

Effective Java, Third Edition : Keepin' it Effective

Speakers : Joshua Bloch

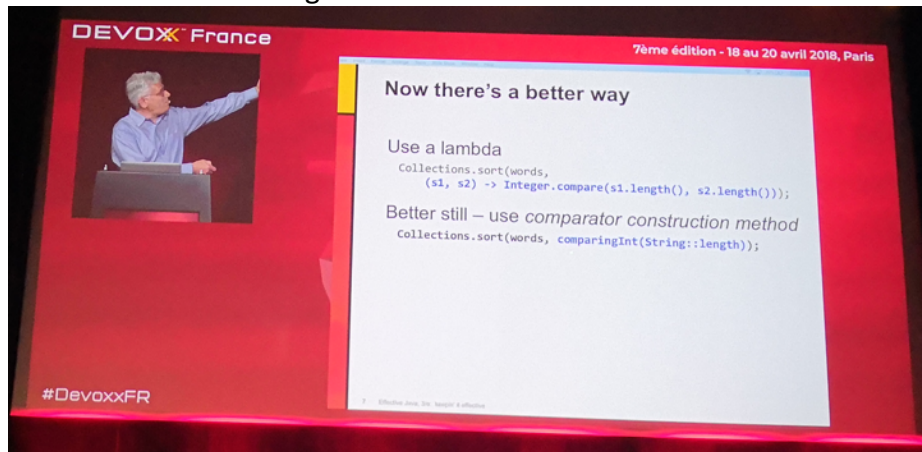
Format : Conférence

Date : 19 avril 2018

Ingénieur logiciel chez Sun puis chez Google, Joshua Bloch vient nous présenter les nouveautés parues dans la 3^{ème} édition de son livre « **Effective Java** » publié en 2017, soit 16 ans après la 1^{ère} édition. Elle comprend un nouveau chapitre et parle de Java 9, des lambdas, des streams, des optionals, des default methods, du try-with-resources et des modules.

1. Prefer lambdas to anonymous classes

Sous le capot, Java utilise l'inférence de type. Cela permet d'avoir un code plus lisible. Nécessite d'utiliser les generics.



Under the hood – *type inference*

- When program says:

```
collections.sort(words, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```
- It means:

```
collections.sort(words, (Comparator<String> (String s1, String s2) -> Integer.compare(s1.length(), s2.length()));
```
- Type inference is magic
 - No one knows the rules but that's OK
 - **Omit types unless they make program clearer**
 - Compiler will tell you if it needs help

Les énumérations peuvent avoir des lambdas pour propriétés :

Now there's a better way

Enum with function object field – impractical sans lambdas

```
public enum Operation {
    PLUS ("+", (x, y) -> x + y),
    MINUS ("-", (x, y) -> x - y),
    TIMES ("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);

    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    @Override public String toString() { return symbol; }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

12 Effective Java, 3rd Edition, Chapter 8

Mises en garde sur les lambdas :

- Elles doivent se passer d'explications
- Ne doivent pas excéder quelques lignes, et si possible une seule
- Si trop longue, extraire une méthode et utiliser une Reference
- Les classes anonymes se diffèrent des lambdas par le fait qu'elles peuvent accéder au this

2. Prefer method references to lambda

En général, utiliser des Reference Methods permet de rendre le code plus succinct.

II. Prefer method references to lambdas

- Lambdas are succinct

```
map.merge(key, 1, (count, incr) -> count + incr);
```

- But *method references* can be more so

```
map.merge(key, 1, Integer::sum);
```

- The more parameters, the bigger the win
 - But parameter names *may* provide documentation
 - If you use a lambda, choose param names carefully!

Un effort est à faire sur le nom des méthodes qui doit être lisible.

Occasionnellement, lorsque le nom de la classe est très long, les lambdas peuvent être privilégiés

Les 5 types de method references que tout développeur Java se doit de connaître :

Type	Example	Lambda Equivalent*
Static	<code>Integer::parseInt</code>	<code>str -> Integer.parseInt(str)</code>
Bound	<code>Instant.now()::isAfter</code>	<code>Instant then = Instant.now(); t -> then.isAfter(t)</code>
Unbound	<code>String::toLowerCase</code>	<code>str -> str.toLowerCase()</code>
Class Constructor	<code>TreeMap<K,V>::new</code>	<code>() -> new TreeMap<K,V>()</code>
Array Constructor	<code>int[]::new</code>	<code>len -> new int[len]</code>

A noter que tout ce qu'on fait avec une Method Reference, on peut le faire avec une lambda.

3. Favor standard functional interfaces

Avant les lambdas, l'utilisation du Template Method était courant.

Depuis, le pattern Strategy est généralement préférable : on passe la lambda au constructeur. La lambda implémente l'interface de la stratégie à utiliser.

```
Before lambdas, Template Method pattern was common
public class Cache<K,V> extends LinkedHashMap<K,V> {
    final int maxSize; // Set by constructor-omitted for brevity

    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return size() > maxSize;
    }
}

Now, Strategy pattern is generally preferable
public LinkedHashMap(EldestEntryRemovalFunction<K,V> fn) { ... }

// Unnecessary functional interface; use standard one instead!
@FunctionalInterface interface EldestEntryRemovalFunction<K,V> {
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
Map<K,V> cache =
    new LinkedHashMap((map, eldestEntry) -> map.size() > maxSize);
```

Java a 43 interfaces fonctionnelles standard (SFI).

Se focaliser sur les 6 interfaces fonctionnelles les plus essentielles : UnaryOperator, BinaryOperator, Predicate, Function, Supplier et Consumer.

The 6 basic standard functional interfaces

Interface	Function Signature	Example
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

Most of the remaining 37 interfaces provide support for primitive types. Use them or pay the price!

L'utilisation des interfaces fonctionnelles rend plus facile l'apprentissage des API car elle réduit leur surface d'apprentissage.

Beaucoup de SFIs fournissent des default method.

Quand ne pas utiliser les SFIs ?

- Lorsqu'il n'y a pas d'équivalent dans les SFI
- Cas spécial du Comparator

Critères pour écrire sa propre FI :

- Lorsque vous écrivez une FI, souvenez-vous qu'il s'agit d'une interface.

4. Use streams judiciously

- Joshua monte un exemple de calcul des anagrammes d'un dictionnaire plus lisible en programmation impérative qu'avec des streams
- Puzzler : les streams ne supportent pas directement les chars
 - String's chars() method renvoie un IntStream

Puzzler Explanation

```
"Hello world!".chars()  
    .forEach(System.out::print);
```

Prints 721011081081113211911111410810033

String's chars method returns an **IntStream**

En conclusion : les streams sont pratiques, mais ce n'est pas la panacée

5. Use caution making streams parallel

- `parallel()` ne doit pas être utilisé sur des streams avec états