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

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Interfaces Fonctionnelles

Lambda Calculs

Fonctions

Collections & Tables & Flux

Compléments

Compléments

Optional. Qu'est-ce que c'est ?

"I call it my **billion-dollar mistake**. It was the invention of the **null reference in 1965**. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that **all use of references should be absolutely safe**, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has **led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years**."

Tony Hoare

Compléments

Optional. Qu'est-ce que c'est ?

- En P.F., le « null » c'est l'antéchrist (avec les exceptions)
  - ✓ Qui n'a jamais provoqué un NPE.
  - ✓ Indique clairement un bug, un effet de bord ou un problème NON TRAITE !
  - ✓ On perd la pureté.
  - ✓ Problème de typage statique qui se concrétise (réifie) à l'exécution
- « null » n'est pas considéré comme une valeur
  - ✓ Mais plutôt comme une **ABSENCE** de valeur
  - ✓ Mais il est aussi utilisé comme une « **valeur à la signification spéciale** »
  - ✓ Dépend des programmeurs. Comment l'utiliseriez-vous ?
- « null » implique l'obligation de tester la nullité avant utilisation d'un élément
  - ✓ Très couteux en terme de code
  - ✓ complexité (imbrications, multiplicités des tests, gestion des retours, …)

Compléments

Optional. Qu'est-ce que c'est ?

- Exemple:

```
static Function<List<Integer>, Double> moyenne = lstOfInt ->
    { return lstOfInt.stream().reduce(0, Integer::sum) / lstOfInt.size(); }
```

- Que retourner si la liste est vide ?
  - ✓ Retourner null ? Lever une exception ? Retourner autre chose ?
- Dans certains langages, on utilise l'opérateur « `?.` » qui indique l'invocation est faite uniquement si l'élément est « non nul »
- En java, utilisation d'annotations (Java5)
  - ✓ Non standard (javax.annotation, javax.validation.constraints) « `@Nullable` » ou « `@NotNull` »
  - ✓ `@Retention(RUNTIME)` → donc gestion pendant l'exécution
  - ✓ Difficulté de vérification à la compilation, peu de cohérence

Compléments

Optional. Qu'est-ce que c'est ?

- En java 8, on a introduit la classe générique « Optional »
  - ✓ Classe d'encapsulation (Wrapper) générique.
  - ✓ Indique **INTENTIONNELLEMENT** une possibilité de « non valeur »
  - ✓ Permet l'unboxing/boxing d'instances d'objets et de la « non valeur »
  - ✓ Peut aussi être vu comme une collection contenant au plus une valeur

- **La logique** : Si le résultat d'un calcul peut ne pas exister, au lieu d'utiliser `null`, on utilise `Optional` qui oblige l'utilisateur à gérer le cas où il n'y a pas de valeur.

- Peut être utilisé en combinaison des annotations java 5

Compléments

Optional. Qu'est-ce que c'est ?

- Il existe des Optional pour les types primitif (OptionalInt, OptionalLong, OptionalDouble)

- Exemple: Trouver la première valeur d'un Stream<E>
  - ✓ Stream .`findFirst`() renvoie Optional<E>

- Exemple Calculer le maximum d'un IntStream :
  - ✓ intStream .`max`() renvoie un OptionalInt

Compléments

Optional. Manipulations

- Optional :
  - ✓ Classe non mutable, une fois créé, l'instance ne peut être modifiée.

- Création:
  - ✓ Il n'y a pas de constructeur. On utilise des méthodes «usine»
  - ✓ `of(T t)` → création d'un Optional à partir d'une instance t de type T **non nulle sinon génère une exception**
  - ✓ `ofNullable(T t)` → création d'un Optional à partir d'une instance t de type T **nulle ou non**
  - ✓ `empty ()` → création d'un Optional « **sans valeur** » de type T (attention au générique)

Compléments

Optional. Manipulations.

- Une fois crée, une instance d'Optional<T> peut être manipulée et testée
- `boolean isPresent()`
  - ✓ renvoie Vrai si l'instance d'Optional est non nul, faux sinon.
  - ✓ En java 11, il existe aussi `isEmpty()` qui est l'inverse de `isPresent`()
- `void ifPresent(Consumer<? super T> c)`
  - ✓ invoque un consumer c de T si l'optional n'est pas nul
  - ✓ Un `Consumer` return `void`, donc les `ifPresent`() ne peuvent donc être enchaînés
- `T get()`
  - ✓ renvoie la valeur encapsulée ou génère une exception si nul
- `T orElse(T other)`
  - ✓ renvoie le valeur encapsulée est non nul ou other sinon
- `T orElseGet(Supplier<? extends T> s)`
  - ✓ idem `orElse()` mais utilise un supplier de T au lieu d'une instance
  - ✓ En Java 11, il existe aussi `or(Supplier<? extends T> s)` qui renvoie un Optional
- `Optional<T> filter(Predicate<? super T> predicate)`
  - ✓ Permet d'appliquer le filtre des Stream (compatibilité)

Compléments

Optional. map() et flatmap().

- `Optional<U> map ( Function<? super T,? extends U> mapper )`
  - ✓ joue le rôle du « ?. » présenté précédemment
  - ✓ Applique une fonction mapper sur l'élément uniquement si l'Optional contient une valeur
  - ✓ Contrairement à `ifPresent`(), `map`() peut être enchaîné car il renvoie un Optional de U
- Par Exemple, si on a :
  - ✓ instance de type C1 qui a une méthode m1 qui renvoie une instance de C2 ou null
  - ✓ La classe C2 qui a une méthode m2 qui renvoie une instance de classe C3 ou null
  - ✓ La classe C3 qui a une méthode m3 qui envoie une instance de classe C4 ou null
- Si on veut faire : **instance.m1().m2().m3()**
  - ✓ En java classique si instance est null ou m1() ou m2() renvoient null → NPE
  - ✓ Avec un Optional, on peut faire une **chaîne** :

```
Optional.ofNullable(instance)
        .map(C1::m1).map(C2::m2).map(C3::m3)
```

Compléments

Optional. Manipulations.

- Ainsi, au lieu de

```
Int result = ...
if (result != null) {
   System.out.println("result is " + result);
} else {
   System.out.println("result not found");
}
```

- Ou de

```
OptionalInt result = ...
if (result.isPresent()) {
   System.out.println("result is " + result.get());
} else {
   System.out.println("result not found");
}
```

- On peut écrire

```
OptionalInt result = …
System.out.println(result
   .map(value - > "result is " + value)
   .orElse("result not found"));
```

Compléments

Optional. Manipulations.

- Revoir les méthodes de streams qui peuvent générer de Optional
  - ✓ `findAny`(), `findFirst`(), `max`(), `min`(), `reduce`()

- Exemple :

```
Optional<FileInputStream> fis =
  names.stream()
  .filter(name -> !isProcessedYet(name))
  .findFirst()
  .map(name -> new FileInputStream(name));
```

Compléments

Optional. Manipulations

- **Attention** : on ne peut avoir un `Optional<Optional <U>>`
- **Problème** : Le méthode map d'Optional renvoie toujours un Optional<U>
  - ✓ Rappel : `Optional<U> map ( Function<? super T,? extends U> mapper )`
  - ✓ Si la fonction de `map` renvoie un type U qui est déjà un `Optional` → pb
- Il peut arriver que la `Function` de mapper utilisée produise un Optional d'un type
  - ✓ On ne peut donc utiliser map qui tentera l'encapsulation d'un type déjà Optional.
  - ✓ Dans ce cas, il faut « flatten »
- Pour résoudre ce problème, on utilise la méthode `flatmap()`

```
public <U> Optional<U> flatMap
                (Function<? super T, Optional<U>> mapper)
```

Coompléments

Optional. map() et flatmap()

- Flatmap :
  - ✓ La méthode `flatmap()` applique une fonction qui prends un type T en entrée et produit un Optional qui contient une valeur et retourne cet `Optional<U>` sans l'encapsuler à nouveau. Sinon renvoie un Optional « sans valeur ».
  - ✓ flatmap() "flatten" pour ne pas produire un `Optional<Optional<U>>`

- Une version simplifiée :
  on utilise `flatmap`() à la place de `map`() lorsque le mapper (la `Function` de map) renvoie un `Optional`.
- Une autre version :
  on utilise `flatmap`() à la place de `map`() lorsqu'on veut enchaîner des `Function` de lambda renvoyant des `Optional` au lieu d'un autre type.

Compléments

Optional. Manipulations.

- Optional a été créé pour faciliter les **traitements**. Pas pour le stockage.
- Peut s'appliquer sur un flux (streams) normalement
  - ✓ Si vos codes génèrent/consomment des instances d'Optional
- MAIS A EVITER dans les collections
  - ✓ Autant éviter d'ajouter des éléments potentiellement vides à une collection
- Pour les variables membres : éviter de stocker/utiliser des Optional en interne

- Au lieu de

```
public class Foo {
  private final Optional<Bar> bar;

  public Foo(Optional<Bar> bar)
    { this.bar = bar; }

  public Optional<Bar> getBar()
    { return bar; }
}
```

- on écrit

```
public class Foo {
  // peut etre null
  private final Bar bar;

  public Foo(Optional<Bar> bar)
    { this.bar = bar.orElse(null); }

  public Optional<Bar> getBar() {
    return Optional.ofNullable(bar);
  }
}
```

Compléments

Laziness (Une des règles de base de l'informaticien)

- En Java, les codes est souvent exécutés « eagerly » (évaluation avide)
  - ✓ Code facile mais code bloquant (invocations synchrones)
  - ✓ Oblige à la disponibilité de l'ensemble des données avant traitement
  - ✓ Problème : les données peuvent être énormes

- D'où l'utilisation de la Laziness (évaluation paresseuse)
  - ✓ Attendre le dernier moment pour faire l'évaluation d'un code
  - ✓ Attendre le dernier moment pour générer, charger la donnée utilisée
  - ✓ Ne charger / générer que la donnée nécessaire

Compléments

Laziness (Une des règles de base de l'informaticien)

- Un exemple de lazyness, la création/instanciation retardée
  - ✓ Exemple: Patron du singleton (instanciation à la première utilisation)
- Un exemple de lazyness en programmation : les opérateurs logiques
  - ✓ Les expressions booléennes A et B peuvent mettre beaucoup de temps à évaluer
  - ✓ A && B, B ne sera évalué que si A est vrai
  - ✓ A || B, B ne sera évalué que si A est faux
- On peut appliquer la même logique aux flux (Streams)

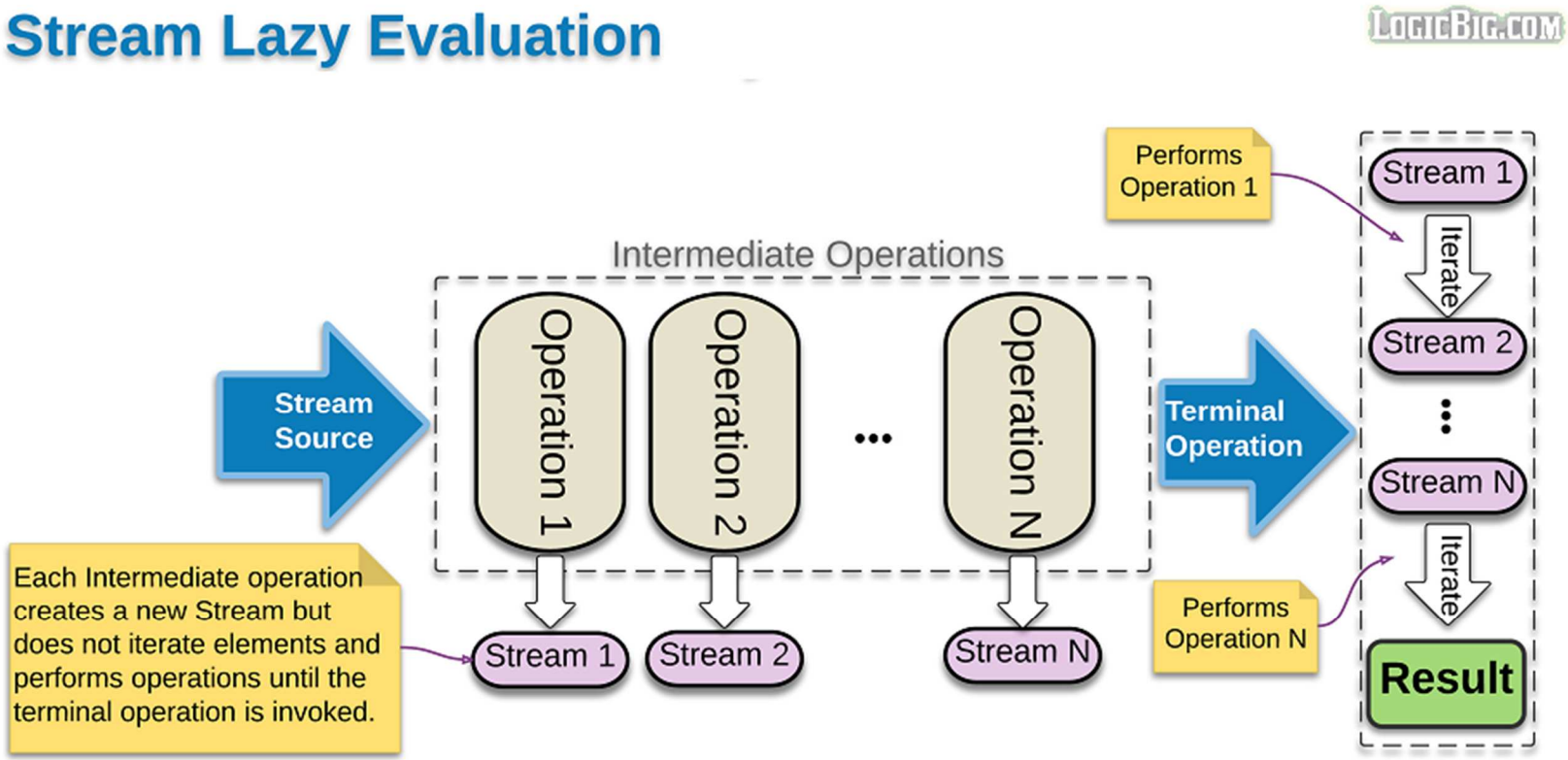
Compléments

Laziness (Une des règles de base de l'informaticien)

- Rappel: Les flux (Streams) ont deux type d'opérations
  - ✓ Les opérations intermédiaires et terminales
- Lorsque l'on enchaîne les opérations sur un flux
  - ✓ le code de chaque opération n'est pas exécuté sur l'instant avant de passé à l'opération suivant.
    - ➢ On ne traite pas toutes les données d'une opération avant de passer à la suivante.
  - ✓ On prend un éléments du flux, et on lui fait subir toute la chaîne d'opération
  - ✓ Les "Consumer" (génération de flux) et les "Supplier" ne sont exécutés que quand nécessaire
  - ✓ Théorie des "pipelines"

Compléments

Laziness (Une des règles de base de l'informaticien)



Compléments

Laziness (Une des règles de base de l'informaticien)

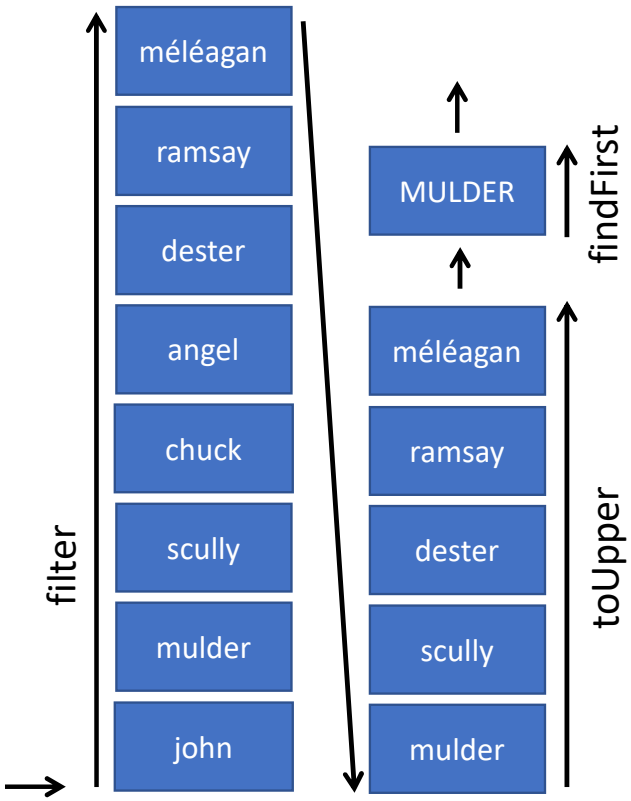
- Pour tester le flot, on peut utiliser la méthode `peek(System.out::println)`
- Exemple (non optimisé, illustratif) :

```
Public static void main (final String args[]) {

      List <String> prenoms = Arrays.asList
            ("john", "mulder", "scully", "chuck", "angel",
             "angel", "dexter", "ramsay", "méléagan");

      final String premierPrenomLong = prenoms.stream()
            .filter(pn -> pn.length() > 5)
            .map (String::toUpper)
            .findFirst().get();
}
```

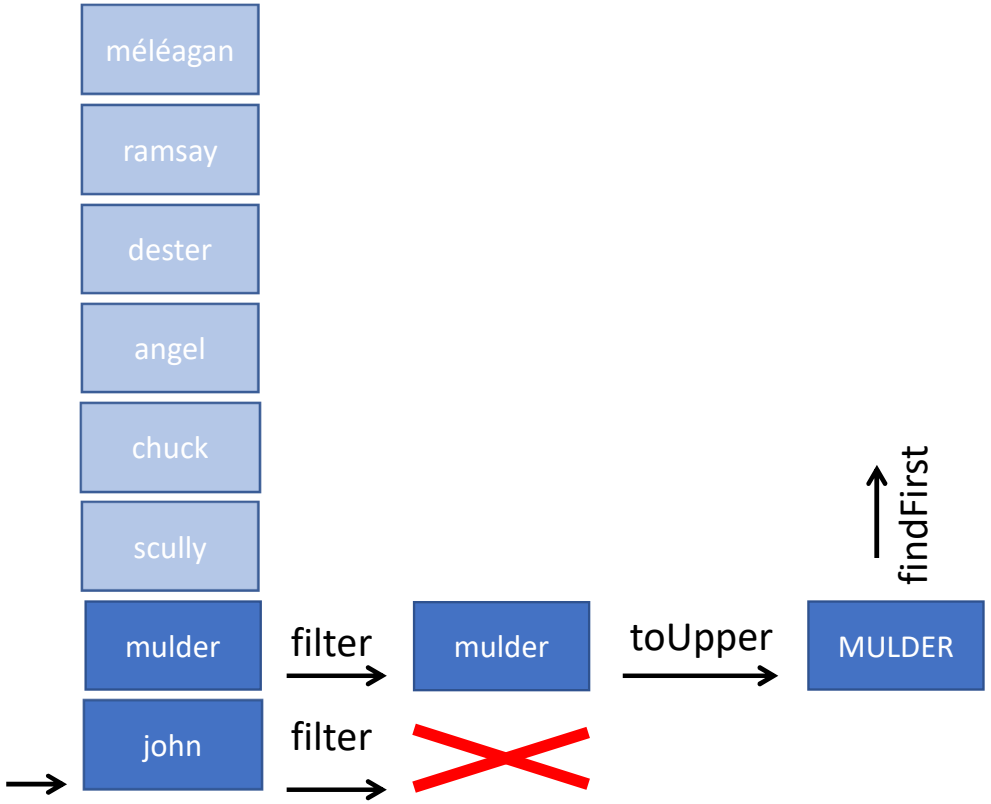
Compléments

Laziness (Une des règles de base de l'informaticien)

Evaluation avide



Evaluation paresseuse



Compléments

Laziness (Une des règles de base de l'informaticien)

- L'évaluation avide n'a pas applicable à une source de données infinies
- L'évaluation paresseuse est invalidée par certaines méthodes intermédiaires
  - ✓ Exemple: sort (qui a besoin de consommer l'ensemble du flux pour trier)
  - ✓ Exemple: flatmap invalide partiellement la lazyness.
- L'évaluation paresseuse est plus tolérante sur l'ordre des opérations
  - ✓ Ainsi `.map(…).limit(10)` ou `.limit(10).map(…)` sont équivalents

- Voir les livres pour plus de détails

Compléments

Parallélisme : flux Parallèles

- L'évaluation paresseuse est un premier pas vers l'asynchronisme
- L'évaluation paresseuse est un premier pas vers la parallélisme
- Parallélisation du traitement des flux :
  - ✓ la méthode magique "`.parallel()`" et son inverse "`.sequential()`"
  - ✓ Il n'y a pas de contrôle fin du parallélisme. Le dernier appel change l'état global.
- "`.parallel`()" délègue le traitement à un pool d'exécution (threads) pour l'ensemble (chaque) des flux parallèles
  - ✓ Le contrôle n'est pas fin. C'est un outil simple et rapide.

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.paralelism","12");
```

- Nécessite un combineur à la fin puisque le flux est découpé en sous flux
  - ✓ Un combinateur (`Combiner`) souvent implicite est prédéfini, sinon il doit être défini.
  - ✓ Contre exemple de combinateur prédéfini : Voir `Collect`() et son "`combiner`"
  - ✓ Un combineur peut être coûteux

Compléments

Parallélisme : flux Parallèles

- La parallélisation a un coût qui peut la rendre inintéressante
  - ✓ Création des partitions (temps non négligeable, puis fusion)
  - ✓ Le Scatter (`Spliterator`/`parallel`), gather (`Combiner`) peuvent être coûteux
  - ✓ Nécessite des mesures de tests
- Vérifier le coût du un/boxing des éléments utilisés et les éviter au maximum
- Certains opérations ne sont pas efficacement parallélisables
  - ✓ Exemple: `.findFirst()` ou `.limit()`
  - ✓ Même si l'évaluation paresseuse offre une bonne résistance aux déperditions de performances dû à l'ordre des opérations, leur influence n'est pas totalement nulle.
  - ✓ Le contrôle de l'ordre éléments dans le flux par `.unordered()`
  - ✓ Des problèmes encore avec .`distinct`() sur des flux ordonnés ou non

Compléments

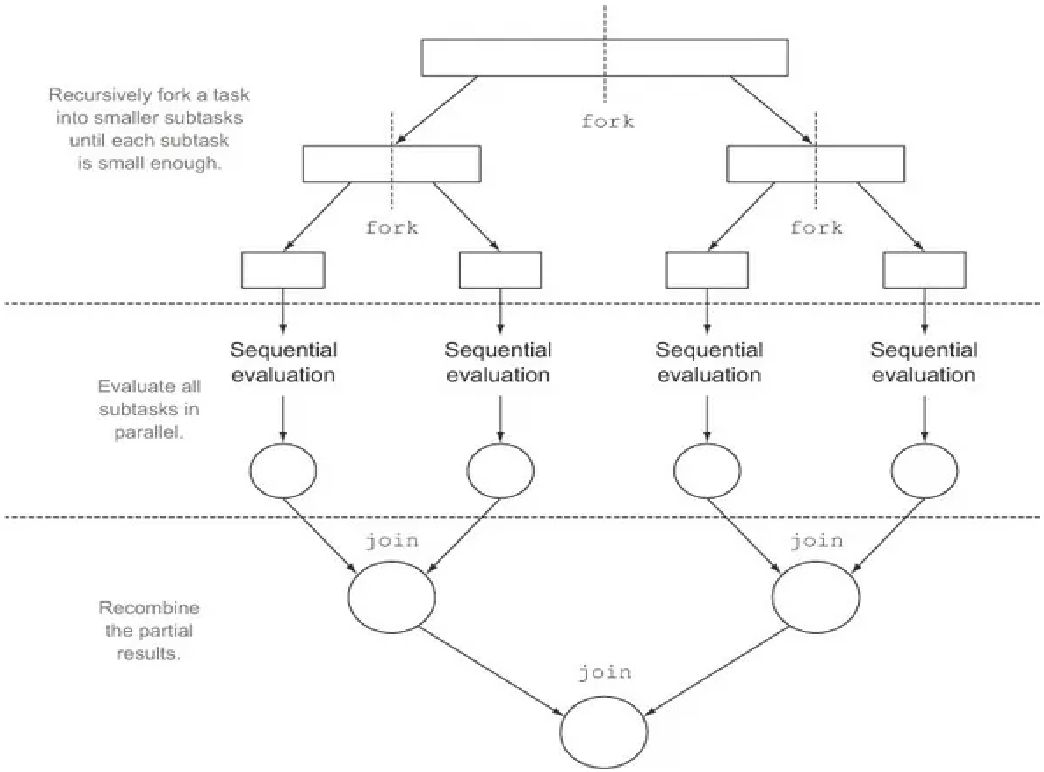
Parallélisme : flux Parallèles

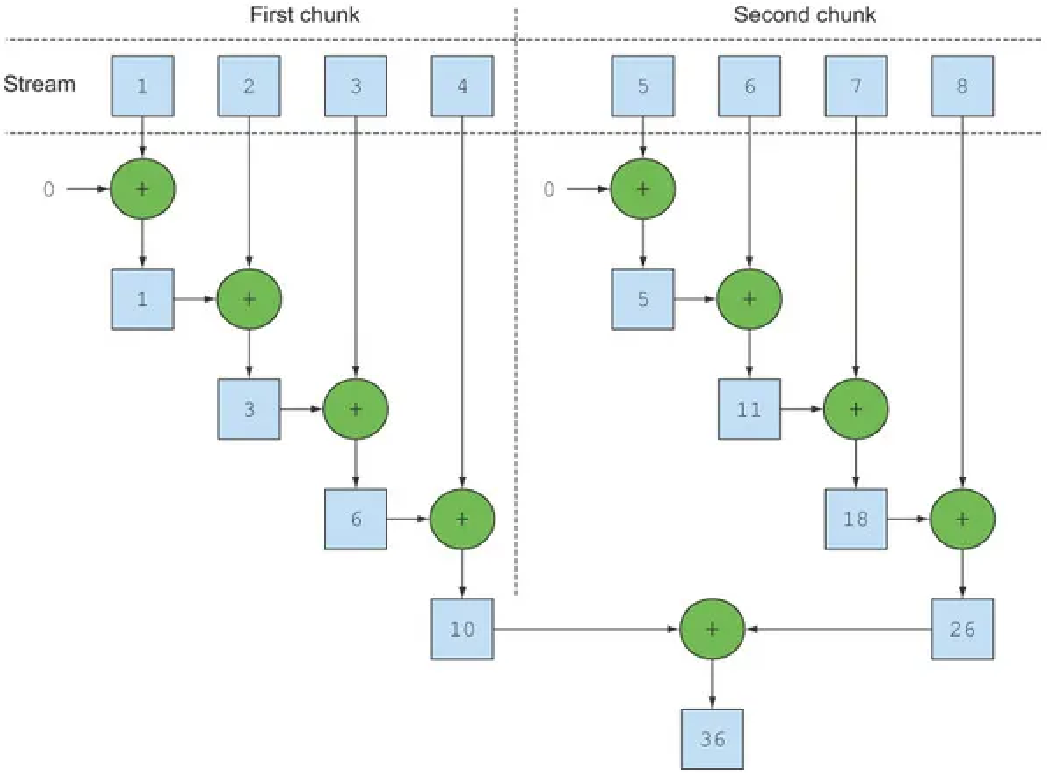
- Certaines sources de données sont plus décomposables que d'autres
  - ✓ `ArrayList`() à base de tableau ou un `range` se décomposent bien et vite
  - ✓ `LinkedList`() mal parce qu'elle est à base de liste chaînée (COÛTEUX)
  - ✓ Nécessite des analyses des sources de données et des essais
- La parallèlisation se fait par le découpage des données en lots (`Spliterator`) qui sont confiés à des "`Executor`" (un pool de thread)
  - ✓ Rappel: `Spliterator` partitionne les données en 2 partitions
  - ✓ Découpage par division par 2 récursives en fonction du nombre de Core
- Vous pouvez implémenter
  - ✓ Vos propres `Spliterator` pour la décomposition des vos propres sources de données
  - ✓ Des outils de découpages de données en lots récursifs et synchroniser leurs traitement via une sous classe de `ForkJoinTask` qui est une `RecursiveTask`.
    - ➢ Danger: Mélange complexe de PF et de concurrence

Compléments

Parallélisme : flux Parallèles

Images de Java 8 in Action, ISBN 978-1-617291-99-9





Compléments

Parallélisme : Futurs / Asynchronisme

- Lorsqu'il programme une `RecursiveTask`, le développeur donne le code :
  - ✓ De répartition des données
  - ✓ De synchronisation pour attendre les résultats
  - ✓ De fusion des résultats
- Mais l'utilisation du framework `ForkJoin` nécessite la compréhension des cours de concurrence.
- En particulier des `Executors` :
  - ✓ C'est un pool de thread dont le seul but est d'exécuter des tâches en boucle
  - ✓ Ces tâches sont confiés par un thread en boucle infinie qui gère la liste des tâches
- Les traitements des sous flux parallèles, après décomposition du flux de données, sont confiés à une instance de `ExecutorService`.

Compléments

Parallélisme : Futurs / Asynchronisme

- La synchronisation utilise une sous classe des "`Future`" : les "`CompletableFuture`"
- Ainsi, on peut définir des exécuteurs de tâches.
  - ✓ Voir `Executor`, `ExecutorService`, `Executors` et les cours de concurrence en Java
- On peut confier deux types de travaux à des exécuteurs
  - ✓ `Runnable` / `Callable`
  - ✓ Un `Runnable` est une tâche sans résultat (donc inadaptée à la PF)
  - ✓ Un `Callable` est une tâche avec un résultat
  - ✓ Le résultat d'une tâche `Callable` n'est pas immédiatement disponible
  - ✓ Il faut donc une boîte qui accueillera (dans le futur) le résultat: un `Future`

Compléments

Parallélisme : Futurs / Asynchronisme

- Les travaux (`Runnable` / `Callable`) s'expriment aisément en Lambda
  - ✓ Ils peuvent être soumis via la méthode `submit`()

- Pour un `Runnable` :
  - ✓ ici le `Future` retournera null comme valeur à la fin du traitement
  - ✓ Un `Runnable` doit implémenter la méthode `run` qui renvoie `void`
  - ✓ `Future<?> submit(Runnable task)`

- Pour un `Callable` :
  - ✓ Ici le `Future` contiendra le résultat de la tâche à la fin du traitement
  - ✓ Un `Callable` doit implémenter la méthode `call` qui renvoie `T`
  - ✓ `<T> Future<T> submit(Callable<T> task)`

Compléments

Parallélisme : Futurs / Asynchronisme

- Il existe plusieurs politiques d'exécuteurs
  - ✓ Une instance d'exécuteur est un `ExecutorService`
  - ✓ Encore une fois faire la différence entre l'interface et la classe d'outils
    Interface `Executor` / Classe `Executors` / Classe `ExecutorService`
- Exemple

```
ExecutorService myExecutor = Executors.newCachedThreadPool();
Callable<Double> cd = () -> { return veryLongCalculus(); }
Future<Double> future = executor.submit (cd);
```

- Il existe ensuite des méthodes sur le `Future`
  - ✓ Méthode `get`() pour attendre le résultat, `isDone`() pour tester la terminaison

Compléments

Parallélisme : Futurs / Asynchronisme

- Les `Future` ne sont pas complétement adaptés aux lambda calculs et aux flux.
- La combinaison des résultats (indépendants) de 2 tâches via des `Future` n'est pas prévu.
  - ✓ Hors c'est utile pour les flux parallélisés.
- Attendre la fin de tout ensemble de tâches via des `Future` n'est pas prévu.
  - ✓ Hors c'est utile pour les flux parallélisés.
- Attendre la fin de la première de tâche via des `Future` n'est pas prévu.
  - ✓ Par exemple pour obtenir un résultat via la plus rapide de plusieurs méthodes
- Ajouté un trigger pour exécuter un code (par ex. de combinaison) avec des `Future` n'est pas prévu
  - ✓ Hors c'est utile pour les flux parallélisés.

Compléments

Parallélisme : Futurs / Asynchronisme

- Le `CompletableFuture` a une logique PF Java
  - ✓ Chaînage de méthodes
  - ✓ Il y a des méthodes pour soumettre des travaux (sous forme de lambda par ex)
    - `supplyAsync`, `runAsync` (méthodes statiques)
  - ✓ Il y a de nombreuses méthodes pour synchroniser des CompletableFuture
    - `allOf` → crée un nouveau `CompletableFuture` qui attends le fin de tous les `CompletableFuture`
    - `anyOf` → crée un nouveau `CompletableFuture` qui attends le fin du plus rapide des `CompletableFuture`
  - ✓ Il y a des méthodes pour récupérer les résultats
    - `get`() ou `join`()

Compléments

Parallélisme : Futurs / Asynchronisme

- Le `CompletableFuture` a une logique PF Java
  - ✓ Il y a des méthodes pour appliquer des traitements à la fin des tâches
    - `thenAccept`() pour appliquer un `Consumer` (opération terminale)
    - `thenApply`() pour appliquer une `Function` (qui peut donc être chaîné)
    - Il existe des version asynchrones (`thenApplyAsync`)

- Nombreuses autres Opérations `whenComplete`(), `thenRun`(), `thenCompose`(), …

Compléments

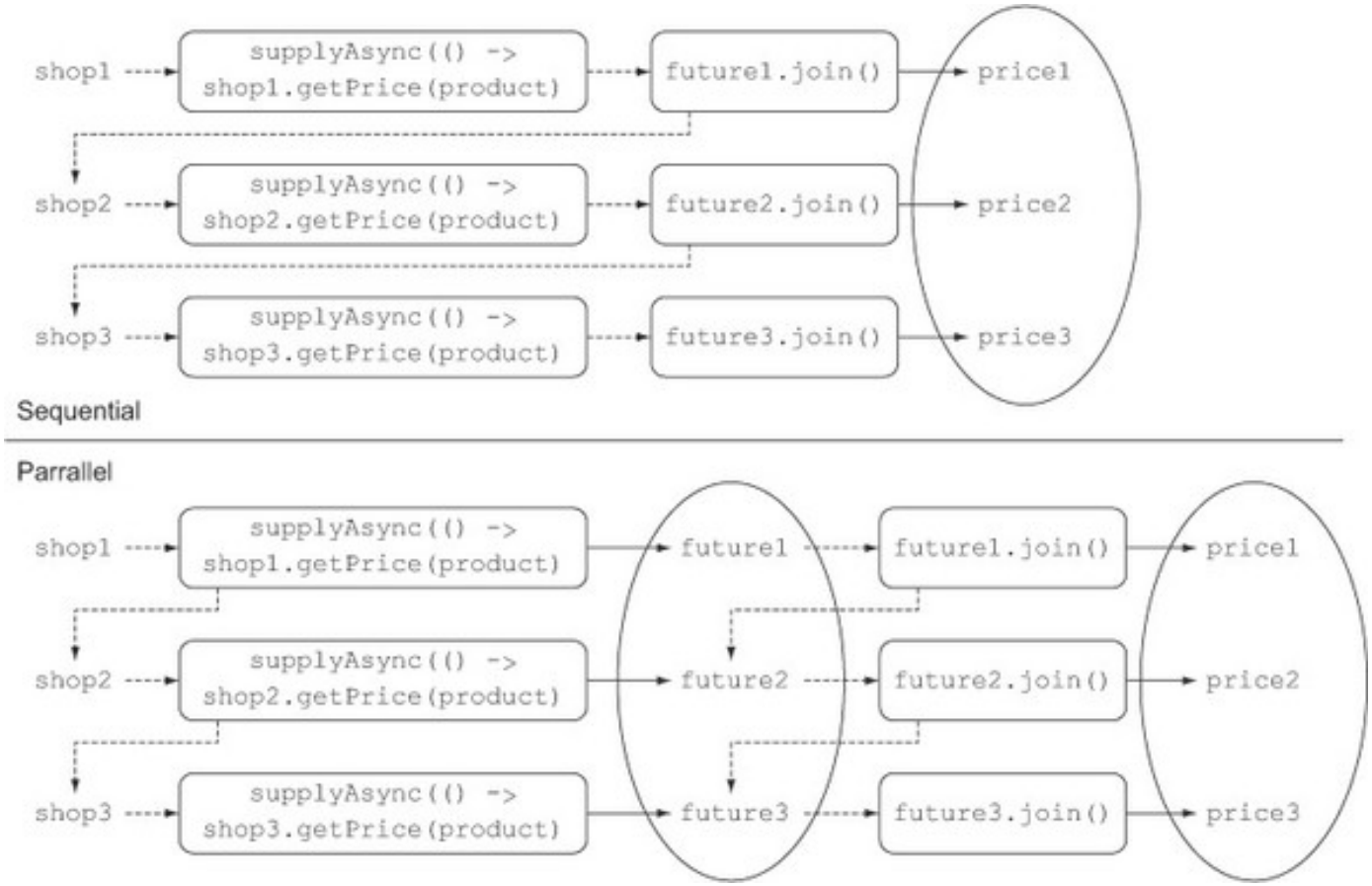
Parallélisme : Futurs / Asynchronisme

- Attention, l'évaluation paresseuse peut détruire le parallélisme
- Exemple:

```
public List<String> findPrices(String product) {
  List<CompletableFuture<String>> priceFutures =
      shops.stream()
           .map(shop -> CompletableFuture.supplyAsync(
              () -> shop.getName() + " price is " +
                              shop.getPrice(product)))
           .collect(Collectors.toList());
  return priceFutures.stream()
          .map(CompletableFuture::join)
          .collect(toList());
}
```

Compléments

Parallélisme : Futurs / Asynchronisme



Images de Java 8 in Action
ISBN 978-1-617291-99-9

- L'évaluation paresseuse force l'évaluation d'un élément complétement avant l'évaluation du suivant.
- Si le flux est séquentielle, même si les lambdas sont asynchrones, on attend le résultat avant de consommer l'éléments suivant
- Si le flux est parallelisable, les lambdas asynchrones seront exécutés en parallèles en fonction du pool.

Compléments

Bibliographie

- P.-Y. Saumont. « Functional Programming in Java ».
  Manning, 2017. ISBN-13: 978-1617292736.
  accessible gratuitement depuis :
  https://www.manning.com/books/functional-programming-in-java

- V. Subramaniam. « Functional Programming in Java, Harnessing the Power of Java 8 Lambda Expressions ».
  The Pragmatic Programmers, 2014. ISBN-13: 978-1937785468.

- J. Bloch. « Effective Java, 3rd Edition ».
  Addison-Wesley, 2018. ISBN-13: 978-0134685991.

Compléments

Webographie

- Oracle. « Lesson: Generics (Updated) ». https://docs.oracle.com/javase/tutorial/java/generics
- https://www.freecodecamp.org/news/functional-programming-in-java-course/
- https://flyingbytes.github.io/programming/java8/functional/part0/2017/01/16/Java8-Part0.html
- https://blog.jooq.org/2015/12/08/3-reasons-why-you-shouldnt-replace-your-for-loops-by-stream-foreach/
- https://www.javaworld.com/article/3314640
- https://tutorials.jenkov.com/java-functional-programming/index.html

- Tom Harding. « Fantas, Eel, and Specification ». http://www.tomharding.me/fantasy-land
- https://www.youtube.com/watch?v=K6BmGBzIqW0
- https://www.youtube.com/watch?v=Ee5t_EGjv0A
- https://www.youtube.com/watch?v=aRYIEoh5tPk
- https://chrisdone.com/posts/dijkstra-haskell-java/
- https://wiki.haskell.org/Typeclassopedia
- http://eed3si9n.com/learning-scalaz/

Compléments

Langages

- ✓ (Java)
- ✓ Scala
- ✓ Kotlin (android)
- ✓ Haskell (« natif »)

Librairies

- ✓ Java [VAVR]: https://www.vavr.io
  « turns java upside down »
- ✓ Kotlin [ARROW] : https://arrow-kt.io
  « a library for Typed Functional Programming in Kotlin »
- ✓ Scala [Cats] : https://typelevel.org/cats/
  « lightweight, modular, and extensible library for functional programming »