L'API Java Stream

Introduction

Depuis la création du JDK 1.0 en 1996, le langage Java a connu de nombreuses versions apportant chacune leur lot de nouveautés.

La version 8 de Java offre des changements plus radicaux qui influent sur la manière d'écrire le code (plus facilement). La simple ligne de code suivante vous en convaincra :

```
inventory.sort(comparing(Apple::getWeight));
```

Ce code permet de trier une collection de pommes sur base de leur poids.

Cette ligne de code très simple résume la clarté et la puissance du Java 8 qui exploite au mieux les possibilités des processeurs multi-cœurs actuels.

Les langages de programmation actuels tendent vers des besoins en données de plus en plus copieux ; les programmes doivent manipuler de grandes quantités de données appelés les Big Data. Il s'agit de collections de multiples téraoctets qu'il faut exploiter grâce aux processeurs multi-cœurs. Le Java 8 fournit une nouvelle API qui supporte des opérations en parallèle pour traiter ces grandes données et ressemble aux requêtes dans les bases de données. Le programmeur exprime ce qu'il veut et l'implémentation (l'API Stream) choisit le meilleur mécanisme de bas niveau pour exécuter celui-ci. Inutile de jouer avec des synchronisations qui sont source de nombreuses erreurs mais aussi moins efficaces avec des processeurs multi-cœurs.

L'**API Stream** est intimement liée à deux autres concepts provenant du Java 8 : les expressions **lambda** et les **méthodes défauts** dans les interfaces.

En outre, Java 8 permet de **paramétriser le comportement des méthodes** en permettant de passer des **fonctions** en argument. Ce qui ouvre le Java à un autre style de programmation appelé la programmation fonctionnelle.

La **programmation fonctionnelle** est un paradigme de programmation (au même titre que la programmation procédurale ou objet) dans lequel le rôle central est donné aux fonctions. En programmation procédurale, les instructions modifient les états des variables. En programmation fonctionnelle, il n'y a pas d'opération d'affectation. Il s'agit d'emboiter les fonctions qui agissent comme des boites noires que l'on peut imbriquer les unes dans les autres. Chaque boîte possède des paramètres en entrée mais une seule sortie ; il n'y a qu'une seule valeur possible pour chaque tuple de valeurs présentées en entrée. Cette façon de penser, très différente des autres démarches de programmation, est parfois difficile à comprendre pour des programmeurs formés aux langages « traditionnels ».

Java 8

Avant de se plonger dans l'API Java Stream, il est impératif de bien comprendre différents nouveaux concepts apportés par le Java 8. Ce qui leur permet aux streams d'être aussi riches, ce sont tous les éléments apportés par cette version 8 du Java : les fonctions Java, les méthodes default, les expressions lambda, etc.

Les fonctions en Java

Le Java 8 introduit la possibilité de considérer une **fonction** comme une **valeur** ; ceci facilite notamment l'usage des Streams.

Supposons que l'on souhaite obtenir un tableau des fichiers cachés d'un répertoire. Il faut d'abord écrire une méthode qui indique si un fichier est caché ; il s'agit de la méthode `isHidden` fournit par la classe `File`. La méthode `accept` qui prend un `File` en paramètre et indique s'il est caché lui délègue donc le travail. Pour utiliser cette fonction, il faut créer un `FileFilter` qui est transmis à la méthode `listFiles`.

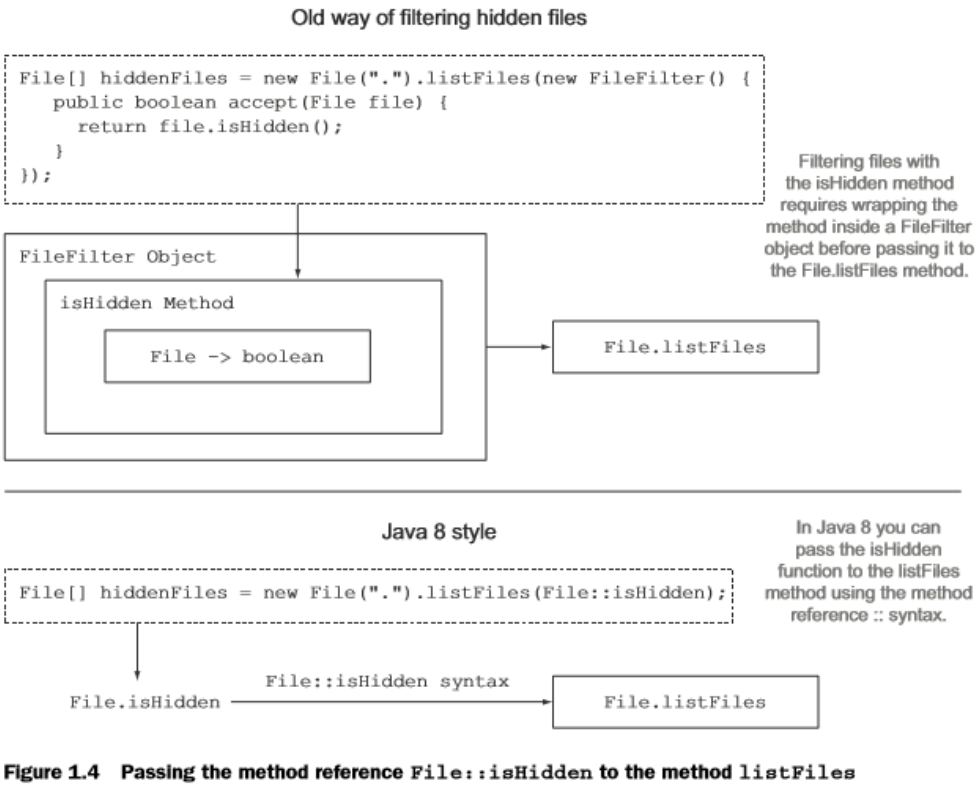

Figure 1.4 Passing the method reference `File::isHidden` to the method `listFiles`

Figure 1 Extrait du livre Java 8 in action

En Java 8, il suffit d'écrire l'unique ligne de code apparaissant ci-dessus. L'usage de la syntaxe `::` permet de référencer la méthode comme une valeur. Cette référence de méthode passée à la méthode `listFiles` en permet l'usage.

Les **expressions Lambda** sont encore plus puissantes car ce sont des fonctions anonymes qu'on écrit de façon très concise : `(int x)->x+1` signifie que lorsqu'on appelle la fonction avec `x`, elle renvoie `x+1`.

Voici un exemple avec une collection de pommes :

```
public static <T> Collection<T> filter(Collection<T> target, Predicate<T> predicate) {
	Collection<T> result = new ArrayList<T>();
	for (T element: target) {
		if (predicate.test(element)) {
			result.add(element);
		}
```

```
        }
        return result;
}
public static void main(String[] args) {
        List<Apple> inventory = Arrays.asList(new Apple(80,"green"), new Apple(155,
        "green"), new Apple(120, "red"));

        Collection<Apple> inventory_big_green = filter(inventory,
                (Apple a) -> a.getWeight()>150 && "green".equals(a.getColor()));
        }
```

`Predicate`, argument de la fonction `filter`, est une interface fonctionnelle (voir après) qui repose sur le concept mathématique de prédicat. En mathématique, un prédicat est comme une *fonction* qui prend une valeur en paramètre et renvoie vrai ou faux.

Il est important de bien comprendre que ce qui est effectivement transmis à la méthode `filter` correspond à de la **paramétrisation de comportement**. On indique ce qu'il faut faire, c'est la méthode qui se charge de les accomplir en exécutant ce comportement paramétré.

Les méthodes default

Les collections Java se transforment en stream comme par magie. Notre liste de choses devient en un seul appel de méthode un fameux stream …
Reprenons justement cet exemple, avant Java 8, `List<T>` ne contenait pas la méthode `stream()`.

Ajouter une méthode abstraite `stream()` dans `Collection` et une méthode concrète dans `ArrayList` aurait pu convenir pour notre cas mais quel cauchemar pour toutes les autres classes qui étendent `Collection` qu'il aurait alors fallu modifier. Il devenait impossible de faire évoluer les interfaces sans interrompre le fonctionnement des implémentations de celle-ci, quel dilemme ! C'est pour cela que Java 8 permet désormais d'avoir des **méthodes default** dans les interfaces ; les classes ne doivent pas fournir d'implémentation pour celles-ci. Il n'est donc plus obligé d'ajouter des implémentations dans toutes les classes qui implémentent des interfaces évoluant. Abracadabra !

Par exemple, il est possible d'invoquer la méthode `sort()` sur une `List`, pourtant, cette dernière n'en possède aucune implémentation directe mais bénéficie de l'implémentation par défaut de `Collection`. De plus, il est possible de paramétriser le comportement du sort en lui fournissant en argument un `Comparator`.

Pour reprendre notre exemple de pommes :

```
inventory.sort((Apple a1,Apple a2)->a1.getWeight().compareTo(a2.getWeight())) ;
```

Le `Comparator` en argument correspond à une expression lambda, autre joyau du Java 8.

Les expressions Lambda

Pour rappel, une **expression lambda** est une représentation concise d'une classe anonyme : elle n'a pas de nom mais possède une liste de paramètres, un contenu, un type de retour et éventuellement des exceptions. Il s'agit en fait d'une **fonction** ; ce n'est pas une classe mais plutôt une méthode avec des paramètres, retour, etc.

Les grands atouts d'une expression lambda sont sa **concision** et la possibilité de la passer en **argument**.

```
(Apple a1,Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

paramètres | flèche | contenu

On utilise des expressions lambda lorsqu'on travaille avec des **interfaces fonctionnelles** (`@FunctionnalInterface`). Une interface fonctionnelle est une interface qui contient **une seule méthode abstraite** ; par exemple `Comparator`, `Predicate`, Consumer, Function, `Runnable`, `Callable`, …

`Consumer` possède une méthode `accept` qui prend un type générique en paramètre et ne renvoie rien du tout.

`Predicate` possède une méthode `test` qui prend un type générique en paramètre et renvoie un boolean.

`Function` possède une méthode `apply` qui prend un type générique en paramètre et renvoie un autre type générique.

Remarquez qu'il existe des interfaces fonctionnelles pour les types primitifs qui permettent d'éviter l'emballage de ceux-ci dans des objets plus couteux (par exemple `Integer` pour `int`). Par exemple : `IntPredicate`, `IntFunction`, …

Exemples d'expressions :

`(Apple a)-> System.out.println(a.getWeight())` correspond à l'interface fonctionnelle `Consumer`.

`(int a) -> a>85` correspond à l'interface fonctionnelle `IntPredicate`.

`(String s) -> s.length()` correspond à l'interface fonctionnelle `Function`.

Une **expression lambda** fournit directement **l'implémentation** de la **méthode abstraite** d'une **interface fonctionnelle** et traite l'expression comme une instance de cette interface.

Les références de méthode

Avec les expressions lambda, on écrit une fonction que l'on passe, par exemple, en paramètre d'une méthode `sort`.

```
inventory.sort((Apple a1,Apple a2)->a1.getWeight().compareTo(a2.getWeight())) ;
```

Si l'on souhaite utiliser une méthode déjà existante, il faut faire ce que l'on appelle une **référence à cette méthode** (`::`). Dans l'exemple, il aurait être naturel d'utiliser la méthode `comparing()` déjà existante dans `Comparator`.

```
inventory.sort(Comparator.comparing(Apple::getWeight) ;
```

Remarquez que la référence de méthode ne contient pas d'invocation de méthode; il s'agit de fournir la méthode définie (`getWeight`) dans quelle classe (`Apple`).

Il s'agit d'une abréviation d'expression lambda. Voyez ces exemples :

1. `System.out::println` pour `(String s) -> System.out.println(s)` qui correspond à un objet `Consumer<String>.`
2. `String::length` pour `(String s) -> s.length()` qui correspond à un objet `Function<String, Integer>.`
3. `List::contains` pour `(list,a)->list.contains(a)` qui correspond à un objet `BiPredicate<List<String>, String>` considérant que `list` est une collection de `String`.
4. `Apple::getWeight` pour `(Apple a)-> a.getWeight()` qui correspond à un objet `Function<Apple, Integer>`.

Les références de méthode sont donc :

- des références vers des méthodes de classe (`static`) : exemple 1.
- des références vers des méthodes des types Java : exemples 2 et 3.
- des références vers des méthodes d'objets existant : exemple 4.

Supposant que la classe Apple possède les constructeurs suivants :

```
public Apple(){}
public Apple(int weight, String color){
	this.weight = weight;
	this.color = color;
}
```

Il est également possible de référencer des constructeurs :

```
Supplier<Apple> supp = Apple::new;
Apple app = supp.get();

BiFunction<Integer, String, Apple> biFunct = Apple::new;
Apple app2 = biFunct.apply(90, "red");
```

Dans ces cas, c'est l'invocation des méthodes `get` et `apply` qui produit les objets `Apple`.

Quelques méthodes utilisées par les expressions lambda

L'API Java fournit un certain nombre de méthodes bien sympathiques dans ses interfaces fonctionnelles. Il est possible de combiner celles-ci pour composer des expressions plus compliquées.

Comparator

Pour trier en ordre **décroissant**, il suffit d'invoquer `reversed` :

```
inventory.sort(comparing[1](Apple::getWeight).reversed());
```

Il est possible de **chainer** davantage la comparaison :

```
inventory.sort(comparing(Apple::getWeight).reversed().thenComparing(Apple::getColo
r));
```

[1] `comparing` est une méthode de classe de `Comparator`, ces lignes de codes fonctionnent avec un `import static java.util.Comparator.*`.

Predicate

Il est possible de combiner les prédicats en utilisant les méthodes `negate, and et or.`

```
Predicate<Apple> pomRouge = a->"red".equals(a.getColor());
Predicate<Apple> pomNonRouge = pomRouge.negate();
Predicate<Apple> pomRougeGrosse = pomRouge.and(a->a.getWeight()>150);
```

Function

Les fonctions sont également combinables par le biais des méthodes `andThen` et `compose`.

`f1.andThen(f2)` applique la fonction f2 en paramètre sur le résultat de la fonction f1. En bref, on opère f2(f1(x)).

`f1.compose(f2)` fournit le résultat de f2 en argument de f1. En bref, on opère f1(f2(x)).

Les objets Java Streams

Presque toutes les applications Java manipulent des collections, ce qui est souvent couteux en nombre d'objets créés et en lignes de codes nécessaires. De plus, il n'est pas toujours évident de comprendre ce code contenant de multiples instructions, conditionnels, etc.

Un **stream** permet de manipuler efficacement les grands volumes de données de façon déclarative : on écrit une requête plutôt qu'une implémentation de code complexe. Les streams traitent en parallèle des données de façon transparente pour le développeur (exploitation des multi-cœurs) et en pipeline pour éviter les intermédiaires de calculs.

Un stream est une séquence d'éléments provenant d'une source qui supportent des traitements de données :

- il ne possède pas les données qu'il traite ;
- il conserve l'ordre des données ;
- il s'interdit de modifier les données qu'il traite ;
- il traite les données en une passe ;
- il est optimisé algorithmiquement et est capable de calculer en parallèle.

En Java:

```
List<Chose> choses = … ;
Stream<Chose> stream = choses.stream();
```

Pour exécuter un stream de façon à exploiter une architecture multi-cœurs, il suffit de changer l'appel `stream()` en `parallelStream()`. C'est l'API Stream qui se charge de paralléliser les traitements des données !

Remarquez qu'il est également possible de générer des streams autrement que par le biais des collections, par exemple à partir d'une ressource I/O.

A partir d'un stream (`java.util.stream.Stream`), il est possible de faire beaucoup de choses (petit coup d'œil à la javadoc) : filtrer, trier, mapper, sommer, parcourir, réduire, compter, … nous y venons !
Considérons la classe suivante pour les exemples :

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;}

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    public enum Type { MEAT, FISH, OTHER }
    @Override
    public String toString() {
        return name;
    }

    public static final List<Dish> menu =
            Arrays.asList( new Dish("pork", false, 800, Dish.Type.MEAT),
                           new Dish("beef", false, 700, Dish.Type.MEAT),
                           new Dish("chicken", false, 400, Dish.Type.MEAT),
                           new Dish("french fries", true, 530, Dish.Type.OTHER),
                           new Dish("rice", true, 350, Dish.Type.OTHER),
                           new Dish("season fruit", true, 120, Dish.Type.OTHER),
                           new Dish("pizza", true, 550, Dish.Type.OTHER),
                           new Dish("prawns", false, 400, Dish.Type.FISH),
                           new Dish("salmon", false, 450, Dish.Type.FISH));
}
```

L'instruction suivante résume ce qu'il est possible de faire avec les streams :

```
menu.stream()
                .filter(d -> d.getCalories() > 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .limit(3)
                .collect(toList[2]());
```

Comprendre ce code est relativement simple :

1) `stream()` fournit une séquence d'objets `Dish`.
2) `filter()` sélectionne ceux dont les calories excèdent 400.
3) `sorted()` trie par ordre croissant de calories.
4) `map()` extrait le nom du `Dish`.
5) `limit()` sélectionne les 3 premiers.
6) `collect()` stocke le résultat dans une nouvelle liste.

Opérer cette instruction en Java traditionnel aurait été très fastidieux et couteux en nombre de lignes de codes et d'objets créés.

L'**internalisation de l'itération** est un grand atout des streams. Prenons un exemple de la vie d'étudiant pour comprendre l'itération externalisée des collections :

Un poil demande à un bleu de vider les verres. Il lui demande s'il reste encore un verre.

Le bleu répond oui, il reste un verre de blanche.

Le poil lui demande de le boire et lui demande s'il en reste encore un autre verre.

Le bleu répond oui, il reste un verre de blonde.

Le poil lui demande de le boire et lui demande s'il en reste encore un autre verre.

Le bleu répond oui, il reste un verre de triple.

Le poil lui demande de le boire et lui demande s'il en reste encore un autre verre.

Le bleu répond oui, il reste un verre de brune.

Le poil lui demande de le boire et lui demande s'il en reste encore un autre verre.

Le bleu répond non. Le poil est satisfait !

[2] La méthode `toList` est une méthode de classe `Collectors` ; l'import `static` **`import static`** `java.util.stream.Collectors.`*`toList`* est bien pratique. Il est également possible d'invoquer la méthode `toSet` pour collecter les données dans un `set`.

En version stream, cela donne :

Un poil demande à un bleu de boire toutes les bières.

Le gros avantage de cette manière de faire est que si le bleu veut boire 2, 3, 4 bières en même temps, il peut le faire. Deuxièmement, si le bleu préfère boire la triple d'abord, il peut le faire. S'il est plus efficace en buvant d'abord la triple, qu'il le fasse !

C'est exactement comme cela que l'API Stream opère, elle internalise l'itération ; ce qui lui permet une meilleure gestion des éléments.

Un autre grand atout des streams est le **pipelining**. Beaucoup d'opérations des streams renvoient également des streams de sorte que cela permet le traitement des éléments de la collection dans un **pipeline**. Il s'agit d'**opérations** dites **intermédiaires** ; elles renvoient un autre stream en résultat, ce qui permet de les connecter pour former une requête. Il s'agit des opérations `filter`, `map`, `sorted`, `distinct` ou `limit`. Les **opérations terminales**, elles, produisent un résultat à partir d'un pipeline. Ce résultat n'est pas un stream, il peut s'agit d'un nombre, d'une liste ou même `void`. Il s'agit des opérations comme `collect`, `forEach` ou `count`.

Utiliser les Streams

Filtrer et réduire un stream

filter()

L'API Stream offre une méthode `filter()` qui prend en argument un `Predicate` et renvoie un autre stream incluant tous les éléments qui respectent le prédicat.

```
List<Dish> vegetarianMenu = menu.stream()
                .filter(Dish::isVegetarian)
                .collect(toList());
vegetarianMenu.forEach(System.out::println);
```

Ce code permet d'afficher en console les plats végétariens du menu.

distinct()

L'API Stream offre une méthode `distinct()` qui renvoie un stream dans lequel chaque élément s'y trouve une seul fois selon l'implémentation du `hashCode` et `equals` de l'élément.

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
          .filter(i -> i % 2 == 0)
          .distinct()
          .forEach(System.out::println);
```

Ce code permet d'afficher en console les nombres pairs de la liste en s'assurant qu'il n'y aura pas de doublon.

limit()

L'API Stream offre une méthode `limit()` qui renvoie une stream dont la taille est limitée par l'argument fourni.

```
List<Dish> dishesLimit3 =menu.stream()
                .filter(d -> d.getCalories() > 300)
```

```
                .limit(3)
                .collect(toList());
dishesLimit3.forEach(System.out::println);
```

Ce code permet d'afficher en console les 3 premiers plats de plus de 300 calories. Si la source est un Set dans lequel l'ordre importe peu, 3 plats seront sélectionnés sans aucune contrainte d'ordre.

skip()

L'API Stream offre une méthode `skip()` qui prend en argument un entier n et qui enlève les n premiers éléments. Si n est supérieur à la taille du stream alors, `skip` renvoie un stream vide.

```
List<Dish> dishes =menu.stream()
                .filter(d -> d.getCalories() > 300)
                .skip(2)
                .collect(toList());
dishes.forEach(System.out::println);
```

Ce code permet d'afficher en console les plats de plus de 300 calories en excluant les deux premiers éléments.

Mapper

En programmation, il est courant de devoir sélectionner des informations de certains objets comme en sql lorsque l'on sélectionne une colonne d'une table. Les méthodes `map` et `flatMap` de l'API Stream offrent ces possibilités.

map()

L'API Stream offre une méthode `map()` qui prend en argument une fonction. Cette fonction est alors appliquée à chaque élément pour créer un nouvel élément. `map` renvoie un autre stream dont le type des éléments correspond à celui du renvoi de la fonction en paramètre.

```
List<String> dishNames = menu.stream()
                       .map(Dish::getName)
                       .collect(toList());
System.out.println(dishNames);
```

Ce code permet d'afficher les noms de tous les plats.

flatMap()

L'API Stream offre une méthode `flatMap()` qui opère plus ou moins la même chose que `map` sauf qu'elle travaille sur des Streams et les "aplatit" pour en renvoyer un seul.

```
List<Dish> dishes1 = menu.stream().filter(Dish::isVegetarian).collect(toList());
List<Dish> dishes2 = menu.stream().filter(a -> a.getCalories() <= 400)
                                                        .collect(toList());

List<List<Dish>> myListsDishes = Arrays.asList(dishes1, dishes2);
List<Dish> myDishes = myListsDishes.stream().flatMap(dishes -> dishes.stream())
                                                    .distinct().collect(toList());
```

Ce code permet de collecter les plats depuis 2 streams en 1 seul stream en supprimant les doublons.

Trouver

L'API Stream offre aussi la possibilité de trouver si des éléments correspondent à une propriété donnée par le biais des méthodes : `allMatch`, `anyMatch`, `noneMath`, `findFirst` et `findAny`.

anyMatch()

L'API Stream offre une méthode `anyMatch()` qui permet de trouver s'il y a un élément qui correspond au prédicat en argument. Cette méthode renvoie un boolean et est donc une opération terminale.

`menu.stream().anyMatch(Dish::isVegetarian)` renvoie un boolean qui indique s'il y a un plat végétarien dans le menu.

allMatch()

L'API Stream offre une méthode `allMatch()` qui permet de trouver si tous les éléments correspondent au prédicat en argument. Cette méthode renvoie un boolean et est donc une opération terminale.

`menu.stream().allMatch(dish-> dish.getCalories()<=700)` renvoie un boolean qui indique si tous les plats du menu ont moins de 700 calories.

noneMatch()

L'API Stream offre une méthode `noneMatch()` qui est l'opposée de `allMatch()`.

`menu.stream().noneMatch(dish-> dish.getCalories()>=1000)` renvoie un boolean qui indique si aucun plat du menu a plus de 1000 calories.

L'API Stream **court-circuite** parfois l'**évaluation** des expressions quand cela est possible. Par exemple, dans notre cas précédent dès qu'un plat de plus de 1000 calories est trouvé on stoppe le traitement des autres éléments du stream.

findAny()

L'API Stream offre une méthode `findAny()` qui permet de sélectionner n'importe quel élément.

`menu.stream().filter(Dish::isVegetarian).findAny()` permet de sélectionner n'importe quel plat végétarien.

Observez le code `menu.stream().filter(dish-> dish.getCalories()>=1000).findAny()`. A priori, il permet de sélectionner n'importe quel plat de plus de 1000 calories. Malheureusement, il n'y a pas de plat dans le menu excédant 1000 calories.

Heureusement, l'API Stream, pour éviter les bugs dus à `null`, a introduit la classe **`Optional`** qui permet de représenter l'existence ou l'absence de résultat. Il s'agit en fait d'un conteneur de réponse qui indique s'il y a une réponse (`isPresent`) ou encore fournit la réponse (`get`).

`menu.stream().filter(dish-> dish.getCalories()>700).findAny().get()` renvoie le `Dish` *pork*.

`menu.stream().filter(dish-> dish.getCalories()>=1000).findAny().isPresent()` renvoie donc false.

findFirst()

Lorsque l'ordre des éléments du stream importe (basé sur une liste ou une autre collection triée), il est parfois nécessaire de sélectionner le premier élément. La méthode `findFirst` permet de sélectionner ce premier élément. Si la place de l'élément retourné importe peu, il est préférable d'utiliser `findAny` car cette méthode est moins contraignante (surtout avec des streams parallèles).

Réduire

reduce()

La méthode `reduce()` fourni par l'API Stream permet de réduire le stream à un résultat unique en effectuant une opération donnée :

- une valeur initiale du même type que celles contenues dans le stream et qui doit être un neutre pour l'opération à effectuer.
- un `BinaryOperator` qui indique l'opération à effectuer c.-à-d. comment combiner deux éléments pour produire une nouvelle valeur. Par exemple l'expression lambda `(a,b) -> a+b` correspond à un opérateur binaire.

```
List<Integer> numbers = Arrays.asList(3,4,5,1,2,-5);

int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

Ce code correspond à la boucle suivante en *vieux* Java :

```
int sum = 0 ;
for(int i : numbers){
	sum+=x ;
}
```

En Java 8, `Integer` propose une méthode de classe `sum`, ce qui rend notre code encore plus concis :

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

Il est également possible de sélectionner le minimum ou maximum en changeant 3 petites lettres :

```
int max = numbers.stream().reduce(0, Integer::max);
int min = numbers.stream().reduce(0, Integer::min);
```

L'opération fournie doit être associative : on ne peut, par exemple, pas donner comme expression lambda `(a,b) -> a/b`.

Il existe une surcharge de la méthode `reduce` qui ne prend pas de valeur initiale en paramètre. Cette version renvoie alors un objet `Optionnal`.

min() et max()

Il existe également des méthodes `min` et `max` qui prennent en argument un `Comparator` et permettent de rechercher les éléments minimum ou maximum selon le critère de comparaison fourni.

```
Dish dish = menu.stream().min(comparing(Dish::getCalories)).get();
```

Ce code récupère le plat le plus pauvre en calories.

Les streams numériques

Les streams primitifs

Lorsqu'on manipule des streams de types primitifs comme `int`, il est nécessaire de passer par les classes d'emballage (`Integer`). Par exemple, pour additionner les calories de tous les plats, il faut écrire :

```
int totCal = menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

On constate que la réduction avec des objets `Integer` est coûteuse car chaque `int` fourni par le stream va être emballé dans un `Integer`.

L'API Stream a donc introduit 3 types de streams primitifs : `IntStream`, `DoubleStream` et `LongStream` qui permettent d'éviter ces coûts d'emballage.

En utilisant un stream primitif, on obtient :

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
System.out.println("Number of calories:" + calories);
```

On peut forcer la transformation d'une `IntStream` en `Stream<Integer>` avec la méthode `boxed`.

Sur un `IntStream`, on peut invoquer les opérations `min` et `max` qui renvoient alors un `OptionnalInt`. Par exemple :

```
OptionalInt maxCalories = menu.stream()
       .mapToInt(Dish::getCalories)
       .max();
int max = maxCalories.orElse(-1);
System.out.println(max);
```

La génération de IntStream

Il est possible de générer des nombres avec les streams en utilisant les méthodes `range` et `rangeClosed`. Ces méthodes prennent en premier argument la valeur de départ et en second la valeur de fin. La valeur de départ est toujours incluse tandis que la valeur de fin sera exclusepar la méthode `range` et incluse par la méthode `rangeClosed`.

```
List<Integer> evenNums = IntStream.rangeClosed(1, 100).boxed().collect(toList[3]
());
```

Ce code liste les entiers de 1 à 100 inclus.

Créer des streams

Il existe d'autres façons de créer des streams qui n'utilisent pas les collections ou la génération d'entiers :

à partir de valeurs	`Stream<String> sString = Stream.of("Leconte", "Damas", "Ferneeuw")`
à partir d'un tableaux	`int[] tab = { 1, 5, 4, 8 };` `IntStream intStream = Arrays.stream(tab);`

à partir d'un fichier	`try(Stream<String> lignes = Files.lines(Paths.get("data.txt"), Charset.defaultCharset())){`
	`  List<String> lS = lignes.collect(toList());`
	`  lS.forEach(System.out::println);`
	`}catch (IOException e) {`
	`// TODO: handle exception`
	`}`

à partir d'une fonction	`Stream.iterate(0, n -> n + 2)`
	`  .limit(10).forEach(System.out::println);`
	`Stream.generate(Math::random)`
	`  .limit(10)`
	`  .forEach(System.out::println);`
	`//Ces méthodes renvoient des stream infinis (unbounded streams) ; l'une à partir d'une valeur initiale en l'incrémentant avec un UnitaryOperator, l'autre à partir d'un Supplier.`

Collecter les données

L'opération terminale `collect` permet de récupérer tous les éléments du stream dans une liste ou un ensemble mais pas seulement … elle permet aussi, en fonction des arguments fournis, de collecter et traiter les éléments de différentes manières afin d'obtenir un résultat unique, qu'il soit de type complexe comme une `map` ou de type simple comme un `int` par exemple.

Réduire et résumer

Une des versions de la méthode `collect` prend en paramètre un objet d'une classe implémentant l'interface `Collector`. La classe utilitaire `Collectors` fournit un bon nombre de méthodes utiles pour créer des instances de `Collector` prêtes à l'emploi comme `toList` (pour récupérer les éléments du stream dans une liste) ou encore `toSet`. Nous allons explorer les autres possibilités fournies par la classe `Collectors`. Mais, sachez qu'il est aussi possible de créer des `Collector` sur mesure en implémentant directement l'interface.

Collectors.counting()

Cette méthode permet de compter le nombre d'éléments. Les deux lignes de code suivantes sont équivalentes :

```
long nbDishes = menu.stream().collect(counting());
long nbDishes = menu.stream().count();
```

[3] Sans oublier l'import `static : import static java.util.stream.Collectors.*;`

Collectors.minBy() – Collectors.maxBy()

Les méthodes `minBy()` et `maxBy()` permettent de calculer les valeurs minimum et maximum d'un stream.

Par exemple, pour trouver le plat le plus pauvre en calories :

```
menu.stream().collect(minBy(comparing(Dish::getCalories)))
```

Ce code renvoie un `Optionnal<Dish>`.

Collectors.summingInt() – Collectors.summingLong() – Collectors.summingDouble()

Les méthodes `summingXXX()` permettent de calculer la somme d'une fonction passée en paramètre.

```
menu.stream().collect(summingInt(Dish::getCalories))
```

fournit le nombre total de calories des plats du menu.

Collectors. averagingInt ()

```
menu.stream().collect(averagingInt(Dish::getCalories))
```

calcule la moyenne des calories des plats du menu.

Collectors.summarizingInt()

Il est assez fréquent lorsqu'on désire obtenir le minimum d'un stream que l'on souhaite également le maximum ou encore la moyenne. Pour éviter des parcours inutiles, il existe une méthode qui résume toutes les informations `summarizingInt` ; cette méthode place les informations calculées, qui pourront être ensuite récupérées, dans un objet de type `IntSummaryStatictics`. Il existe des méthodes équivalentes pour les `Long` et les `Double`.

Ce code

```
System.out.println(menu.stream().collect(summarizingInt(Dish::getCalories)));
```

affiche

```
IntSummaryStatistics{count=9, sum=4300, min=120, average=477,777778, max=800}
```

Collectors.joining()

La méthode `joining` permet de concaténer en une seule `String` toutes les `Strings` du stream.

```
System.out.println(menu.stream().map(Dish::toString).collect(joining(",")));
```

Le paramètre facultatif `","` permet une meilleure lisibilité du résultat puisqu'il délimite chaque `String`.

Collectors.reducing()

Toutes les méthodes abordées dans cette partie, `counting`, `minBy`, … sont en fait des méthodes qui effectuent une opération de façon agréable et lisible pour le programmeur. Par exemple, le code
`menu.stream().collect(reducing((d1,d2)->d1.getCalories()>d2.getCalories()?d1:d2))`
est équivalent à
`menu.stream().collect(maxBy(comparing(Dish::getCalories))`

Ce code permet d'obtenir un `Optional<Dish>` qui contient le plat le plus riche en calories.

L'API Stream offre diverses manières de réaliser les mêmes opérations ; certaines plus lisibles, d'autres plus générales ou réutilisables ou encore personnalisables.

Regrouper

Le regroupement (`group by`) est une opération fréquente en base de données mais elle devient vite complexe en programmation traditionnelle. Par exemple, pour regrouper les plats selon leur type :

```
Map<Type, List<Dish>> platsGroupes = new HashMap<>();
for (Dish d : menu) {
      Type t = d.getType();
      if (platsGroupes.get(t) == null)
            platsGroupes.put(t, new ArrayList<Dish>());
      platsGroupes.get(t).add(d);
}
```

En Java 8, la ligne de code suivante suffit :
`Map<Type, List<Dish>> pGroup2 = menu.stream().collect(groupingBy(Dish::getType));`

`groupingBy` est une fonction de classification. Elle peut reposer sur une propriété de l'élément mais aussi sur des critères plus complexes.

Considérant l'`enum CaloricLevel { DIET, NORMAL, FAT }`

```
menu.stream().collect(
            groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            } ))
```

permet de construire une `Map<CaloricLevel, List<Dish>>`.

Il est également possible de faire des groupements multiples.

```
menu.stream().collect(
        groupingBy(Dish::getType,
             groupingBy((Dish dish) -> {
                 if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                 else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                 else return CaloricLevel.FAT;
             } )
        )
 )
```

Ce code permet de construire une `Map<Dish.Type, Map<CaloricLevel, List<Dish>>>`.

On peut également transmettre en second argument de la méthode `groupingBy` n'importe quel autre collector :

```
Map<Type, Long> pg = menu.stream().collect(groupingBy(Dish::getType,counting()));
```

qui compte le nombre de plats de chaque type.

Ou encore

```
Map<Type, Optional<Dish>> pg = menu.stream().collect(
            groupingBy(Dish::getType,maxBy(comparingInt(Dish::getCalories))));
```

qui fournit le plat le plus riche en calories de chaque type. L'`Optionnal<Dish>` récupéré par la comparaison n'est pas vraiment utilisable. Il est alors possible de transformer le résultat de la collection par la méthode `collectingAndThen` :

```
Map<Type, Dish> p2 = menu.stream().collect(
            groupingBy(Dish::getType,collectingAndThen(
                  maxBy(comparingInt(Dish::getCalories)),Optional::get)));
```

Partitionner

Le partitionnement est un cas particulier de regroupement basé sur un prédicat appelé fonction de partitionnement.

```
Map<Boolean, List<Dish>> vegs =
menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

`vegs` contient `{false=[pork, beef, chicken, prawns, salmon], true=[french fries, rice, season fruit, pizza]}`

`vegs.get(true)` équivaut à `menu.stream().filter(Dish::isVegetarian).collect(toList());`

L'avantage du partitionnement est que l'on conserve tous les plats du stream.

Bibliographie

Urma R.G., Fusco M. and Mycroft A., Java 8 in action (Lambdas, streams, and functional-style programming), United States Of America, 2015, Manning, https://www.manning.com/books/java-8-in-action