JDK10 - JDK11 - JDK12

Depuis JDK9, Oracle change le cycle de sorties des différentes versions de Java. Alors qu'avant, une nouvelle version de Java sortait tous les 2, 3 ans, c'est maintenant tous les 6 mois qu'une nouvelle version sera disponible.

Parmi ces version, certaines sont étiquetées *long term support* (LTS) et sont supportées par Oracle plusieurs années tandis que, pour les autres, le support ne durera que 6 mois. Les « prochaines » versions LTS de Java sont la **8** et la **11**.

Ce billet recense quelques nouveautés des versions 10, 11 et 12 de Java et ne sera pas exhaustif. Je ne m'intéresse qu'à quelques fonctionnalités visibles pour les développeurs et les développeuses. Mes sources principales sont les pages Oracle recensant les *JDK Enhanced Proposal* (JEP).

JDK10

Cette version du JDK voit passer plusieurs améliorations du *garbage collecor* (GC), par exemple du parallélisme JEP307, une amélioration du *class data sharing* (CDS) introduit avec JDK5 JEP301,la possibilité d'allouer de la mémoire sur d'autres périphériques JEP316, des améliorations internes...

Ce qui est le plus visible dans cette version 10, est l'inférence de type.

Inférence du type des variables locales

Local-variable type inference JEP286 étend l'inférence de type aux déclarations de variables locales avec initialiseurs, la variable d'un *enhanced for* et les variables d'un *for*. Le mot utilisé est ***var***. Il n'est pas un mot-clé (*keyword*) mais un nom de type réservé ce qui limitera l'impact de l'ajout sur les codes existants.

Il est donc possible d'écrire :

```
var i = 5;
var s = "Hello world";
var l = new ArrayList<String>();
var heu = videos.stream()
    .map(v -> v.getAuthor())
    .sorted()
    .distinct()
    .count();
var j = foo(i);
```

Le troisième exemple me convainc peu par-ce que je préfère que `l` soit de type `List<String>` et pas `ArrayList<String>` mais j'aime bien le quatrième exemple où l'on peut envisager que lorsque l'on commence à enchainer les appels de méthodes, on ne sache pas encore très bien quel sera le type « final ».

Il est à noter que cela ne fonctionnera pas pour un tableau ou `null`.

Certaines personnes ajoutent que `var` amène une meilleur lisibilité de l'indentation (puisque le « type » fait toujours 3 lettres).

Pour plus de détails, lire par exemple l'article de Nicolai Parlog chez codefx.

JDK11

La version 11 apporte — à mon sens — deux changements visibles pour les développeurs *lambdas*.

Local-variable syntax for lambda parameters

local-variable syntax for lambda parameters ajoute le mot clé `var` aux lambdas. Là où l'on écrivait :

```
(x, y) -> x.process(y)
```

on pourra écrire :

```
(var x, var y) -> x.process(y)
```

Cet ajout est principalement là dans un soucis d'uniformisation de la syntaxe JEP323

Launch single-file source-code programs

Ça y est, Java n'est plus compilé et on peut écrire des scripts Java ! Carrément des ***shebang files*** JEP330. En fait, ce n'est pas vrai mais c'est proche.

Pour rappel, le lanceur Java peut lancer un fichier `class`, lancer la classe principale d'un fichier `jar` ou encore lancer la classe principale d'un module. Aujourd'hui, il peut également lancer une classe déclarée dans un fichier source.

On peut donc écrire :

```
public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello " + args[0]);
    }
}
```

et exécuter le programme avec la commande suivante. Aucun fichier `.class` n'est généré, le *bytecode* se trouve en mémoire.

```
$ java Hello.java Alice
```

Si mon fichier ne respecte pas la convention de nommage java et s'appelle par exemple `hello`, je peux écrire :

```
$ java --source 11 hello Alice
```

Il reste un pas à franchir pour écrire un script en Java. Allons-y. J'écris le code suivant dans un fichier nommé `hello` et je rends le fichier exécutable.

```
#!/usr/lib/jvm/java-11-openjdk-amd64/bin/java --source 11

public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello " + args[0]);
    }
}
```

Je peux alors entrer la commande :

```
./hello Alice
```

Dans les changements peu visibles pour les développeurs et développeuses, je note :

- *nest-based access control* JEP181 concernant les classes internes.

  Pour une classe `Outer` et une classe interne `Inner`, les *bytecodes* générés dans `Outer.class` et `Outer$Inner.class` seront différents par rapport aux *bytecodes* générés dans les versions antérieures de Java et le contrôle d'accès au code amélioré.

  Pour comprendre : la JEP181, un article de Ganesh Pagade chez Baeldung et de Peter Verhas chez dzone.

- *dynamic class-file constants* JEP309 est complexe. Le but est d'étendre le *constant pool* et d'y ajouter d'autres types de valeurs.

  Actuellement, Java utilise une table contenant les constantes du programme, le (*constant pool*). C'est assez facile à comprendre pour les types primitifs et les chaines (`String`). C'est d'ailleurs pour ça que :

  ```
  String s1 = "Marlène";
  String s2 = "Marlène";
  System.out.println(s1 == s2); // true
  ```

  Au fil des versions de Java, d'autres valeurs ont été ajoutées à ce *constant pool* comme les « littéraux de classes constants » (*class literal constants*). Toutes les classes, les interfaces, les énumérations… sont des instances de `Class` dans une application Java et `Class` permet par exemple l'introspection :

  ```
  Class<String> c = String.class;
  System.out.println(Arrays.toString(c.getMethods()));
  ```

`String.class` est une instance de `Class<String>` comme `"Hello world"` est une instance de `String`. `String.class` est constant et se trouve dans le *constant pool*.

Actuellement si `A` et `B` sont deux constantes et se trouvrent donc dans le *constant pool*, `Math.max(A, B)` ne s'y trouve pas. Ceci devrait évoluer.

- *HTTP client* standardisé comme présenté dans le JDK9. J'en avais parlé dans ce billet.
- mise à jour pour rester *up to date* : *Key Agreement with Curve25519 and Curve448*, *ChaCha20 and Poly1305 cryptographic algorithms*, *Unicode 10* et *TLS1.3*

Pour le reste dans les changements apportés au JDK et non présentés ici, ça se passe là JDK11

JDK12

Dans cette version du JDK, une seule fonctionnalité est destinée aux développeurs et développeuses « standard ». Il s'agit du ***switch*** qui devient une expression (JEP325). *switch* pourra être utilisé comme une **instruction** — comme avant — ou comme une **expression**.

Cette fonctionnalité est en *preview* c'est-à-dire qu'elle est pleinement fonctionnelle et documentée et est présente afin d'être complètement testée. Elle pourrait disparaitre et pour être utilisée il est nécessaire de compiler et d'exécuter son code avec le *switch* `--enable-preview`.

1. Au `case <label>:` s'ajoute un `case <label> ->`

   Ceci permet de ne pas utiliser de `break` car seule l'instruction (ou le bloc d'instructions) suivant la flèche sera exécuté. On pourra donc écrire, pour peu qu'une `enum` *Season* existe :

   ```
   Season s = // a season
   switch (s) {
       case SPRING, SUMMER -> System.out.println("It's hot");
       case WINTER, AUTUMN -> System.out.println("It's cold");
       default -> System.out.println("Claim climate change");
   }
   ```

2. Le *switch* est désormais également une **expression**, il a un type et une valeur. On pourra par exemple écrire :

   ```
   Season s = // a season
   String message = switch(s){
       case SPRING, SUMMER -> "It's hot";
   ```

```
    case WINTER, AUTUMN -> "It's cold";
    };
```

Les autres fonctionnalités concernent la *java virtual machine* (JVM). Le *garbage collecor* (GC) : ajout de *Shenandoah*, un GC à faible temps de pause à titre expérimental et, à ma connaissance, c'est toujours G1 qui est utilisé. Optimisation de G1 pour qu'il rende plus rapidement la mémoire non utilisée du tas (*heap*) au système lorsque le processus Java est en pause. Pratique pour les environnements conteneurisés et possibilité de fixer un objectif de temps maximal d'exécution du *garbage collector*. Au sujet des architectures, *One AArch64 Port, Not Two*, supprime un des deux ports de Java pour les architectures ARM pour n'en conserver qu'une seule.

Pour les détails, ça se passe là JDK12

Au vu des cycles de développement, je pense que je vais avoir tendance à utiliser les versions LTS.

Crédit photo chez DeviantArt par TODO