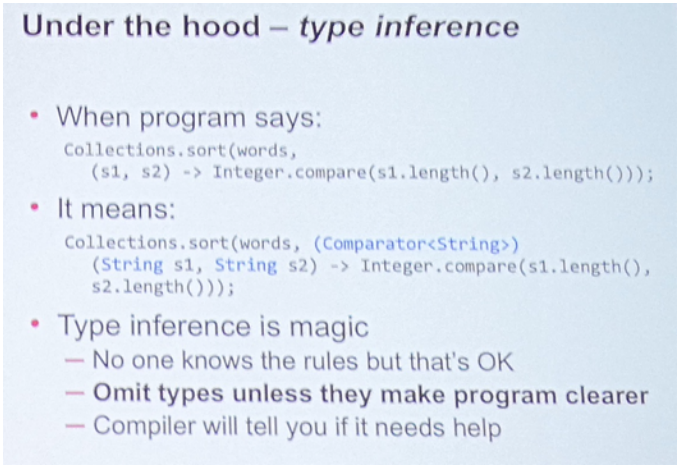
Effective Java, Third Edition : Keepin' it Effective

Speakers : Joshua Bloch
Format : Conférence
Date : 19 avril 2018

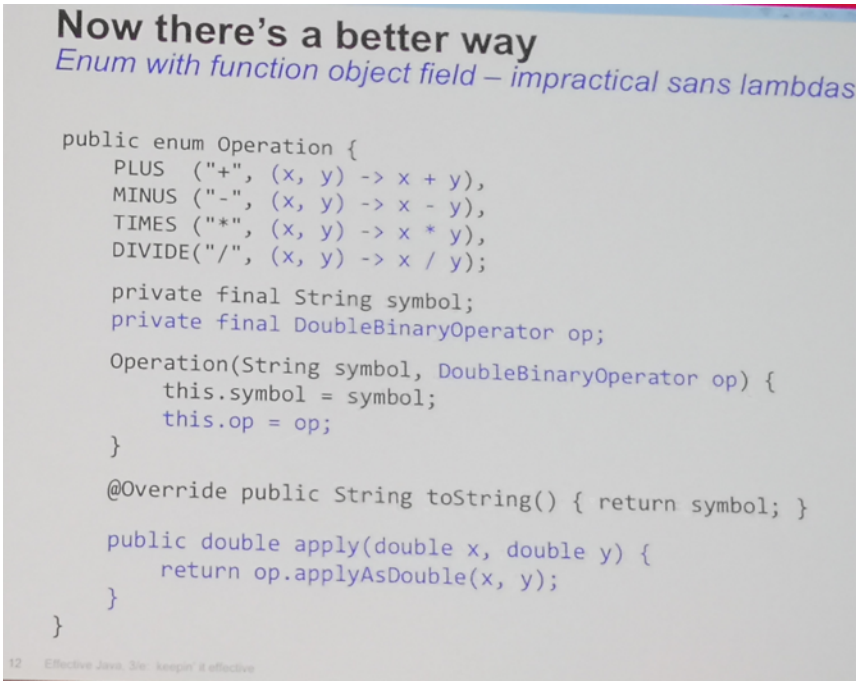
Ingénieur logiciel chez Sun puis chez Google, Joshua Bloch vient nous présenter les nouveautés parues dans la 3ième édition de son livre « **Effective Java** » publié en 2017, soit 16 ans après la 1ière édition. Elle comprend un nouveau chapitre et parle de Java 9, des lambdas, des streams, des optionals, 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.



Les énumérations peuvent avoir des lambdas pour propriétés :

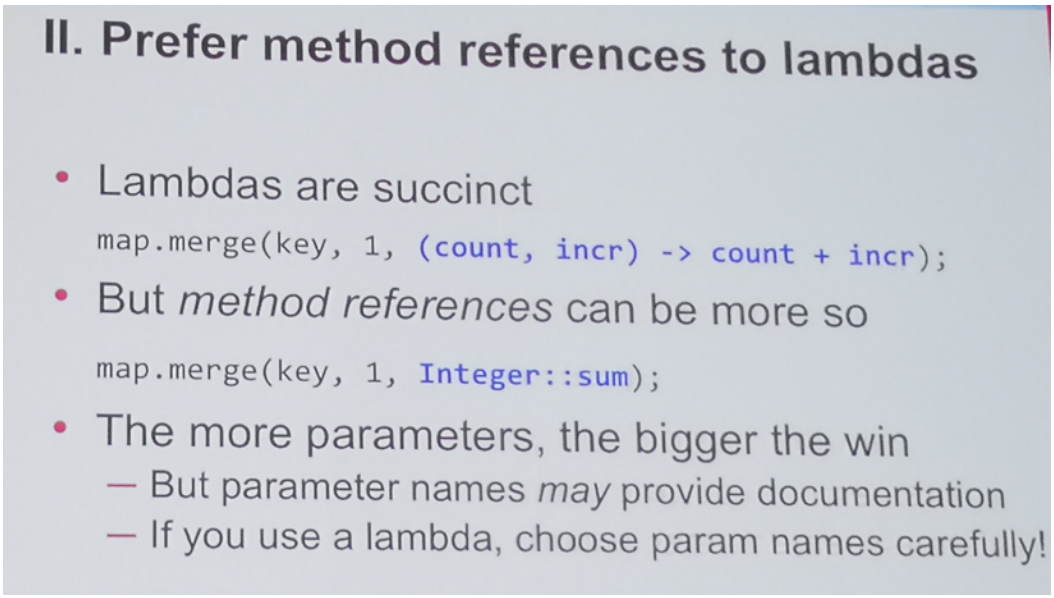


Mises en garde sur les lambdas :

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

2. Prefer method references to lambda

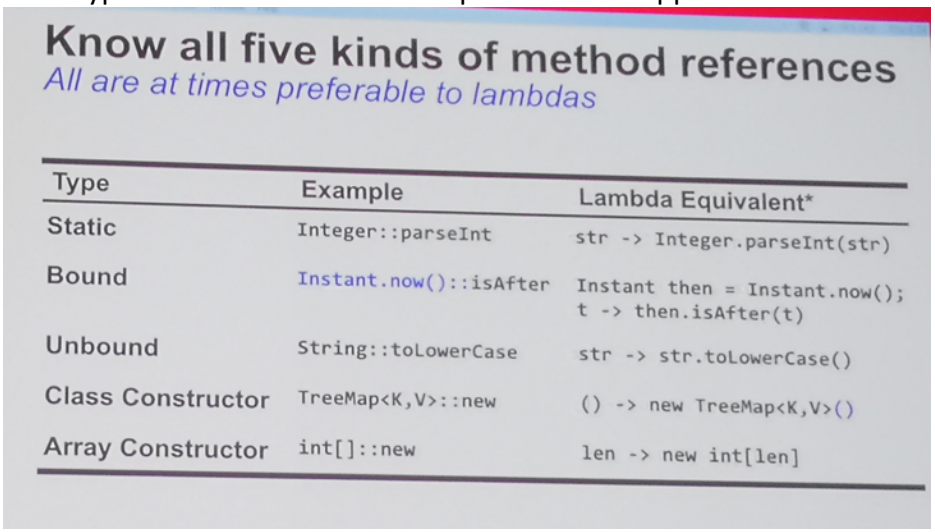
En général, utiliser des Reference Methods permet de rendre le code plus succinct.



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 privilégier

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



Know all five kinds of method references

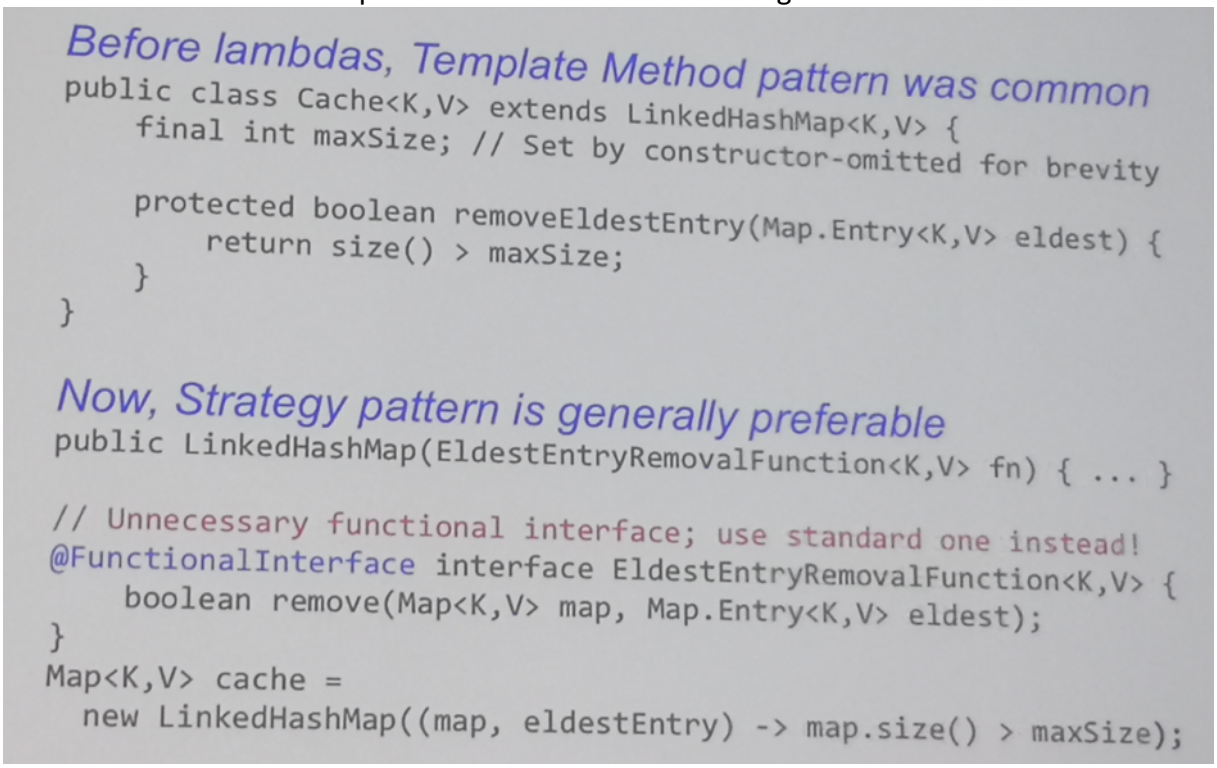
All are at times preferable to lambdas

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

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.



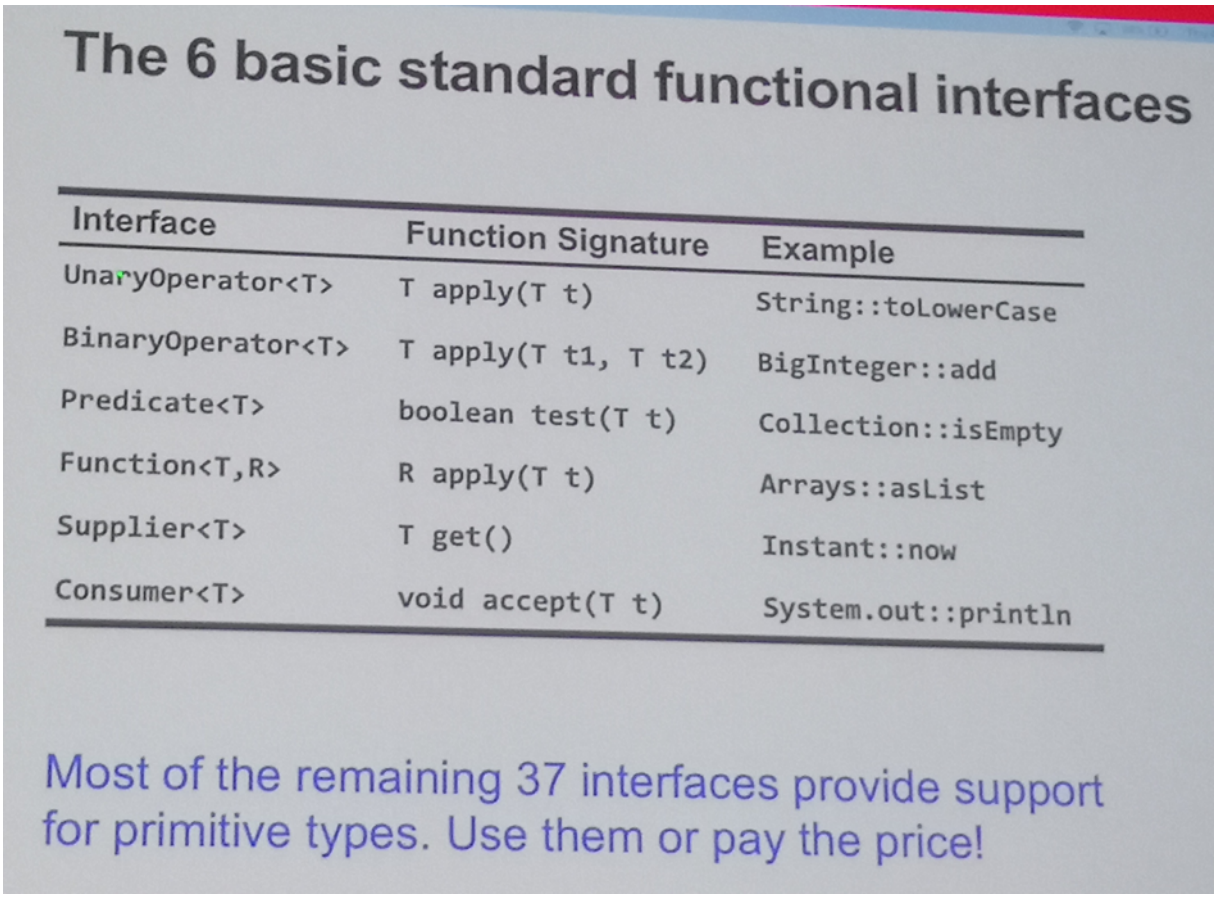
Before lambdas, Template Method pattern was common

```
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.



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 ?

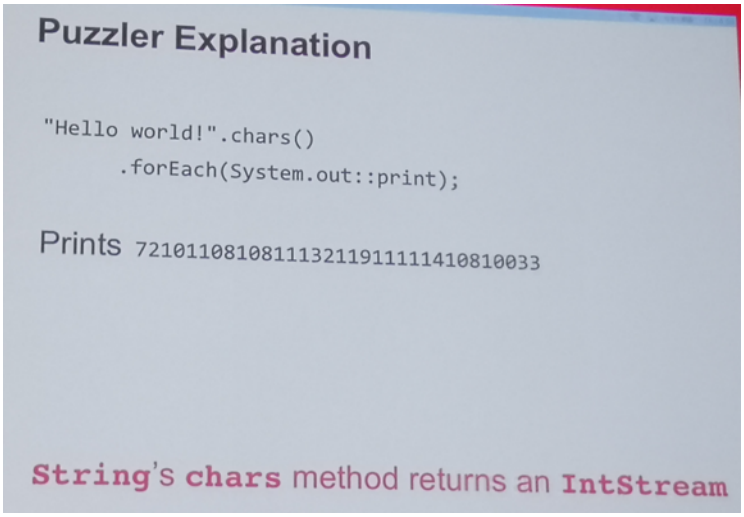
- Lorsqu'il n'y a pas d'équivalent dans les SFI
- Cas spécial du Comparator

Critères pour écrire sa propre FI :

- Lorsque vous écrivez une FI, souvenez-vous qu'il s'agir 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



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

5. Use caution making streams parallel

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