**Interface fonctionnelle prédéfinit**

Tout d’abord c’est quoi une interface fonctionnelle :

C’est une interface qui contient une seule méthode á redéfinir, cette condition nous a permettre d’utiliser l’expressions lambda, qui est un raccourci de la création d’une implémentation de l’interface fonctionnelle

Prenons l’exemple d’interface Logger qui a une seule méthode log qui prend un paramètre de type String, et essayons de l’implémenter avec les différentes manières possibles :

**interface** Logger {

**void** log(String msg);

}

**class** LoggerImp **implements** Logger{

@Override

**public** **void** log(String msg) {

System.***out***.println(msg);

}

}

**void** exp1() {

Logger logger = **null**;

//En utilison la class LoggerImp

logger = **new** LoggerImp();

logger.log("En utilsation Implementation LoggerImp");

//En utilisaon class anonyme

logger = **new** Logger() {

@Override

**public** **void** log(String msg) {

JOptionPane.*showMessageDialog*(**null**, msg);

}

};

logger.log("En utilsation Implementation anonyme");

//En utilisant expression lamba

logger = (msg) -> System.***out***.println(msg);

logger.log("En utilsation Expression lambda");

//En utilisant méthode de référence, cette méthode

doit être avec la même signature que la méthode log

de l’interface

logger = System.***out***::println;

logger.log("En utilisant méthode de reférence");

}

Expression lamba : équivalent á une implémentation d’une interface avec une classe anonyme, sauf qu’elle très bref.

Si la méthode á redéfinir contient un seul paramètre et une seule instruction qui le retour de la fin : (parm1) -> {return para1 + 1;} ou bien

Param1 -> para1 + 1 ;

Les parenthèses ne sont pas obligatoires si la méthode contient un seul paramètre d’entrée.

Les accolades et le return sont obligatoires si la méthode se compose de 2 ou plusieurs instructions.

**A-Function** : Function<T,T> première type générique c’est un paramètre, 2ème c’est le type de retour.

Exemple d’utilisation : On veut par exemple utiliser l’interface Function pour calculer la somme de 1 jusqu’à un paramètre entrée n ;

En utilisant une classe interne :

**private** **void** exp01() {

/\*Instanciation on utilisant la classe interne créé qui prend un entier

\* comme paramétre et retourne un entier \*/

Function<Integer, Integer> f = **new** FuncImp();

**int** n = 3;

**int** somme = f.apply(n);

System.***out***.println("somme de "+n+" = "+somme);

}

/\*Pour utiliser l’interface Functionnelle Function il faut implémenter

* La méthode apply() et l’adapter á votre besoin\*/

**class** FuncImp **implements** Function<Integer, Integer>{

@Override

**public** Integer apply(Integer n) {

**if**(n == 0) **return** 0;

**else** **return** apply(n-1) + n;

}

}

En utilisant une expression lambda :

**void** exp02() {

Function<Integer, Integer> f = (n) -> {

**int** som = 0;

**for**(**int** i = 0;i <= n;i++) som += i;

**return** som;

};

**int** n = 5;

System.***out***.println("somme de "+ n + " = "+f.apply(n));

}

En utilisant un méthode référence : (une méthode qui a la même signature que la méthode de l’interface fonctionnelle).

**void** exp03() {

//En utilisant methode static

Function<Integer, Integer> f = Examples::*somme*;

**int** n = 5;

System.***out***.println("somme de " + n + " = " + f.apply(n));

//En utilisant une méthode normale, ce qui implique de passe par une instance

Function<Integer, Integer> f2 = **new** Examples()::somme2;

**int** n2 = 5;

System.***out***.println("somme de " + n2 + " = " + f2.apply(n2));

}

**public** **static** **int** somme(**int** n) {

**return** (n == 0) ? 0 : *somme*(n - 1) + n;

}

**public** **int** somme2(**int** n) {

**return** (n == 0) ? 0 : *somme2*(n - 1) + n;

}

**2-BiFunction :** est une interface fonctionnelle qui prend 3 paramètres génériques, 2 en entrée et un en sortie (La méthode á redéfinir c’est apply() ).

Exemple d’utilisation : essayer de concaténer 2 chaines en utilisant BFunction

**void** exp04() {

BiFunction<String, String, String> bF = **null**;

String s1 = "chaine 1";

String s2 = "chaine 2";

String res = "";

//En utilisant expressions lambda

bF = (a, b) -> a+ " " +b;

res = bF.apply(s1,s2);

System.***out***.println("resultat de concatenation en utilisant lambda: "+res);

//En utilisant méthode de référence, comme première paramètre objet appelant

bF = String::concat;

res = bF.apply(s1, " "+ s2);

System.***out***.println("resultat de concatenation en utilisant methode referecence: "+res);

}

**3-Predicate :** accepte un paramètre générique et retourne un booléen (la méthode á redéfinir et s’appelle test)

Exemple d’utilisation : tester par exemple c’est une chaine comporte 32 caractères.

**void** exp06() {

Predicate<String> isValid = **null**;

//En utilisant expression lambda

isValid = (s) -> s.length() <= 32;

System.***out***.println(isValid.test("45"));

}

Il y’a aussi BiPredicate qui accepte deux paramètres en entrée et retourne un booléen.

**4-BinaryOperator** : équivalent á BiFunction mais avec un paramètre générique qui regroupe les 2 paramètre d’entrée est celle de la sortie.

Exemple d’utilisation : Création d’un operateur plus, moins, multiplication, division … (la méthode á redéfinir c’est apply())

**void** exp07() {

BinaryOperator<Integer> plus = **null**;

//En utilisant expression lambda

plus = (a, b) -> a + b;

System.***out***.println(plus.apply(4, 9));

}

**5-Supplier :** accepte un seul type générique, qui est retourner par son méthode get()

Exemple d’utilisation : Créer un supplier qui permet de générer des entiers aléatoirement :

**void** exp08() {

Supplier<Integer> s = **null**;

//En utilisant une expression lambda

s = () -> (**int**)(Math.*random*()\*100);

**int** res = s.get();

System.***out***.println("Nombre aléatoire entre 0 et 100: "+res);

}

**6-Consumer :** accepte un seul type générique mais comme paramètre d’entre de la méthode accepte(), et il n’a pas un type de retour et c’est exactement pourquoi il est consommateur.

L’API STREAM : C’est une structure de type MONAD qui se base sur des fonctions enchainée, l’entrée de l’une est la sortie de l’autre, se compose des fonctions intermédiaire (tel que map, filter ..) et des fonctions terminal (tel que collect, forEach …)

Remarque : exécution des Stream est retardé et optimisé, si on a pas une fonction terminal les fonctions intermédiaires seront pas exécutés et si on atteint limit de la fonction limit(), le Stream arrête son d’exécution (optimiser)

Pour avoir un objet Stream sur les différentes structures :

* List : il faut appeler la méthode stream() 🡪 list1.stream().
* Tableau : il faut appliquer la méthode sur le tableau 🡪 Arrays.stream(tab).
* File lines : on fait appelle á la méthode Files.lines(Paths.of(‘’laSource’’)) , ou bien d’utiliser la classe BufferedReader qui est instancier sur un FileReader, on utilise la méthode lines() sur cette instance, on arrive á avoir Stream des lignes.
* Stream.of("one", "two", "three") 🡪 retourne directement un Stream<String> steams.

Diffèrent fonctions du Stream :

* filter() accepte comme paramètre un Predicate qui permet d’appliquer un test sur chaque élément du Stream, si l’élément satisfait le test, l’ élément sera retourner avec les autres élément du Stream qui satisfissent aussi le test Predicate.
* map() accepte comme paramètre un Function, qui pour chaque élément du Stream on opère sur lui une instruction pour retourner d’autre chose (cette chose peut être un nouveau type , ou même type) sert aussi pour convertissions.
* collect() permet de transférer un Stream vers une collection, il utilise comme paramètre un collector,(Collectors prédéfinit : Collectors.toList(), Collectors.toMap() …)
* sorted() permet de trier un Stream, si le type passer dans le Stream implémente l’interface comparable, pas besoin de passer un paramètre, sinon il faut passer un paramètre de type comparator (Compartor existant : Comparator.comparing(Class ::getter() 🡪 permet d’appliquer un comparator d’une certaine classe on se basant sur un champ en utilisant un getter)
* Stream.generate() accepte comme paramètre un supplier, permet de générer un stream d’objet retourner par le supplier.
* Limit() : accepte un entier n , permet de limiter la génération ou le traitement sur un Stream n fois.

Quelque remarque :

Java 8 á introduit les concepts suivants Api Stream et Collector, interface fonctionnelle, lambda expression, méthode de référence, default méthode dans une interface, Objet Optional.

Interface fonction : est une interface qui possède une seul méthode, on distingue 4 Interfaces fonctionnelles introduites en java 8 :

* Consumer<T> :: accept(T t).
* Supplier<T> :: T get().
* Predicat<T> :: boolean test(T t).
* Function<T, R>:: R apply(T t).

Lambda expression : est une implémentation d'une interface fonctionnelle.

***Méthode référence :***

Est une autre façons d'implémenter une interface fonctionnelle, on utilisant une méthode déjà prédéfinie á condition que cette méthode soit avec la même signature que la méthode de l'interface fonctionnelle :

* Consumer<String> c = **System.out::println** 🡪 equivaut á **s -> System.out.println(s).**

On utilisant méthode de référence en gagne en terme de performance et de lisibilité comparant á une expression lamba.

En effet, on a 4 types de méthode référence:

* **Bound Instance** : invocation d'une méthode sur un objet : **System.out :: println**, l'objet c'est System.out, et la méthode c'est println.
* **Static :** il s'agit d'une méthode static, une méthode de la classe: Math::sqrt
* **Unbound Instance**: une méthode qui s'applique sur l'un de paramètres de l'expression lamba, (s1, s2) -> s1.indexOf(s2) , ici index of est appliquer sur 1ère paramétre, donc cette lamba expresion on peut l'ecrire comme suivant 🡪 String::indexOf
* **Constructor**: par exemple EL qui nous retourne un objet de type user 🡪 **name -> new User(name)**, donc celle-ci est équivaut á la méthode référence suivante **User:: new.**

***Api Stream :***

Généralement utilisé pour itérer sur une ensemble de donnée, l'Api Stream est fondé sur le pattern :

Map, Filter et Reduce. Pour comprendre ce pattern on prend l'exemple suivant :

Connaitre la moyenne d'âge d'une liste d'utilisateur qui on plus de 20 ans:

List<User> users :: user 🡪 ages 🡪 ages > 20 🡪 moyenne.

Passer d'une liste d'utilisateur á une liste d'âge s'appelle mapping.

Passer vers une liste des âges supérieur á 20 c'est le filtring.

Calculer la moyenne á partir d'une liste d'âge c'est le reducing.

***1.mapping :***

Le mapping change le type, mais garde l'ordre et le nombre d'objet.

Le mapping est modélisé par : Function<T,R> .

Convertir liste des utilisateur vers des ages:

**Stream<Integer> ages = users.stream().map(u -> u.getAge()).**

***2.Filtrage :***

Le filtrage conserve le type, ne conserve pas le nombre d'objet.

Le Filtrage est modélisé par: Prédicat<T>.

Laisser que les âges supérieur á 20:

**Stream<Integer> agesFiltred = ages.filter(a -> a >= 20).**

***3.Réduction :***

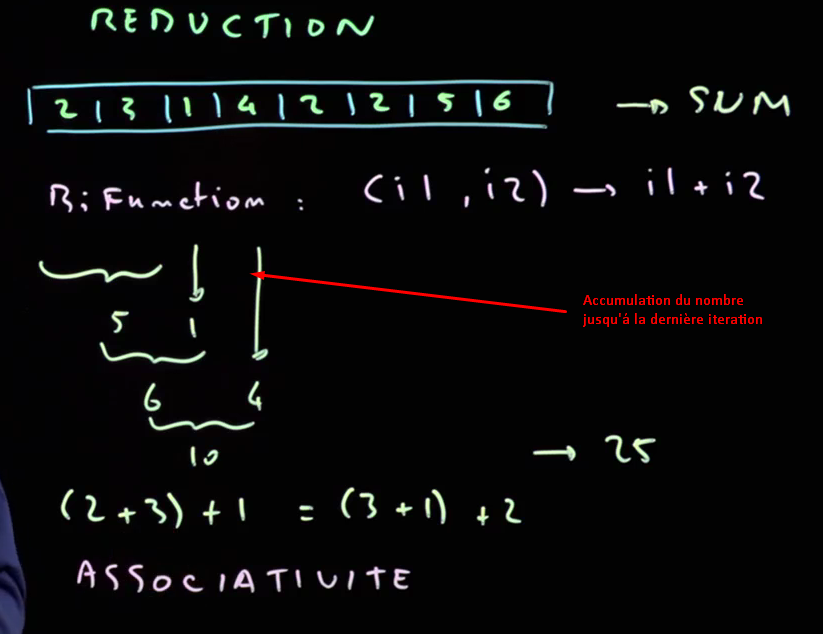
La réduction ressemble á l'agrégation SQL

La réduction est Modélisé par BiFunction<T,T,R>

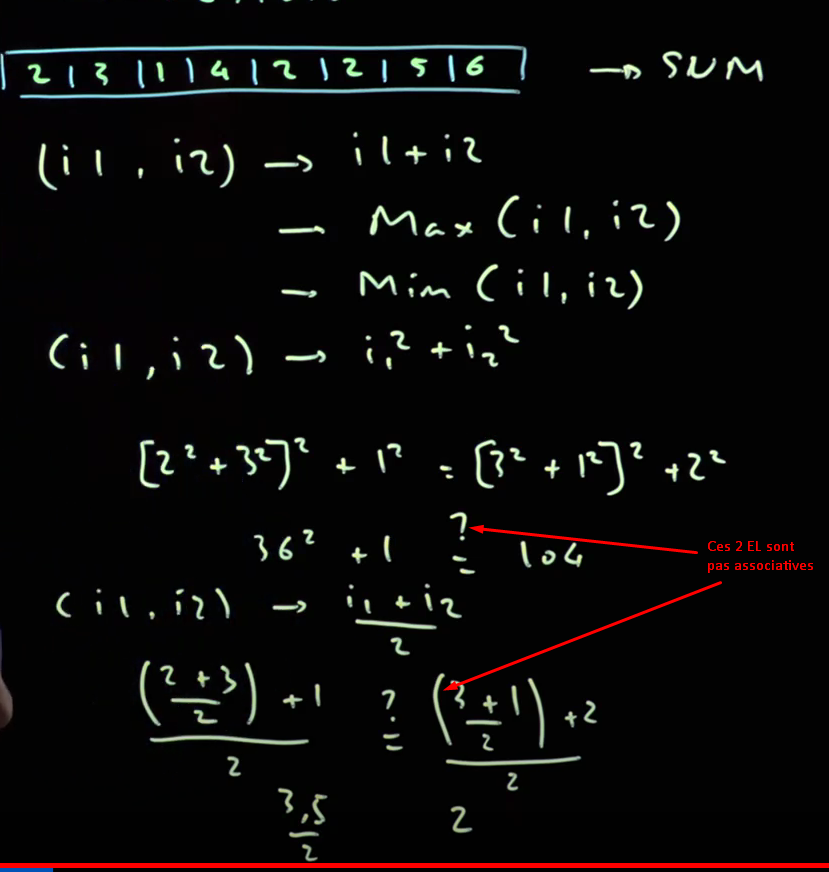
Calculer la moyenne d'age:

**Integer Moyenne = agesFiltred.average();**

Autre exemple : calcule de la somme d'une liste d'entiers.



Il faut que lambda expression du BiFunction soit associative pour que nous aurons un Reduce pertinent.



Stream est un objet vide, map et filter retourne toujours un autre Stream (Opération intermédiaire), Reduce par contre retourne un objet(Opération terminal).

En effet il y'a 2 méthodes reduce():

* Reduce (0, (i1, i2) -> i1 + i2) 🡪 celle-ci utiliser pour calculer la somme, on constante que 0 comme premier paramètre, son intérêt c'est dans le cas où notre liste ne contient aucun élément, ce qui fait la première itération sera i1 + i2 qui sont 2 éléments du Stream qui n'existent pas, par conséquent une exception nullPointerExcpetion sera déclencher. Voila pourquoi c'est mieux de commencé par 0 et comme n'existe pas un élément suivant le 0 sera retourné.
* Reduce(Operateur): utilisé dans le cas de min, max, average, il retourne un objet Optional.

**Opération intermédiaire :** aucun traitement n'est réalisé encore, c'est juste une déclaration d'opération.

* **Map(Function<T,R> t) 🡪 Stream<R>**
* **Filter(Predicat<T> t) 🡪 Stream<T>**
* **Distinct() 🡪 Stream<T>** , ne change pas de type, élémine les doublant.
* **Sorted() 🡪 Stream<T>** , lorsque T implémente l'interface Comparable<T>.
* **Sorted(Comparator<T> cmp) 🡪 Stream<T>,** lorsque T n'implémente pas Comparable<T>.
* **Limit(Long x) 🡪 Stream<T>,** utiliser pour prendre que les x premiers éléments du stream.
* **Skip(Long x) 🡪 Stream<T>,** utiliser pour ignorer les x premiers élements.
* **flatMap(Function<T,R> t) 🡪 Stream<R>,** malgré que le retour soit de type Steram<Stream<R>>, il fusion les stream renvoyé dans un seul Stream.

**Opération terminale (reduce):** exécution de toute les traitements, (ne retourne pas un Stream)

* **forEach(Consumer<T> c)** 🡪 retourne, utiliser pour boucler sur les éléments de Stream.
* **Count()** 🡪 retourne un Long qui est le nombre des éléments du Stream.
* **allMatch(Predicate<T> p)** 🡪 dès qu'un élément de la liste ne respecte pas le prédicat, all match arrête le parcours de la liste, et renvois les éléments précédents.
* **noneMatch(Predicate<T> p)** 🡪 fonctionnement inverse de allMatch().
* **collect(Collectors.toList())** 🡪 retourne une liste List<T>.
* **collect(Collectors.toSet())** 🡪 retourne un Set<T>.
* **collect(Collectors.joining())** 🡪 permet de retourner une chaine de caractères où tous les éléments de Stream son concaténées, **joining(",")** séparation des champs par un virgules, **joining(",", "{", "}")**, 🡪 séparation par "," et chaque élément est entouré par "{ }"
* **toArray( …)** 🡪 permet de retourner un tableau de éléments.
* **min( Comparator<T> cmp)** 🡪 retourne l'élément min on se basant sur un comparator.
* **max( Comparator<T> cmp)** 🡪 retourne l'élément max on se basant sur un comparator.
* **sum()** 🡪 retourne la somme lorsque Stream contient des entiers.

***API Collector :***

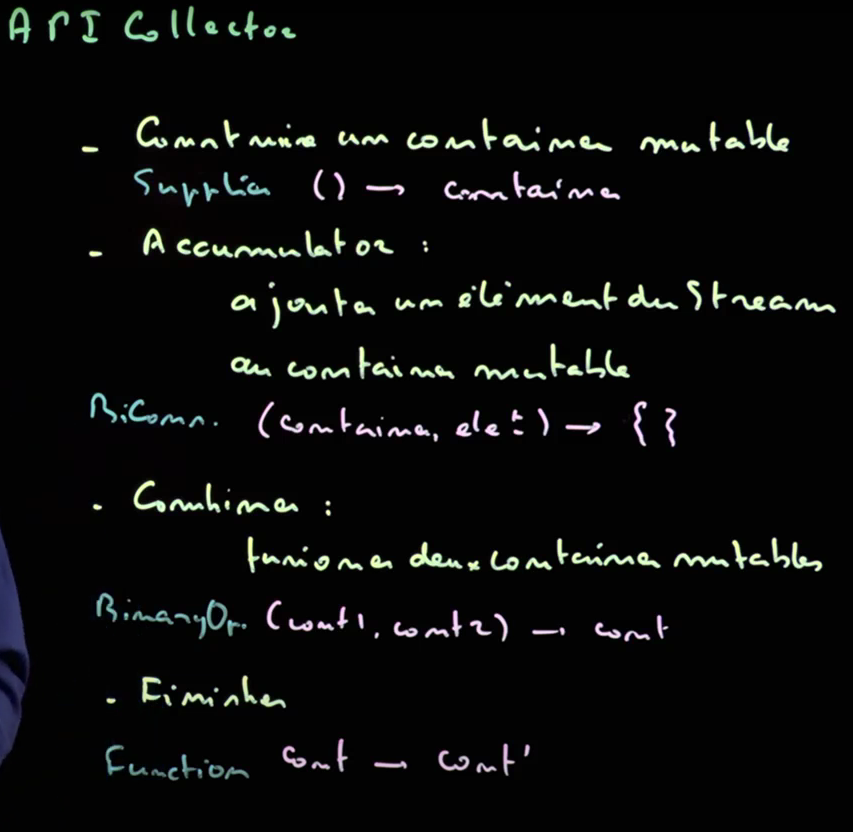
Permet de collecter les éléments d'un Stream par l'invocation de la méthode collect(Collector c) dans une List, un Set ou chaîne de caractères.

🡺 Collectors.toList(), Collectors.toSet() , Collectors.joining().

Autre Collectors :

* Collectors.groupBy(s -> s.length()) 🡪 permet de retourner un HashTable<Integer, List<String>>, comme clé la taille et comme valeur une liste des éléments qui ont cette taille.
* Collectors.groupBY(s -> String::lenght, Collectors.toSet()) 🡪 groupBy peut accepter un 2ième arguement qui détermine la valeur de hashTable, dans cette exemple plutôt d'avoir les valeurs comme liste, on les mets dans un Set.
* toMap(Function<T,R> keyMapping, Function<X,Z> valueMapping) 🡪 keyMapping n'accepte pas les doublants.

***Les 4 types de collectors :***



***Optional :***

Optional c'est un type wrapper, mais un wrapper qui peut être vide, ainsi qu'il ne peut pas contenir une valeur null. Il s'agit d'une classe final qu'on ne peut pas étendre et ses constructeur son privé, par conséquent on peut l'instancier en utilisant new, mais plutôt avec des méthodes factory :

* Optional.empty() : permet de créer un Optional vide.
* Optional.of( …) : permet de faire l'autoboxing de l'objet passé en paramètres, si on passe null comme paramètre, on aura une exception NullPointerException.
* Optional.ofNullable( …): cette méthode permet de créer un objet Optional en utilisant null, mais il s'agit d'un Optional vide, ne contient pas de valeur null.

Les méthodes de test et de récupération de la valeur d'un Optional:

* Boolean isEmpty(): permet de savoir est ce que Optional á une valeur vide.
* Boolean isPresent(): permet de savoir est ce que Optional á une valeur non vide.
* T get(): permet de récupérer la valeur englober par Optional.
* T orElse(T t): dans le cas ou Optional a une valeur vide, l'objet de type t qui sera récupéré.
* T orElseGet(Supplier<T> t): dans le cas où Optional est vide, l'objet de type t qui sera récupéré.
* Exception orElseThrow(): dans le cas où Optional est vide, une exception sera déclancher.