  avec un tuple Collector sans "finisher"
<R> R collect
(Supplier<R> supplier, BiConsumer<R,? super T> accumulator,
                                    BiConsumer<R,R> combiner)
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()



Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- Un « `Collector<T,A,R>` » contient les 4 opérations suivantes :
  - ✓ Un '**supplier**' de type « Supplier » qui sera utiliser pour construire un nouveau conteneur de type « A » (pour accumulator)
    `Supplier<A>`
  - ✓ Un '**accumulator**' qui va ajouter des éléments « T » (provenant du flux) au conteneur de type « A »
    `BiConsumer<A,T>`
  - ✓ Un '**combiner**' qui va combiner deux conteneurs « A » en un seul « A ». Utiliser uniquement lorsqu'on parallélise pour fusionner les résultats de chaque partition
    `BinaryOperator<A>`
  - ✓ Un '**finisher**' (optionnel) qui va appliquer une opération finale (à la fin du traitement du flux) sur l'accumulateur de type « A » (voir sur chacun des éléments du conteneur) pour renvoyer un résultat de type « R »
    `Function<A,R>`

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- Collect() donne donc les algorithmes suivant (séquentielle ou //) pour deux éléments t1 et t2 :

```
// pas de parallèlisation →
// 1 seul conteneur
A a1 = supplier.get();
accumulator.accept(a1, t1);
accumulator.accept(a1, t2);


// on applique directement le finisher
R r1 = finisher.apply(a1);
```

```
// 2 partitions →
// 2 conteneurs / accumulateurs
A a2 = supplier.get();
accumulator.accept(a2, t1);
A a3 = supplier.get();
accumulator.accept(a3, t2);


// on applique combiner + finisher
R r2 = finisher.apply(
        combiner.apply(a2, a3));
```

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- « `Collector` » est une interface qui possède des méthodes usines statiques
  - ✓ `static <T,A,R> Collector<T,A,R> of (Supplier<A> supplier, BiConsumer<A,T> accumulator, BinaryOperator<A> combiner, Function<A,R> finisher, Collector.Characteristics... characteristics)`
  - ✓ `static <T,R> Collector<T,R,R> of (Supplier<R> supplier, BiConsumer<R,T> accumulator, BinaryOperator<R> combiner, Collector.Characteristics ... characteristics)`
- Il existe des « `Collector` » prédéfinis
- « `Collectors` » contient des « `Collector` » prédéfinis
  - ✓ On peut faire un « `reduce`() » en utilisant la méthode « `collect`() » et « `Collector` » produit par un « `Collectors.reducing`() »

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- Les type de « Collector » prédéfinis les plus utilisés sont
  - ✓ « `Collectors.groupingBy`() » qui permet d'instancier un « `Collector` » groupBy comme en SQL sur un flux
  - ✓ « `Collectors.summarizingInt`() » qui permet d'instancier un « `Collector` » qui va appliquer une fonction renvoyant un entier résultant de l'application d'une fonction sur chacun des éléments du flux (map)
  - ✓ « `Collectors.sumingInt`() » qui permet d'instancier un « `Collector` » qui va somme les entiers du flux
  - ✓ « `Collectors.partitioningBy`() » qui permet d'instancier un « `Collector` » pour partionner les éléments en fonction d'un prédicat (donc 2 partitions)

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/

Reduction Operation(s)	Method(s)	Purpose
averaging	averagingDouble(), averagingLong(), averagingInt()	To average elements of type Double/Long/Integer after applying a mapping function to the elements to extract respective values to be averaged
counting	counting()	Count the number of stream elements
grouping	groupingBy()	To produce Map of elements grouped by grouping criteria provided
String concatenation	joining()	For concatenation of stream elements into a single String
mapping	mapping()	Applying a mapping operation to all stream elements being collected

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/

Reduction Operation(s)	Method(s)	Purpose
minimum and maximum determination	minBy() maxBy()	To find minimum/maximum of all stream elements based on Comparator provided
partitioning	partitioningBy()	To partition stream elements into a Map based on the Predicate provided
reduction	reducing()	Reducing elements of stream based on BinaryOperator function provided
summarization	summarizingDouble(), summarizingLong(), summarizingInt()	To summarize stream elements after mapping them to Double/Long/Integer value using specific type Function
summation	summingDouble(), summingLong(), summingInt()	To sum-up stream elements after mapping them to Double/Long/Integer value using specific type Function

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- https://www.javabrahman.com/java-8/java-8-java-util-stream-collector-basics-tutorial-with-examples/

Reduction Operation(s)	Method(s)	Purpose
collect into a Collection	toCollection()	To collect stream elements into a collection
collect into a Map/ConcurrentMap	toMap() toConcurrentMap()	To collect stream elements into a map/concurrent map after applying provided key/value determination Function instances to the elements
collect in a List	toList()	Collects stream elements in a List
collect in a Set	toSet()	Collects stream elements in a Set
collect and transform	collectingAndThen()	Collects stream elements and then transforms them using a Function

Collections & Tables & Flux

« Stream » (les flux) : Manipulations - opération de collect()

- Exemple de collect():

```
List<String> elements = Arrays.asList (
      "elem1", "elem2", "elem2", "elem3", "elem3", "elem4"
      "elem4", "elem3", "elem4", "elem2", "elem5");
Set<String> ensemble = elements.stream().collect(
      () -> new HashSet<String>(), // supplier
      (s, e) -> s.add(e), // accumulator
      (s1, s2) -> s1.addAll(s2) // combiner (seulement pour //)
      );
System.out.println(ensemble);
```