Introduction

Le langage Java a subi plusieurs changements depuis JDK 1.0, ainsi que de nombreuses additions de classes et de packages à la bibliothèque standard.

Java 8 est sorti en Mars 2014. Les modifications apportées à Java étaient plus profondes que toute modificationdans son histoire. Ces changements ajoutent le concept de la programmation fonctionnelle qui permet d'écrire des programmes plus facilement.

C'est quoi la programmation fonctionnelle ?

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions.

La programmation fonctionnelle repose sur plusieurs principes :

Immutabilité :

Ce concept permet de s'assurer de la valeur des variables du début à la fin de nos actions. Cela va permettre d'avoir un code plus robuste et plus stable, donc moins de bugs et moins de maintenances. Une fonction ne doit pas modifier les variables qui lui sont passées en paramètre.

Les fonctions d’ordre supérieur :

Un mécanisme puissant des langages fonctionnels est l'usage des fonctions d'ordre supérieur. Une fonction est dite d'ordre supérieur lorsqu'elle peut prendre des fonctions comme arguments (aussi appelées callback) et/ou renvoyer une fonction comme résultat. On dit aussi que les fonctions sont **des objets de première classe**, ce qui signifie qu'elles sont manipulables aussi simplement que les types de base.

Récursivité

Il est possible d'utiliser la programmation fonctionnelle de manière récursive.Cela va permettre d'avoir un code plus lisible et plus court. De plus, le code s'auto documenter de lui-même.

Java 8 comme un langage fonctionnel

Java 8 introduit un nouveau API (Streams) qui supporte plusieurs opérations de processus des données et ressemble les requêtes des langages de base de données.

L'addition des Streams au Java peut être considérée comme une cause directe des deux autres additions : les techniques concises pour passer le code aux méthodes (les expressionsLambda et les références des méthodes), et les méthodes par défaut dans les interfaces.

Supposant qu'on veut écrire deux méthodes qui diffèrent en seulement quelque lignes de code ; on peut maintenant passer juste le code des parties qui diffèrent comme argument, cette technique de programmation est plus courte, plus claire et plus fiable que la tendance commune à copier et coller tout le code.

On passant du code aux méthodes (et aussi être capable de les retourner et l'intégrer dans des structures de données),on fournit un accès à un ensemble de techniques supplémentaires qui sont connus comme un style de programmation fonctionnelle. Par conséquent, Java 8 fait un pas vers la programmation fonctionnelle.

Lambda expressions

Les Lambda expressions,Un des plus grandes fonctionnalités du java 8, facilite la programmation fonctionnelle et simplifie le développement.

Avec les expressions Lambda, on peut passer le code d'une manière concis, sans avoir utilisé les classes anonymes ou écrire des classes que vous n'utilisez qu'une seule fois dans votre code.

Le résultat est un code plus clair, plus concise et plus flexible. Par exemple on peut utiliser Lambda expression pour créer un comparateur objet :

Avant :

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
        }
};
```

Après(Lambda expression):

```
Comparator<Apple> byWeight =
                (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Remarque que le code est plus simple après l'utilisation des Lambda expressions.

Syntaxe :

Une expression lambda peut être comprise comme une représentation concise d'une classe anonyme:

- Il n'a pas de nom comme une méthode normale.
- Capable d'être passé comme un argument d'une méthode ou une valeur dans un variable.
- Concise : moins de code àécrire contrairement aux classes anonymes.

Une Lambda expression est composée des paramètres, une flèche, et un corps :

```
(Paramètres)->Corpsde l'expression
```

- Une liste des paramètres : dans l'exemple on a les paramètres du méthode compare d'un Comparator.
- La flèche : sépare les paramètres et le corps.
- Le corps : dans l'exemple il compare deux objets, l'expression est considérée comme la valeur du retour.

Parmi les caractéristiques d'une expression lambda :

- Déclarations du type est optionnel pour les paramètres. Les parenthèses sont optionnelles et requis uniquement pour multiples paramètres.
- Si le corps a une seule instruction les accolades ne sont pas nécessaires.
- Le mot clé **return** n'est pas nécessaire si on a une seule instruction.

Exemples d'utilisations :

Cas d'utilisationExamples

Boolean:**(List<String> list) -> list.isEmpty()**

Création des objets :**() -> new Apple(10)**

Consummation:**(Apple a) -> {**
System.out.println(a.getWeight());
}

Selection/extraction:**(String s) -> s.length()**

Combinaison des valeurs :**(int a, int b) -> a * b**

Comparaison: **(Apple a1, Apple a2) ->a1.getWeight().compareTo(a2.getWeight())**

Lambda expressions En Action

```
publicclassJava8Tester{
publicstaticvoid main(String args[]){


System.out.println("13 + 7 = "+ operate(13,7,(int a,int b)-> a+b));

System.out.println("19 x 5 = "+ operate(19,5,(int a,int b)-> a*b));



Greetingmessage1 = message ->System.out.println("Hello "+ message);

Greetingmessage2 = message->System.out.println("Hello "+ message);


message1.sayMessage("khalid");
message2.sayMessage("anass");

}


interfaceGreeting{
void sayMessage(String message);
}



interfaceMathOperation{
int operation(int a,int b);



int operate(int a,int b,MathOperation mathOperation){
return mathOperation.operation(a, b);
```

Interfaces fonctionnelle

Introduction

Parfois on utilise des interfaces avec exactement une seule méthode abstraite, ces interfaces sont appelés **les interfaces fonctionnelles (functional interfaces)**.

Les expressions lambda correspondent en fait à des types spécifiés par ces interfaces.

Remarque : une interface qui spécifie une seule méthode abstraite est toujours une interface fonctionnelle même si elle a plusieurs méthodes par default.

Par exemple si on prend les interfaces suivant :

```
public interface Adder{
        int add(int a, int b);
  }

public interface SmartAdder extends Adder{
        int add(double a, double b);
  }

public interface Nothing{
  }
```

Parmi ces interfaces, Adder est la seule interface fonctionnelle.

SmartAdder n'est pas une interface fonctionnelle parce qu'elle spécifie deux méthodes abstraites : add et la méthode add hérité de l'interface Adder.

Nothing n’est pas une interface fonctionnelle, car elle ne déclare aucune méthode abstraite.

Lambda expressions et les interfaces fonctionnelles

Les interfaces fonctionnelles sont utiles parce que la signature de la méthode abstraite peut décrire la signature d’expression Lambda, Cette signature est appelé le descripteur de fonction.

Donc pour utiliser des expressions Lambda différents, on a besoin d’un ensemble des interfaces fonctionnelles déjà disponible dans la Java API comme Comparable, Predicate etc…

Parmi les nouvelles interfaces fonctionnelles introduites dans java.util.function package :

- **Predicate :**

  L’interface **Predicate<T>**définit une méthode abstraite appelé **test**qui accepte un objet générique de type T et renvoie un booléen :

```
@FunctionalInterface
public interface Predicate<T>{
        boolean test(T t);
}
```

  On peut utiliser cette interface pour représenter une expression booléen qui utilise un argument de type T, par exemple :

```
        public static <T> List<T> filter(List<T> list, Predicate<T> p) {
            List<T> results = new ArrayList<>();
           for(T s: list){

                if(p.test(s)){

                    results.add(s);
            }
}
return results;
        }
//cette prédicat pour tester si un argument String est vide ou non.
  Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
//après on utilise la méthode filter qu'on a défini au début.
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
//la méthode filter prend une liste et un prédicat comme arguments et
renvoie une nouvelle liste tester par le prédicat.
```

- ## Consumer :

L'interface **Consumer<T>**définit un méthode abstraite appelé **accept** qui prend un objet générique de type T et renvoie aucun résultat (void) :

```
        @FunctionalInterface
        public interface Consumer<T>{
            voidaccept(T t);
        }
```

On peut utiliser cette interface si on veut accéder un objet de type T et effectuer certaines opérations, par exemple :

```
public static <T> void forEach(List<T> list, Consumer<T> c) {
            for(T i: list){
```

```
                c.accept(i);
            }
    }
    ForEach(Arrays.asList(1,2,3,4,5),(int i) -> System.out.println(i));
```

//la méthode **forEach** prend une liste et un Consumer comme arguments et applique la méthode **accept** du Consumer aux éléments de liste.
//dans ce cas, **accept** prend les éléments de la listes et les affiche.

- ## Function :

L'interface **Function<T,R>** définit un méthode abstraite appelé **apply** qui prend un objet générique de type T et renvoie un objet générique de type R :

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

On peut utiliser cette interface si on veut définir une Lambda qui mappe des informations de l'entrée à la sortie, par exemple :
Si on veut transformer une liste des chaines de caractères à une liste des nombres entiers contenant la longueur de chaque chaine :

```
public static <T,R> List<R> map(List<T> list, Function<T,R> f) {
        List<R> result = new ArrayList<>();
        for(T i: list){
            result.add(f.apply(i));
        }
        return result;
}
```

```
List<Integer> l=map(Arrays.asListe("lambdas","in ","action") , (String s)- > s.length() );
```

/* l'expression lambda implémentée par la méthode apply renvoie la longueur de la chaine donnée. C'est-à-dire la liste l va contenir [7, 2, 6] */

Méthodes utiles pour composer les expressions Lambda

Plusieurs interfaces fonctionnelles in Java 8 contiennent des méthodes pratiques qui permettent de composer plusieurs expressions lambda pour construire des expressions plus compliquées. Spécifiquement les interfaces comme Comparator, Function et Predicate.

- ## Composition des Comparators :

On sait que l'interface Comparator utilise La methode statique Comparator.comparing pour retourner un Comparator :

```
Comparator<Apple> c = Comparator.comparing(Apple::getWeight);
```

Ordre inversée :

Si on veut trier une liste de type Apple par poids décroissant, on n'a pas besoin de construire une autre instance de Comparator. L'interface inclut une méthode par default **reversed()** qui inverse l'ordre de tri :

```
Inventory.sort( comparing(Apple::getWeight).reversed() );
```

Chainage des Comparators :

Pendant le tri, si on trouve deux pommes avec le même poids, on peut construire un autre Comparateur pour enrichir la comparaison. Pour cela on utilise la méthode par default thenComparing qui prend une fonction comme

argument et permet la méthode de sort d'avoir un Comparateur secondaire utilisé dans le cas d'égalité pour le premier comparateur :

```
Inventory.sort( comparing(Apple::getWeight)
        .reversed() );//tri par poids décroissant.
        .thenComparing(Apple::getCountry)) ;/* tri par getCountry dans le cas d'egalite pour le premier comparateur */
```

- ## Composition des Predicates :

  L'interface Predicate a trois méthodes qui permettent de réutiliser des prédicats existants pour créer des prédicats plus compliquées.

  Negate() : on peut utiliser **negate()** pour retourner la négation d'un prédicat :

```
Predicate<Apple> notRedApple = redApple.negate();
/* negate() retourne la negation du prédicat redApple,c'est-à-dire la pomme qui est pas rouge */
```

  and() :on peut combiner deux Lambdas pour exprimer qu'une pomme est par exemple est rouge et lourd :

```
Predicate<Apple>  redAndHeavyApple = redApple.and(a-> a.getWeight()> 150 );
```

  Or() : on peut aussi combiner deux Lambdas pour exprimer deux choix diffèrent :

```
Predicate<Apple>  redAndHeavyOrGreen = redApple.and(a-> a.getWeight()> 150 )
.or(a-> "green".equals(a.getColor()) );
```

  /* on a créé un prédicat qui exprime les pommes qui sont rouge et lourd OU les pommes vert */

- ## Composition des Functions :

  Finalement, on peut composer des Lambdas représentée par l'interface Function.

andThen() : cette méthodeapplique la premier fonction donnée puis applique la deuxième fonction et renvoie le résultat :

```
Function<Integer, Integer>  f = x -> x + 1;
Function<Integer, Integer>  g = x -> x * 2;
Function<Integer, Integer>  h = f.andThen(g) ;// similaire au  g( f(x) )
int result = h.apply(1);
//le resultat est 4
```

Compose() : cette méthode est similaire à **andThen()**, la seule différence est que compose applique la deuxième fonction puis la premier et renvoie le résultat :

```
Function<Integer, Integer>  f = x -> x + 1;
Function<Integer, Integer>  g = x -> x * 2;
Function<Integer, Integer>  h = f.compose(g) ;// similaire au f( g(x) )
int result = h.apply(1);
//le resultat est  3
```

Références de méthodes

Les références de méthodes sont utiles pour remplacer des expressions Lambda qui font appel à une méthode qui existe déjà.
Par exemple on peut remplacer :

```
Function<Integer, String> toString = i -> String.valueOf(i);
```

Par:

```
Function<Integer, String> toString = String::valueOf;
```

Mais pourquoi utiliser les références de méthodes ? On peut les considérée comme un raccourci pour les lambda qui appelle une seule méthode existant.

Syntaxe :

Lorsqu'on a besoin d'une référence de méthode, la référence cible est placé avant le séparateur :: et la méthode est placé après.

Construction de la référencede méthode

Il y a trois sortes des références de méthodes :

1. Une référence à une méthode statique.

   Par exemple la méthode *parseInt* de *Interger* : **Integer::parseInt**

2. Une référence à une instance de méthode d'un type.

   Par exemple la méthode *length*de *String* : **String::length**

3. Une référence à une instance de méthode d'un objet existant.

   Par exemple on suppose qu'on a un variable *Nom* qui contient un objet de type *String* et qui supporte une méthode*length* : **Nom::length**

Exemples des Lambdas et leur référence de méthode équivalente :

Lambdaréférence de Méthodeéquivalente
(Apple a) -> a.getWeight()Apple::getWeight
() -> Thread.currentThread().dumpStack() Thread.currentThread()::dumpStack
(str, i) -> str.substring(i) String::substring
(String s) -> System.out.println(s) System.out::println

Remarque :

On peut créer une référence pour un constructeur existant on utilisant le mot clé **new.** Ça marche similairement aux méthodes statiques :

```
Supplier<Apple> c1 = Apple::new;
//equivalent à Supplier<Apple> c1 = new Apple();
Apple a1= c1.get();
```

Streams

Introduction

Les Streams sont une nouvelle couche abstraite introduite dans Java 8, qui nous permet de traiter des collections desdonnées d'une manière similaire aux requêtes SQL.

Avant Java 8, on a utilisé des boucles et des contrôles répétés pour tester et filtrer par exemple. Pour résoudre le problème, Java 8 a introduit un concept qui permet de traiter des donnés d'une manière déclarative et plus simple :

Par exemple, si on veut sélectionner les employés avec un salaire *>4000 :*

Java7

```
List<Emp> e=new ArrayList<>() ;

for(Emp p :LesEmp)

 {    if(p.getSalary()<4000)

e.add(p);

}

Collections.sort(e,new Comparator<Emp>(){

public  int compare(Emp e1,Emp e2) {

returnDouble.compare(e1.getSalary(),e2.getSalary());

}});

List<String>lesNomEmp=new ArrayList<>();

for(Emp  p:e)

lesNomEmp.add(p.getName());
```

Dans cet exemple, on a créé un variable **e** qu'on va utiliser une seule fois. Avec Java 8, cette implémentation n'est plus utilisée :

Avec java8

```
List<String>lesNomEmp=LesEmp.stream()
.filter(d ->d.getSalary()<4000)
                                        .sorted(comparing(Emp::getSalary))
                                        .map(Emp::getName)
.collect(toList());
```

Qu'est-ce qu'un Stream ?

Un Stream représente un flux d'objets provenant d'une source, et qui supportedes opérations de traitement de données.

Voici les caractéristiques des Streams :

- **Un courant d'objets :**un Stream fournit un ensemble d'éléments de type spécifique d'une manière séquentielle. Le courant traite les éléments sur demande, et ne stocke jamais comme les collections.

- **Source :** les Streams prend les Collections, Arrays, ou les ressources I/O comme une source d'entrée.

- **Operations de traitement de données :** les Streams offre des opérations similaire aux opérations utilisés en base des données (SQL par exemple) pour manipuler et traiter les données comme ***filter, map, reduce***…

- **Pipelining :** Les opérations renvoie un Stream, permettant à enchainer et former des flux plus grands.

- **Itérations automatique ou interne :** avec les Streams, on fait des itérationsinternes sur les éléments fournis, contrairement aux collections où l'itération explicite est nécessaire.

Prenant l'exemple précédant pour expliquer ces idées :

```
List<String>lesNomEmp=LesEmp.stream()
//génère un stream d'après  'LesEmp'
                                        .filter(d ->d.getSalary()<4000)
//création d'un "pipeline",commençant par filtrer les gens avec des salaires <4000
.sorted(comparing(Emp::getSalary))
//tri par salaire
                                        .map(Emp::getName)
 //prend les noms des éléments de Stream
                                        .collect(toList());
                                    //stocker le résultat dans une autre liste.
```

Streams vs. Collections

La différence entre les Collections et les Streamsréside dans la façon de traitement des données. Une collection est une structure de données en mémoire qui contient toutes les valeurs. Chaque élément de la collection doit être traité avant qu'il puisse être ajouté à la collection, on peut ajouter et retirer des éléments, mais chaque moment dans le temps, les éléments sont stockés dans la mémoire.

En revanche, un Stream est un flux de données fixe (on ne peut pas ajouter ou supprimer des éléments) dont les éléments sont calculés sur la demande.

L'idée est que l'utilisateur extrait uniquement les valeurs dont il a besoin d'un Stream, et ces éléments sont produits invisiblement pour l'utilisateur et lorsque c'est nécessaire.

Traversable une seule fois

Un Stream est traversable une seule fois. Après le Stream est consommé, on peut obtenir un nouveau Stream à partir de la source de données initiale, sinon, si on essaie d'appliquer une opération pour la deuxième fois, le code lance une exception. Par example:

```
List<String> title = Arrays.asList(«java8», « on », « action ») ;

Stream<String> s = title.stream();

s.forEach(System.out::println);

s.forEach(System.out::println);// java.lang.IllegalStateException
```

Itération externe vs. Itération interne

Une autre différence entre les Collections et les Streams est la façon de gestion des itérations.

L'utilisation des Collections exige l'itération faite par l'utilisateur (par exemple on utilisant foreach), ceci est appelé *itération externe*. Contrairement, la Streams API utilise l'itération interne, c'est-à-dire elle fait les itérations automatiquement et stocke les résultats quelque part. On donne seulement une fonction pour décrire ce qu'il faut faire.

Voici un exemple d'une itération externe :

```
List<String> names = new ArrayList<>();
for(Dish d : menu){  //itération du menu
        names.add(d.getName());  //extraction de nom et l'ajouter au liste.
}
```

Avec les Streams, on peut simplement faire comme ceci :

```
List<String> names = menu.stream()
.map(Dish::getName)                                    //donne la méthode dans les paramètres de map() pour extraire les noms.
                                        .collect(toList());//collection des données…Sans itération !
```

On utilisant les itérations internes, le traitement des objets peut être effectué de manière transparente en parallèle ou dans un ordre diffèrent qui peut être plus optimisé. Ces optimisations sont difficiles si on utilise les itérations externes

comme on est habitué à faire, et c'est ça la raison pour laquelle les itérations internes sont préférables.

Les opérations des Streams

L'interface java.util.stream.Stream définit plusieurs opérations, ils peuvent être classés en deux catégories :

- Des opérations **intermédiaires** : comme **filter()**, **map()**, **limit()**…
- Des opérations **terminales** : comme **collect()**.

Les opérations intermédiaires

Les opérations intermédiaires comme **filter** ou **map**renvoient un autre stream comme type de retour. Cela permet de connecter les opérations.

Les opérations terminales

Les opérations terminales permettent de produire un résultat de n'importe quel type qui n'est pas un Stream (Int, List, void…etc), par réunir tous les éléments en utilisant un Collector.

Les opérations en détail

Les Streamspermettent de passer des itérations externes à des itérations internes. Au lieu d'écrire un code ou on a besoin de gérer les itérations, on peut utiliser les streams API qui supporte les opérations de filtrage, et qui gère les itérations sur les collections pour vous.

Cette façon de travailler avec les données est très utile parce qu'on laisse la gestion des données pour les Streams API. Par conséquence, l'API peut travailler sur plusieurs optimisations en coulisses, et peut aussi décider d'exécuter le code en parallèle.

Pour cela l'API Streams utilise des méthodesqui permet de faire des opérations comme :

- ***Filtrage***

On peut sélectionner des éléments d'un Stream on utilisant la méthode ***filter()*** qui prend un prédicat comme argument et renvoie un Stream qui contient les éléments qui correspondent au prédicat :

```
List<String>vegetarianMenu = menu.stream()
.filter(Dish::isVegetarian)                    //la référence de méthode vérifie si un plat est végétarien
    .collect(toList());
```

Pour filtrer les éléments unique on utilise la méthode **distinct()** qui renvoie un Stream avec les éléments unique seulement :

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

On peut aussi limiter ou bloquer les nombres d'éléments d'un Stream avec la méthode **limit()** qui prend la taille comme argument, par exemple si on veut les trois premiers plats avec des calories **>300 :**

```
List<Dish> dishes = menu.stream()
.filter(d ->d.getCalories() > 300)
.limit(3)
                                    .collect(toList());
//notez que la méthode  limit() fonctionne aussi sur les Streams non triés
```

On peut aussi utiliser la méthode **skip(N)** pour retourner un Stream après rejeter les N premiers éléments. Si le Stream a moins d'éléments que N, un Stream vide est retourné :

```
List<Dish> dishes = menu.stream()
.filter(d ->d.getCalories() > 300)
.skip(2)
 .collect(toList());
//les méthodeslimit() et skip() sont complémentaires
```

▪ *Mapping*

- **map()**

La méthode **map()** prend une fonction comme argument, la fonction est appliqué au éléments de Stream.

Par exemple dans l'exemple suivant on passe une fonction qui extrait les noms des plats :

```
List<String>dishNames = menu.stream()
.map(Dish::getName)
.collect(toList());
/* la méthode getName renvoie un String, alors map va renvoyer un Stream de type Stream<String>. */
```

Un autre exemple est de retourner une liste des caractères unique d'une liste des mots, si on prend la liste **["Hello", "World"]** en renvoie la liste **["H", "e", "l", "o", "W", "r", "d"].**

on peut essayer avec une **map** suivi par la fonction **distinct** :

```
words.stream() .map(word ->word.split(""))

.distinct() .collect(toList());
```

cela ne fonctionnera pas parce que la méthode map renvoie un String[] pour chaque mot du Stream, alors le Stream retourné par map est de type **Stream<String[]>**, mais on veut un **Stream<String>** qui représente un Stream des caractères.

On a besoin d'un Stream des caractères au lieu des tableaux. Pour résoudre ce problème on utilise la méthode **Arrays.stream()** qui prend un tableau et produit un Stream des caractères :

```
words.stream() .map(word ->word.split(""))

.map(Arrays::stream)

.distinct() .collect(toList());
```

Encore, cette solution ne fonctionnera pas ! C'est parce que maintenant la deuxième map va retourner une liste des Stream **Stream<Stream<String>>.**

- **flatMap()**

Heureusement, l'API Streams contient une méthode pour résoudre ce problème, **flatMap()** : cette méthodeapplique sa fonction au contenu du Stream et non pas le Stream lui-même. C'est-à-dire flatMap remplace chaque élément d'un Stream avec un autre Stream et concatène après tous les Stream dans un seul Stream. On peut dire que la méthode permet de mettre à plat un Stream :

```
words.stream() .map(word ->word.split(""))
                    .flatMap(Arrays::stream)
.distinct() .collect(toList());
```

- ***Finding and matching***

- **anyMatch()**

Une autre façon de traitement des données est de trouver si un des éléments dans un ensemble de données correspondent à un prédicat donné. La méthode **anyMatch()** est une opération terminale qui retourne un booléen :

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

- **allMatch()**

Pour tester si un prédicat vérifie tous les éléments d'un Stream on utilise la méthode **allMatch()** :

```
booleanisHealthy = menu.stream() .allMatch(d ->d.getCalories() < 1000);
```

- **nonMatch()**

le contraire de **allMatch()** est…**nonMatch()** ! Elle vérifie si aucun élément du Stream ne correspond au prédicat donné. Par exemple, on peut refaire l'exemple précédantcommececi :

```
booleanisHealthy = menu.stream() .noneMatch(d ->d.getCalories() >= 1000);
```

- **findAny()**

La méthode **findAny()**sert à retourner un élément arbitraire de type**Optional** (voir le chapitre du classe Optional) .Par exemple :

```
Optional<Dish> dish =menu.stream() .filter(Dish::isVegetarian) .findAny();
```

La méthode va optimiser le Stream en terminant des que le résultat est trouvé.

- **findFirst()**

Une autre méthode est **findFirst()**qui sert à retourner le premier élément rencontré. Par exemple, dans l'exemple suivant on donne une liste des nombres pour trouver le premier carrée divisée par 3 :

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

Optional<Integer>firstSquareDivisibleByThree =

someNumbers.stream()

.map(x -> x * x)

.filter(x -> x % 3 == 0) .findFirst(); // 9
```

Reducing

On peut combiner les éléments d'un Stream pour exprimer des requêtes plus compliqué comme « le calcul du somme des calories dans un menu » on utilisant l'opération **reduce().** Cetterequêtecombine tous les éléments du Stream pour produire une seule valeur Int.

La méthode **reduce()** est une opération terminale qui réduit les éléments du Stream avec un **BinaryOperator**.

Par exemple, si on veut la somme des nombres d'une liste :

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

Il y a deux paramètres dans ce code :

- *une valeur initiale : 0*
- *un **BinaryOperator**<T>pour combiner deux éléments et produire une nouvelle valeur. Ici on utilise une expression Lambda.*

On peut facilement modifier ce code pour appliquer une multiplication pour tous les éléments du Stream on utilisant l'expression Lambda **(a, b) -> a * b :**

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

On a aussi l'option d'utiliser une variante de **reduce** qui ne prend pas une valeur initiale, mais elle renvoie un objet **Optional** :

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

Mais pourquoi renvoyer un **Optional<Integer>** ? Si on considèrele cas quand le Stream ne contient pas des éléments. La méthode **reduce** ne peut pas retourner une somme parce qu'il ne dispose pas d'une valeur initiale. Ceci est la raison pour laquelle le résultat est retourné dans un objet **Optional** pour indique que la somme peut être absent.

L'opération**reduce** nous permet aussi de trouver le max/min d'un stream ! On a besoin d'une expression Lambda qui prend deux éléments et renvoie le maximum/minimum. La méthode **reduce** va utiliser la nouvelle valeur avec la va valeur prochaine jusqu'à tous les éléments sont consommés :

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```

```
Optional<Integer> max = numbers.stream().reduce(Integer::min);//pour le min
```

On peut utiliser une expression lambda `(x,y)->x<y?x:y`au lieu de référence **Integer::min,** mais le dernier est plus facile à lire.

Collection des données avec les streams

Les Collectors sont utilisés comme un paramètre de la méthode **collect**pour combiner le résultat de la transformation sur les éléments d'un Stream. Le Collector applique une fonction qui transforme les élémentsde Stream, et accumule le résultat dans une structure de données qui contient le résultat final de ce processus. Par exemple :

```
List<Transaction> transactions = transactionStream.collect(Collectors.toList());
```

Les Collectors utilisent des méthodes statiques pour effectuer des fonctionnalités comme réduire, regrouper, ou partitionner les éléments…etc.

- ***Réduire***

On peut utiliser les Collectors pour combiner tous les éléments d'un Stream dans un seul résultat.

- **Maximum & Minimum :**

Supposant on veut trouver le plat avec la valeur maximum des calories, on peut utiliser deux Collectors : **Collectors.maxBy** et **Collectors.minBy**, pour calculer le maximum ou le minimum des valeurs d'un Stream. Ces collectors prennent un Comparator comme argument pour comparer les éléments de Stream :

```
Comparator<Dish>dishCaloriesComparator =
Comparator.comparingInt(Dish::getCalories);

Optional<Dish>mostCalorieDish =
menu.stream().collect(maxBy(dishCaloriesComparator));
```

On a utilisé le type **Optional<Dish>**parce qu'on a considéré le cas où le menu est vide. Avec la classe **Optional,** on peut représenter l'idée qu'on peut avoir aucun élément dans le Stream.

- **Somme et Moyenne :**

On peut utiliser la méthode statique **Collectors.summingInt**qui prend une fonction qui transforme un élément à un objet int, puis renvoie un Collector qui effectue l'addition après être passé au méthode collect() :

```
inttotalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

Les méthodes **Collectors.summingLong**et **Collectors.summingDouble**font la même chose pour l'addition des valeurs Long ou Double.
Pour la moyenne on utilise la méthode **Collectors.averaginInt, Collectors.averaginLong**et **Collectors.averaginDouble** :

```
doubleavgCalories =
menu.stream().collect(averagingInt(Dish::getCalories));
```

Il y a aussi une méthode qui peut retourner tous ces résultats avec une seule opération : **Collectors.summarizingInt.**
Par exemple, on peut calculer le nombre des plat, obtenir le somme, la moyenne, le maximum et la minimum des calories de chaque plat avec une seul opération :

```
IntSummaryStatisticsmenuStatistics =
menu.stream().collect(summarizingInt(Dish::getCalories));
```

Ce collector regroupe toutes ces informations dans une classe **IntSummaryStatistics**qui utilise des méthodes getter pour accéder les résultats.

- **Joindre les Strings :**

La méthode **Collectors.joining()** invoque la méthode toString de chaque élément et retourne une concaténation des éléments du Stream.

```
String shortMenu = menu.stream().collect(joining());
```

Le variable **ShortMenu** contient le String:
porkbeefchickenfrenchfriesriceseasonfruitpizzaprawnssalmon,

qui est pas très lisible, heureusement, la méthode joining a une version qui accepte une chaine de caractères entre deux éléments consécutives, par exemple :

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

Maintenant le variable **ShortMenu** contient le String:
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon.

- **La méthode reducing() :**

Toutes les méthodes qu'on a vues sont des cas spécifiques du processus de réduction qui utilise la méthode statique **collectors.reducing().** Par exemple on peut calculer le total des calories dans un menu avec un collector créé par reducing :

```
inttotalCalories = menu.stream()
                .collect(reducing(0, Dish::getCalories, (i, j) -> i + j));
```

La méthodereducing prend 3 arguments :

- La valeur initiale et la valeur retournée si le Stream est vide.
- La méthode qui transforme les éléments en un entier.
- Une méthode qui retourne la somme de deux valeurs.

Similairement, on peut utiliser une méthode reducing() qui prend un seul argument :

```
Optional<Dish>mostCalorieDish =
        menu.stream()
.collect(reducing((d1, d2) -> d1.getCalories()> d2.getCalories() ? d1 : d2))
```

- ***Regrouper***

  On peut utiliser la méthode statique**Collectors.groupingBy** pour regrouper des éléments dans des groupes, basés sur un ou plusieurs attributs.

  ```
  Map<Dish.Type, List<Dish>>dishesByType =
  menu.stream().collect(groupingBy(Dish::getType));
  ```

  Le résultat va être :
  {FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza], MEAT=[pork, beef, chicken]}

  Si on veut regrouper les éléments par des conditions plus compliqués au lieu d'un attribut simple, par exemple des groupes qui dépend au nombre des calories, on peut exprimer ce logique avec les expressions Lambda :

  ```
  publicenumCaloricLevel { DIET, NORMAL, FAT }

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

On peut aussi utiliser les deux façons pour regrouper les éléments du Stream avec une version de la méthode **groupingBy** qui prend un deuxième argument de type Collector. Alors pour effectuer un regroupement de deux niveaux, on passe une **groupingBy interne** au **groupingBy externe** :

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

Le résultat de ce regroupement est :

{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
FISH={DIET=[prawns], NORMAL=[salmon]},
OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}

Cette opération du multi-niveau regroupement peut être prolongée au plusieurs niveaux, et un n-niveau regroupement a un résultat de n-niveau Map qui contient n-niveau des structures.

Le deuxième argument de la méthode **groupingBy**accepte n'importe quelle Collector, et pas juste un autre **groupingBy,** par exemple, on peut

compter le nombre des plat de chaque type dans le menu, on passant **counting** comme and deuxième argument de collector**groupingBy** :

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

Le résultat de cette opération :
{MEAT=3, FISH=2, OTHER=4}

Remarque : la méthode**groupingBy(f)** ou f est la fonction de classification, est en réalité une manière raccourcie d'écrire**groupingBy(f,toList())**.

- ***Partitionner***

Le partitionnement est un cas spécial de regroupement, on utilisant un prédicat comme une fonction de classification qui retourne un booléen, c'est-à-dire le résultat va être une Map qui contient un booléen comme clé, donc on va avoir au maximum deux groupes : une pour true et l'autre pour false.

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

Le résultat est :

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

Après si on veut accéder ces valeurs on indique la clé dans la méthode get du Map :

```
List<Dish>vegetarianDishes = partitionedMenu.get(true);
```

On peut avoir le même résultat si on utilise la méthode filter :

```
List<Dish>vegetarianDishes =
menu.stream().filter(Dish::isVegetarian).collect(toList());
```

Mais c'est préféré d'utiliser la méthode de partitionnement parce que çadonne l'avantage de garder les deux listes des éléments de Stream.

On peut aussi passer un deuxième argument de type Collector, par exemple :

```
Map<Boolean, Map<Dish.Type, List<Dish>>>vegetarianDishesByType=
menu.stream().collect(
partitioningBy(Dish::isVegetarian,
groupingBy(Dish::getType)));
```

Cette opération va générer une map de deux niveaux :

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

L’interface Collector

L’interface Collector se compose d’un ensemble de méthodes qui fournissentun plan pour savoircomment implémenter cette interface. Chaque méthode Collector implémente les méthodes déclarées dans l’interface Collector :

```
public interface Collector<T, A, R> {
Supplier<A>supplier();
BiConsumer<A, T>accumulator();
Function<A, R>finisher();
BinaryOperator<A>combiner();
Set<Characteristics>characteristics();
}
```

Chaque Collector accepte les éléments de type T et produit des valeurs de type R (par exemple R = List<T>). Le type générique <A> définit le type intermédiaire qu’on va utiliser pour accumuler les éléments de type T :

La méthode supplier :

Cette méthode retourne une fonction qui crée une instance d’un accumulateur qu’on va utiliser dans le processus de collection des éléments de type T.

La méthode accumulator :

Cette méthode retourne une fonction qui prend deux arguments et effectue une opération de réduction. Le premier argument est l’accumulateur et le deuxième est le *n*ième élément traversé du Stream. La fonction retourne *void* parce que l’accumulateur est modifié dans la fonction.

La méthode combiner :

Cette méthode est utilisée pour relier deux accumulateurs en un seul. On utilise le combiner si le Collector est exécuté en parallèle.

La méthode finisher :

Cette méthode prend l'accumulateur de type A et il se transforme en une valeur de résultat final de type R.

La méthode characteristics :

La méthode characteristics, retourne un ensemble des caractéristiques définissant le comportement du Collector.Characteristics est une énumération contenant trois éléments :

- UNORDERED :
  Le résultat de réduction n'est pas affecté par l'ordre dans lequel les éléments du Stream sont traversés et accumulés.

- CONCURRENT :

  La fonction de l'accumulateur peut être appelée simultanément à partir de plusieurs threads, et ce Collector peut effectuer une réduction en parallèle.

- IDENTITY_FINISH :
  Cela indique que la fonction retournée par la méthode finisher est la fonction d'identité. Dans ce cas l'objet accumulateur est utilisé comme un résultat final de la réduction.

Exemple de ToListCollector :

On peut implémenter une classe ToListCollector<T>qui rassemble tous les éléments d'un Stream dans une Liste :

```
Import java.util.* ;
Import java.util.function.* ;
Import java.util.stream.Collector ;
Import static java.util.stream.Collector.Characteristics.* ;

public class ToListCollector<T> implements Collector<T, List<T>, List<T>>{

    public Supplier<List<T>>supplier() {
        returnArrayList::new;

    }

    public BiConsumer<List<T>, T>accumulator() {
        return List::add;
    }

    public Function<List<T>, List<T>> finisher() {
        returnFunction.identity();
    }

    publicBinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
        list1.addAll(list2);
        return list1; }
    }

    publicSet<Characteristics> characteristics() {
        returnCollections.unmodifiableSet(
EnumSet.of(IDENTITY_FINISH, CONCURRENT));
    }}
```

Méthodes par default

Normalement, une interface java contient un ensemble des méthodes, et chaque classe qui implémente cette interface doit donner une implémentation de toutes ses méthodes.
Java 8 Introduit un nouveau concept d'implémentation des méthodes « par default » pour les interfaces.Ce nouveau mécanisme nous permet d'ajouter des implémentations par default au interfaces sans être obligé de donner une implémentation dans la classe à chaque fois.

Par exemple, l'interface 'List' n'a pas la déclaration du méthode 'sort',alors si on l'ajoute on va perturber le fonctionnement de toutes les implémentations de Framework List.
La méthode sort par exemple est nouvelle dans Java 8 et définit comme ceci :

```
default void sort(Comparator<? super E> c){
Collections.sort(this, c);
  }
```

Syntax

Les méthodes par default commence par le mot '**default**' et contient un corps comme une méthode déclaré dans un classe.

Exemple :

```
public interface A {
default void print(){
      System.out.println("this is A!");
   }
}

public interface B {
default void print(){
      System.out.println("this is B!");
}}
```

Problème :
Avec les méthodes par default, il y a la possibilité d'avoir une classe implémente 2 interfaces avec la même méthode par default.
On a deux solutions pour ce problème :

- Première solution consiste à créer une méthode proprequi remplace l'implémentation par défaut :

```
publicclassCimplementsA,B{
defaultvoidprint(){
System.out.println("this is C !");
}
}
```

- La deuxième solution c'est d'appeler la méthode par default d'interface spécifiée on utilisant **super :**

```
publicclassCimplementsA,B{
defaultvoidprint(){
A.super.print();
}
}
```

Remarque: Dans l'exemple si l'interface B hérite A, la classe C va implémenter la méthode de l'interface B.

Static Default methods

Avec Java 8 on peut aussi avoir des méthodesstatiques par default :

```
publicinterfaceA{
defaultvoidprint(){
System.out.println("this is A!");
}

staticvoidtest(){
System.out.println("test test!!");
}
}
```

Classe Optional

Introduction

Java 8 introduit une nouvelle classe appelée **Optional**(java.util.Optional<T>).

Optional est un conteneur objet qui est utilisé pour représenter une valeur qui peut être nulle, on utilisant différentes méthodes qui facilite la gestion des valeurs comme *« disponible »* ou *« non disponible »*au lieu de vérifier des valeurs NULL.

Par exemple : prenant les classes *Person*,*Car* et *Insurance* comme un modèle :

```
public class Person {
private Car car;
public Car getCar() { return car; }
}

public class Car {
        private Insurance insurance;
        public Insurance getInsurance() { return insurance; }
}

public class Insurance {
        private String name;
        public String getName() { return name; }
}
```

Si vous connaissez une personne peut ou non avoir une voiture, la variable ‘car’ à l’intérieur de la classe ne doit pas êtredéclarée de type voiture (et affecté à une référencenulle lorsque la personne ne possède pas une voiture) mais doit plutôt être de type Optional<car> :

```
public class Person {
privateOptional<Car> car;
publicOptional<Car> getCar() { return car; }
}//une personne peut ou non avoir une voiture

public class Car {
        privateOptional<Insurance>insurance;
        public Optional<Insurance>getInsurance() { return insurance; }
} //une voiture peut ou non avoir une assurance

public class Insurance {
        private String name;
        public String getName() { return name; }
}//mais une compagnie d’assurance doit avoir un nom.
```

Remarque que utilisation du classe Optional a amélioré ce modèle. Le fait qu’un personne fait référence à uneOptional<Car> nous donne une idée à la disponibilité d’objet ‘Car’.

De la même façon, le fait que le nom de la compagnie est déclaré de type String rend évident que c’est obligatoire pour une instance compagnie d’avoir un nom.

Ainsi, nous pouvons prédire et éviter les exceptions ;vous ne devez pas ajouter un test de nulle pour le nom d’compagnie parce que ça sert à cacher le problème au

lieu de le fixer. Une compagnie doit avoir un nom, donc si vous trouvez un sans nom, c'est-à-dire qu'il y a un problème dans les données et pas dans le code.

Il est important de noter que l'intention de la classe Optional est de vous aider à créer des APIs plus compréhensibles pour les utilisateurs et n'est pas de remplacer toutes les références nulles.

Les méthodes de classe Optional

La première étape avant de travailler avec Optional est d'apprendre comment créer des objets Optional, il y a plusieurs façons :

- **Static <T> Optional <T>empty() :** Empty Optional
  On peut utiliser la méthode statique Optional.empty() pour un objet Optional vide.

  ```
  Optional<Car>optCar = Optional.empty();
  ```

- **Static<T>Optional<T>of(T value) :** Optional d'une une valeur non nulle
  On peut aussi créer un Optional d'après une valeur non nulle avec la méthode statique Optional.of() :

  ```
  Optional<Car>optCar = Optional.of(car);
  ```

- **Static<T>Optional<T>ofNullable(T value) :**Optional d'une valeur nulle
  Finalement, avec la méthode statique Optional.ofNullable(), on peut créer un Optional qui peut possède une valeur nulle :

  ```
  Optional<Car>optCar = Optional.ofNullable(car) ;
  ```

  Si 'car' est nulle, 'optCar' sera un objet Optional vide.

Pour lire les valeurs retournées par des objets Optional on utilise les méthodes suivant :

- **Get()** : si la valeur est présente il la renvoie, sinon il jette l'exception *NoSuchElementException.* Et cela pose le même problème que nous avions avant l'utilisation de classe Optional.

- **T orElse(T other)** : cette méthode permet de donner une valeur par default si l'objet Optional ne contient pas une valeur.

- **T orElseGet(Supplier<? extends T>other)** :retourne la valeur si présent, sinon retourne la valeur 'other' donné par les paramètres.

- **<? extendsthrowable> T orElseThrow(Supplier<? extends X>exceptionSupplier):**retourne le contenu si il est présent, sinon lance l'exception crée par le 'Supplier' donné dans les paramètres.

- **VoidifPresent(Consumer<? super T>consumer):** Si la valeur est présent,invoque le 'Consumer' spécifié avec la valeur, sinon ne fait rien.

- **BooleanisPresent() :** renvoie **true**si la valeur est présente, sinon **false**.

Extraction et transformation des valeurs les objets Optional :

Pour les exemples qu'on va voir on utilise le modèle présenté au début du chapitre.

- **<U>Optional<U>map(Function<? super T, ? extends U>mapper)** :
  Normalement, si on veut extraire le nom de la compagnie par exemple, en utilise un code comme :

```
String name = null;
if(insurance != null){
        name = insurance.getName(); }
```

  la classe Optionaldonne une méthode **map**, similaire à la méthode de Stream qu'on a vu, cette méthode applique la fonction passé on argument au contenu de l'objet Optional, si l'objet est vide, rien ne se passe.

  Appliquant la méthode map dans l'exemple précédant :

```
Optional<String> name = insurance.map(Insurance::getName);
```

  Maintenant si on veut utiliser cette méthode pour écrire un code des méthodes chainées comme ceci :

```
public String getCarInsuranceName(Person person) {
        returnperson.getCar().getInsurance().getName();
}
```

  La première idée que vous aurez est d'utiliser map :

```
Optional<Person>optPerson = Optional.of(person);
Optional<String> name =optPerson.map(Person::getCar)
                                .map(Car::getInsurance)
                                .map(Insurance::getName)
```

Mais ce code ne compile pas. Pourquoi ? la variable optPerson est de type Optional<Person>, alors c'est normal d'utiliser la méthode map. Mais getCar renvoie un objet Optional<car>, c'est-à-dire le résultat de la méthode map va renvoie un objet du type Optional<Optional<car>>. Par conséquent, l'appel de la méthode getInsurance est invalide parce que l'objet Optional le plus externe contient comme sa valeur un autre objet de type Optional, qui ne supporte pas la méthode getInsurance.

Pour résoudre ce problème, on utilise une autre méthode inspiré par les méthodes de streams : flatMap.

- **<U>Optional<U>flatMap(Function<? super T , Optional<U>>mapper)** :

  La méthode flatMap prend une fonction comme argument et l'applique au contenu d'objet Optional, et renvoie un Optional, ce qui donne un Optional d'Optional. Mais l'effet de flatMap remplace chaque objet Optional par leur contenu, par conséquent, le résultat final sera un seul objet Optional.

  Retournant au code précédant, on peut le modifier comme ceci :

```
Optional<Person>optPerson = Optional.of(person);
Optional<String> name =optPerson.flatMap(Person::getCar)
                                .flatMap(Car::getInsurance)
                                .map(Insurance::getName)
.orElse("Unknown");
```

  Note qu'on a utilisé la méthode orElse() pour renvoyer une valeur donner dans le cas où le résultat du map() est vide.

- **Optional<T>filter(Predicate<? super T>predicate)** :
  Normalement si on veut vérifier le nom de l'assurance par exemple, il faut passer par vérifier la présence du valeur et après tester l'égalité avec equals() :

```
Insurance insurance = ...;
if(insurance != null && "CambridgeInsurance".equals(insurance.getName()))
        {  System.out.println("ok");  }
```

  On peut modifier cet exemple on utilisant la méthode filter() :

```
optInsurance = optInsurance
                .filter(insurance -> "CambridgeInsurance".equals(insurance.getName()));
```

  La méthodefilter() prend un prédicat comme un argument.si la valeur présente dans l'objet Optional correspond au prédicat, la méthodefilter renvoie la valeur ; sinon il renvoie un objet Optional vide.

  Cette méthode est similaire à la méthode filter des streams. Si l'objet Optional est vide, la méthode n'a aucun effet, sinon elle applique le prédicat à la valeur contenue dans l'Optional. Si cette application renvoie **true**, l'objet Optional est retourné sans changement, sinon la valeur est filtrée et l'objet Optional sera vide.