



Java 8



- ▶ Interfaces fonctionnelles
- ▶ Méthodes statiques dans les interfaces
- ▶ Méthodes par défaut (Defender)
- ▶ Lambdas
- ▶ `java.util.function`
- ▶ Références de méthode
- ▶ Stream
- ▶ `java.time`
- ▶ Annotations multiples
- ▶ Nashorn
- ▶ Stamped Locks
- ▶ Autres

Interfaces fonctionnelles

- ▶ Anciennement SAM (Single Abstract Method)
Ne possède qu'une seule méthode abstraite
- ▶ Une interface fonctionnelle peut être annotée : `@FunctionalInterface`
Cela implique une erreur de compilation s'il existe plus d'une méthode abstraite

- ▶ Exemple :

```
@FunctionalInterface  
public interface Filter<T>{  
    public boolean match(T tElem);  
}
```

Méthodes statiques dans les interfaces

- Implémentation de méthodes statiques dans les interfaces

Exemple (dans l'interface précédente) :

```
public static <T> List<T> match(
    Filter<T> filter,
    Collection<T> tElemCollection
){
    List<T> subList = new ArrayList<>();
    if(tElemCollection!=null && filter!=null){
        for(T tElem : tElemCollection){
            if(filter.match(tElem)){
                subList.add(tElem);
            }
        }
    }
    return subList;
}
```

Méthodes par défaut (Defender)

- ▶ Implémentation par défaut de méthodes dans les interfaces
- ▶ Une méthode par défaut doit être préfixée du mot clé : '**default**'
- ▶ Exemple (dans l'interface Filter précédente) :

```
public default List<T> match(Collection<T> tElemCollection){  
    return Filter.match(this, tElemCollection);  
}
```

Lambdas

- ▶ Ce sont des fonctions anonymes
- ▶ Affectables dans une interface fonctionnelle (SAM)

- ▶ Exemples :

```
Filter<String> filter = (tElem)-> tElem!=null && tElem.startsWith(searchedStr);
```

OU

```
Filter.match((tElem)-> tElem!=null && tElem.startsWith(searchedStr), testList);
```

- ▶ Remarque : Cela évite la verbosité des implémentations de Filter

java.util.function (1/3)

- ▶ L'objet '**Function<A,B>**' permet de référencer une fonction prenant en argument une instance de type **A** et retournant une instance de type **B**

```
Function<String, Boolean> filter = (str) -> str != null && str.startsWith(startStr);  
Filter.match2(filter, testList);
```

- ▶ Pour exécuter un objet de type '**Function**' on utilise la méthode '**apply**'

```
public static <T> List<T> match2(  
    Function<T, Boolean> filterFuntion,  
    Collection<T> tElemCollection  
) {  
    List<T> subList = new ArrayList<>();  
    if (tElemCollection != null && filterFuntion != null) {  
        for (T tElem : tElemCollection) {  
            if (filterFuntion.apply(tElem)) {  
                subList.add(tElem);  
            }  
        }  
    }  
    return subList;  
}
```

java.util.function (2/3)

- ▶ Un '**Consumer<A>**' prend un objet de type **A** et ne retourne rien

```
List<String> result = new ArrayList<String>();  
Consumer<String> consumer = (tElem)-> {  
    if(tElem!=null && tElem.startsWith(searchedStr)) result.add(tElem);  
};  
testList.forEach(consumer);
```

`testList.forEach(System.out::println); //Exemple plus cohérent`

- ▶ Remarque : une méthode `forEach()` a été ajoutée aux '**Iterable**'
- ▶ Un objet de type '**Consumer**' s'exécute via la méthode '**accept**'

java.util.function (3/3)

- ▶ Un '**BinaryOperator<T>**' permet d'effectuer une opération entre deux objets de type **T** et retourne un objet également de type **T**

```
BinaryOperator<Integer> operator = (x,y) -> x+y;
```

- ▶ Un objet de type '**BinaryOperator<T>**' s'exécute via la méthode '**apply**'

```
public class FunctionsList<E> extends ArrayList<E> {  
    public E reduce(final E identity, final BinaryOperator<E> binaryOperator)  
    {  
        E result = identity;  
        for(E e : this) {  
            result = binaryOperator.apply(result, e);  
        }  
  
        return result;  
    }  
}
```

Références de méthodes

- ▶ Il est possible de référencer les méthodes (statiques et non statiques) en utilisant l'opérateur '::'

```
Supplier<Double> staticRandom = Math::random;           //() -> Math.random()
```

```
Random r = new Random();  
Supplier<Double> instRandom = r::nextDouble;           //() -> r.nextDouble();
```

```
Function<Random,Double> instRandom2 = Random::nextDouble;
```

```
Function<String,Thread> threadFactory = Thread::new;
```

- ▶ Supplier : la méthode get() retourne une instance de type B
- ▶ Function<A,B> : apply(A) retourne une instance de type B

Stream

- ▶ Représente un flux de données que l'on peut manipuler à la volée (collection, tableau, entrée/sortie, etc...)
- ▶ Les Streams peuvent être séquentiels ou parallèles
- ▶ Opération
 - **Intermédiaire** : Permet de transformer un flux en un autre en manipulant les données (map/filter)
 - **Terminale** : permet d'obtenir un résultat (reduce/collect)
- ▶ Exemple (addition des nombres supérieurs à 9 dans une liste de String) :

```
int result = numberList.stream()  
    .map((str) -> Integer.parseInt(str))  
    .filter((i)-> i>9)  
    .reduce(0, (x, y) -> x+y);
```

java.time (1/2)

- ▶ Nouvelle API de gestion des dates
 - Immuable
 - Chainable
- ▶ **'Instant'** représente un point relatif à l'epoch
- ▶ **'Duration'** représente une durée
- ▶ **'LocalDate'** (date), **'LocalTime'** (heure) et **'LocalDateTime'** (date + heure) correspondent aux dates du système sans indication du fuseau horaire
- ▶ **'ZoneId'** est un identifiant de fuseau horaire et fournit des règles de conversion entre **Instant** et **LocalDateTime**

java.time (2/2)

- ▶ **'ZoneOffset'** correspond à un offset de fuseau horaire (Décalage avec Greenwich. Par exemple : "+01:00")
- ▶ **'ZonedDateTime'** représente une date avec un fuseau horaire (ISO-8601 : par exemple 2016-05-18T12:00:00+02:00 Europe/Paris)
- ▶ **'OffsetDateTime'** représente une date avec un offset par rapport à Greenwich sans indication de localité (ISO-8601 : par exemple 2016-05-18T12:00:00+02:00)

Annotations multiples

- ▶ Il est désormais possible de répéter plusieurs fois une annotation de même type pour un élément donné d'un programme.
- ▶ Les annotations répétables sont signalées à l'aide de l'annotation : `@Repeatable` prenant pour valeur l'interface la contenant

- ▶ Exemple :

```
public @interface Schedules {  
    Schedule[] value();  
}
```

```
@Repeatable(Schedules.class)  
public @interface Schedule {  
    ...  
}
```

Nashorn (1/2)

- ▶ Implémenté entièrement en Java
- ▶ Exécution en ligne de commande via 'jjs'

- ▶ Exécution depuis java :

```
ScriptEngineManager factory = new ScriptEngineManager();  
ScriptEngine engine = factory.getEngineByName("nashorn");  
engine.eval("print(15 + 10)");
```

- ▶ Récupération d'une variable

```
engine.eval("var test = 15 + 10;");  
Integer test = (Integer) engine.get("test");
```

Nashorn (2/2)

- ▶ Invocation d'une fonction :

```
engine.eval("function sum(a, b) { return a + b; }");  
Double sum = (Double)((Invocable) engine).invokeFunction("sum", 30, 20);
```

- ▶ Il est possible d'exécuter du Java depuis le JavaScript

```
var java = Packages.java;  
print('Avant Java');  
java.lang.Thread.sleep(10000);  
print('Après Java');
```

Ou

```
var java = new JavaImporter(java.lang);  
with(java) {  
    print('Avant Java');  
    Thread.sleep(10000);  
    print('Après Java');  
}
```

Stamped Locks

- ▶ Verrou Lecture/Ecriture plus performant mais plus complexe à mettre en œuvre

```
//Lock optimiste en lecture
long stamp = lock.tryOptimisticRead();
work();
//Si une ecriture a eu lieu durant work()
if (!lock.validate(stamp)){
    //Lock pessimiste en lecture
    stamp = lock.readLock();
    try {
        work();
    }
    finally {
        lock.unlock(stamp);
    }
}
```

Autres

- ▶ **'LongAdder'** : Permet de lire et écrire des valeurs numériques de manière concurrente et à grande échelle. (utilisation des méthodes 'add' et 'intValue')
- ▶ **Tri en Parallèle** : `Arrays.parallelSort(myArray);`
- ▶ Méthodes 'Exact' dans la classe Math qui lève une ArithmeticException quand la valeur d'une opération dépasse sa précision
- ▶ SecureRandom : force la JVM à choisir un fournisseur sécurisé pour les randoms
- ▶ Références Optionnelles Optional<T> : permet de spécifier qu'une référence peut être null
- ▶ **'StringJoiner'** : Permet de créer une String où chaque valeur est séparée par un séparateur
`StringJoiner sj = new StringJoiner(",").add("apple").add("banana");`

Sources

- ▶ <http://www.developpez.com/actu/68971/Java-8-est-disponible-la-plate-forme-se-met-aux-expressions-lambdas-tour-d-horizon-des-nouveautes/>
- ▶ <http://www-igm.univ-mlv.fr/~dr/XPOSE2013/JDK18/lambdas.php>
- ▶ <http://www.infoq.com/fr/articles/Java-8-Quiet-Features>
- ▶ <https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html>
- ▶ <http://blog.paumard.org/2014/04/22/50-nouvelles-choses-que-lon-peut-faire-avec-java-8/>

Merci de votre attention

QUESTIONS ?