

# Java 10 ou Java 11

## Ne loupez pas le train !

La grève de la SNCF n'est peut-être pas terminée, mais ne loupez pas pour autant les nouveautés du Java et faite donc le tour de son *release train*. Ici, le billet est gratuit ! « Attention au départ... »

**L'** une des principales nouveautés concerne le système de release de Java : le Java va passer d'un système de *feature release* tous les deux ans – qui, dans la réalité, approchait plutôt des trois ans – à un *release train* bien plus fréquent ; une release tous les six mois ! Jusqu'alors, Oracle définissait un ensemble de fonctionnalités devant être intégrées dans une future version du Java et c'est seulement lorsqu'elles étaient prêtes que la version en question pouvait être relâchée.

### Ne ratez pas le nouveau « release train »

À partir de Java 10, une version sortira donc tous les six mois en *release train*. Les nouvelles fonctionnalités sont développées dans des branches spécifiques : lorsqu'elles sont prêtes, elles sont ajoutées à la branche principale et, en suivant, à la future release. Avec un tel système, les sorties de versions apporteront moins de nouvelles fonctionnalités mais seront plus fréquentes et plus régulières. Enfin, il y a version et version : la plupart des versions du Java ne seront supportées que jusqu'à la sortie de la version suivante, soit seulement six mois. Seule une version tous les trois ans sera en Long Term Support (LTS), en commençant

par le Java 11 annoncé pour l'automne 2018. La version 9, qui avait pourtant apporté de nombreux changements, ne sera supportée que pendant six mois de

plus, ce qui fait que personne ne va continuer à l'utiliser en production. Comme il n'y a aucune procédure prévue entre les versions non-LTS, il va falloir passer très vite à la nouvelle version (11) ou rester un moment sur une précédente taguée LTS – avant la version 9. Ce système de *release train* et de version LTS ressemble beaucoup à ce que pratiquent plusieurs distributions Linux comme Ubuntu ou Fedora. Pour

The java launcher will display version strings and system properties as follows, for a hypothetical build 13 of JDK 10.0.1:

```
$ java --version
openjdk 10.0.1 2018-04-19
OpenJDK Runtime Environment (build 10.0.1+13)
OpenJDK 64-Bit Server VM (build 10.0.1+13, mixed mode)
```

Similarly, for a hypothetical build 42 of JDK 11, an LTS release:

```
$ java --version
openjdk 11 2018-09-20 LTS
OpenJDK Runtime Environment (build 11+42-LTS)
OpenJDK 64-Bit Server VM (build 11+42-LTS, mixed mode)
```

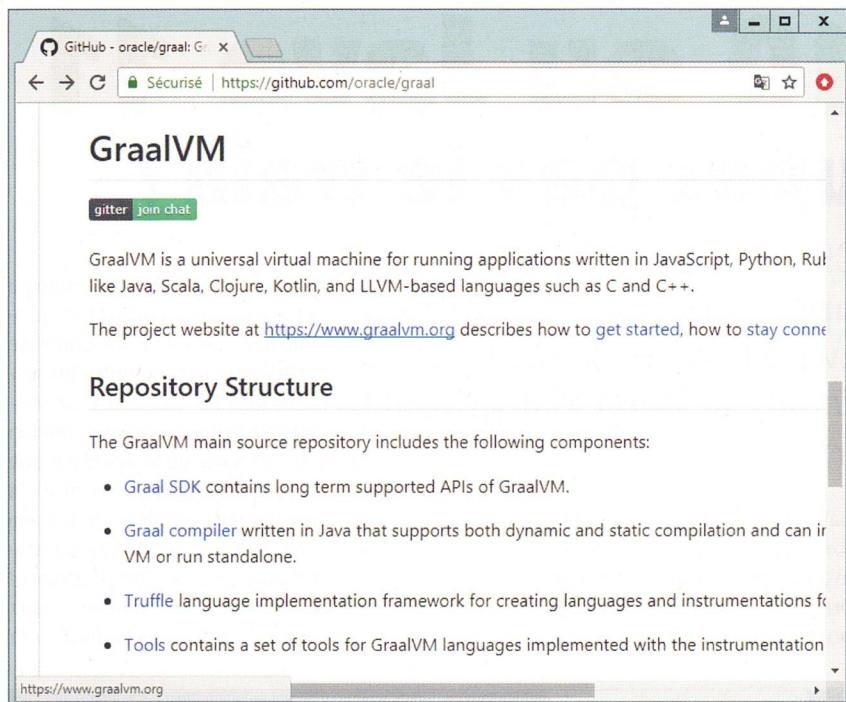
If an implementor assigns a vendor version string of, e.g., 18.9 to a JDK 11 LTS build then it would be displayed:

```
$ java --version
openjdk 11 2018-09-20 LTS
OpenJDK Runtime Environment 18.9 (build 11+42-LTS)
OpenJDK 64-Bit Server VM 18.9 (build 11+42-LTS, mixed mode)
```

In detail, the output of the java launcher's version-report options will be formatted as follows, where \${LTS} expands to "\u0020LTS" if the first three characters of \$OPT are "LTS", and \${JVV} expands to "\u0020\${java.vendor.version}" if that system property is defined:

```
$ java --version
openjdk ${java.version} ${java.version.date}${LTS}
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})
```

```
$ java --show-version < ... >
openjdk ${java.version} ${java.version.date}${LTS}
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})
```



la petite histoire, un système de versionning basé sur les dates à été envisagé puis – fort heureusement – abandonné. Nous allons maintenant faire un petit tour des nouveautés apportées à la version 10 – et version 11 également puisqu'elles sont quasiment identiques.

## Inférence de type pour les variables locales

C'est sans nul doute la grande nouveauté de Java 10 : le mot clé var qui, comme en C#, peut être utilisé pour faire trouver au compilateur le type de la variable. Ce mécanisme s'appelle *l'inférence de type*. Il ne faut pas en abuser, car le code pourrait vite devenir illisible, mais dans certains cas, notamment la déclaration de variables de type complexe, il le simplifie élegamment. Il ne peut être utilisé qu'à l'intérieur d'une méthode, donc seulement pour les variables locales, mais pas pour les variables membres de la classe. L'autre condition est, bien entendu, que le type puisse être inféré par le compilateur et qu'il

y ait initialisation : var peut aussi être employé dans des boucles for – surtout de type foreach pour le type de la variable tampon – ainsi que dans les blocs try-with-resource. Le Java n'en reste pas moins un langage strict et fortement typé. Le mot clé var permettra de simplifier les déclarations en évitant de répéter deux fois le type, comme montré ci-dessous.

En Java 9 :  

```
TypeComplex obj = new TypeComplex();
Map<String,List<TypeComplex>>
map = new
HashMap<String,List<TypeComplex>>();
```

En Java 10 avec le mot clé var :  

```
var obj = new TypeComplex();
var map = new
HashMap<String,List<TypeComplex>>();
```

La visibilité n'en est que meilleure, la répétition du type n'apportant rien de particulier, l'écriture est plus concise sans aller dans l'incertitude. Ce nouveau mot clé est, pour l'instant du moins, limité aux variables locales des méthodes. Les constantes (les « immuables ») ne peuvent être définies ainsi. Cela ne veut absolument pas

**Partez, vous aussi, à la quête du Graal (...) en exécutant vos programmes avec le compilateur JIT écrit en Java :**  
<https://github.com/oracle/graal>.

dire qu'il faut l'employer n'importe où et n'importe comment, par « flegme », au risque d'introduire des problèmes dans le code, comme dans cet exemple :

En Java 9 :  

```
byte flags = 0;
short mask = 0xffff;
long base = 17;
```

En Java 10 - Dangereux, le compilateur a choisi par inférence le type int pour toutes les variables :

```
var flags = 0;
var mask = 0xffff;
var base = 17;
```

Le mieux est d'éviter d'utiliser le mot-clé var pour les variables littérales, de type primitif (int, float, double, char...). De toute façon, dans ce cas de figure, il diminue la lisibilité du code, masquant la déclaration du type ou obligeant à la déduire lors de la relecture. Dans ce code, en revanche, encore tiré du site de l'openjdk, le bénéfice est plus évident :

En Java 9 :  

```
void removeMatches(Map<? extends String, ? extends Number> map, int max) {
    for (Iterator<? extends Map.Entry<? extends String, ? extends Number>> iterator =
        map.entrySet().iterator();
        iterator.hasNext();)
        Map.Entry<? extends String, ? extends Number> entry = iterator.next();
        if (max > 0 && matches(entry)) {
            iterator.remove();
            max--;
        }
}
```

L'utilisation de var supprime les déclarations de type longues et illisibles de ces variables locales. Il n'y a généralement aucun intérêt à écrire explicitement les types imbriqués pour l'Iterator ainsi que les Map.Entry locaux dans ce genre de boucle. La partie initialisation du for s'en voit grandement raccourcie et simplifiée, tout comme la lecture

globale du code, et sans aucune perte d'information.

En Java 10 :

```
void removeMatches(Map<? extends String, ? extends Number> map, int max) {
    for (var iterator = map.entrySet().iterator(); iterator.hasNext(); ) {
        var entry = iterator.next();
        if (max > 0 && matches(entry)) {
            iterator.remove();
            max--;
        }
    }
}
```

## Copy factory methods

L'API Collection a été enrichie de méthodes statiques permettant la copie de collection existantes : List.copyOf(), Set.copyOf() et Map.copyOf(). Dans les trois cas la collection retournée sera une collection immuable. Si la collection existante était déjà une collection immuable, elle sera retournée directement. Il n'y a en effet nul besoin de créer une copie d'une collection immuable qui sera elle-même immuable. Rappelons que ce sont les collections qui sont immuables et non les objets qui y sont stockés. Voici, pour exemple, l'implémentation actuelle de List.

```
copyOf():
static <E> List<E> copyOf(Collection<? extends E> coll) {
    if (coll instanceof
        ImmutableList)
        return (List<E>)coll;
    } else {
        return (List<E>)List.of(coll.
        toArray());
    }
}
```

## Optional. orElseThrow()

Optional.orElseThrow() est désormais la méthode à privilégier pour récupérer un élément dans un Optional, ce au détriment de Optional.get().

Comme son nom semble l'indiquer, s'il n'y a pas d'élément, une exception sera lancée – ce qui, au passage, est déjà le cas pour Optional.get(), même si les développeurs l'oublient parfois. D'autres ajouts devraient être apportés à la classe Optional dans les prochaines releases.

## API Process

Il y a toujours eu des possibilités assez réduites en Java pour contrôler et gérer les processus du système d'exploitation. Cela restait (très) compliqué et variait en fonction du système d'exploitation. La simple récupération du PID demandait de nombreuses lignes de code. Maintenant, c'est aussi simple que ça, et valable quel que soit le système d'exploitation sur lequel s'exécute l'application : `Process.GetCurrentPid()`

D'autres méthodes directes de manipulation des données des processus (PID, noms, état, occupation mémoire) ont été ajoutées à cette API. Elles étendent l'aptitude de Java à interagir de manière uniforme avec les systèmes d'exploitation.

## Reader. transferTo(Writer)

Amélioration plus mineure, une nouvelle méthode a été ajoutée à

la classe Reader. Elle permet de transférer directement tous les caractères d'un objet Reader vers un Writer :

```
Reader reader = new FileReader("from.
txt");
Writer writer = new FileWriter("to.txt");
reader.transferTo(writer);
```

## Parallel Full GC for G1

C'est sans doute le changement le plus important en termes de performance. G1 GC (Garbage First Garbage Collector) est, depuis Java 9, l'algorithme par défaut du Garbage Collector. Jusqu'alors, quand le collector n'arrivait plus à nettoyer assez rapidement la mémoire de manière concurrente (le heap ou tas, zone mémoire où est gérée l'allocation dynamique), il déclenchaît un Full GC mono-threadé, « freezant » généralement l'application. L'implémentation du G1 a été revue en Java 10 et est désormais multi-threadée, le nombre de threads étant défini par la variable `-XX:ParallelGCThreads`. Beaucoup apprécieront cette nouvelle fonctionnalité, même sans le savoir. Pour des programmes très consommateurs de ressources mémoire, cet apport est très important, mais il faudra réfléchir à la meilleure valeur possible pour la dite variable.

## Thread-Local Handshakes

Dans les précédentes versions, Java 9 inclusive, la seule manière de réaliser des actions impliquant plusieurs threads simultanément – récupérer un dump de thread, gérer des locks (verrous) – consistait à placer tous les threads dans un safepoint. Le problème est que cette opération est coûteuse en ressources et nécessite de coordonner tous les threads de l'application. Java 10 apporte le mécanisme de Handshakes dont la principale caractéristique est, justement, de

## Collectors to unmodifiable List

**Les collectors de l'API Stream ont été enrichis de quatre nouvelles méthodes permettant la création de collections immuables :**

- `Collectors.toUnmodifiableList()`
- `Collectors.toUnmodifiableSet()`
- `Collectors.toUnmodifiableMap (keyFunc, valueFunc)`
- `Collectors.toUnmodifiableMap (keyFunc, valueFunc, mergeFunc)`.

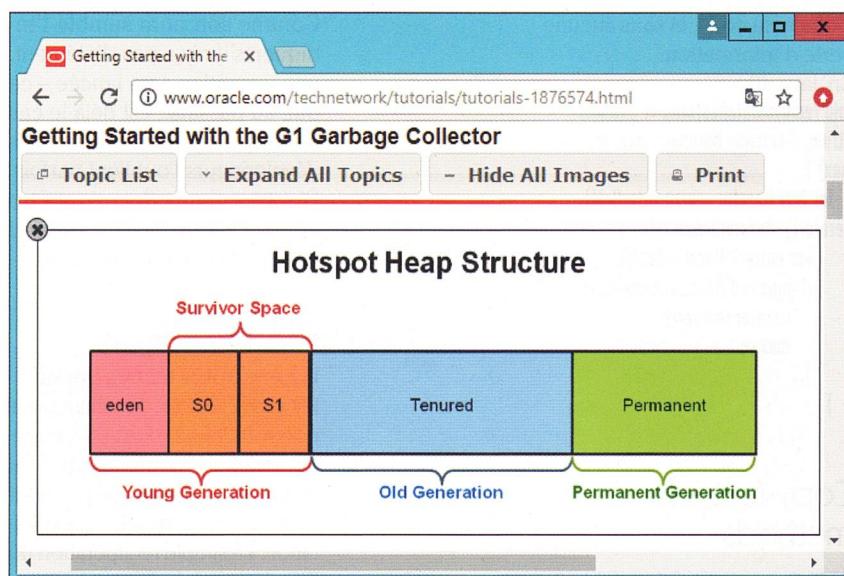
pouvoir exécuter des callbacks sur des threads sans nécessairement passer par un safepoint. Cette fonctionnalité plutôt bas niveau ouvre la porte à d'autres optimisations du même genre pour la JVM qui devraient pointer le bout de leur nez d'ici quelques releases.

## Heap Allocation on Alternative Memory Devices

Les applications « modernes » ont, malheureusement, besoin de toujours plus de mémoire. Les serveurs n'en manquent généralement pas, loi de Moore oblige, mais malgré cela, il n'y en a quand même jamais assez! La première et principale piste est de regarder si notre programme ne pourrait pas être moins gourmand... En attendant, et comme tout le monde ne le fera pas, il faut essayer de trouver des palliatifs. Il existe, fort heureusement, des devices de stockage avec des performances en termes de lecture-écriture proches de la DRAM. Cette nouvelle fonctionnalité, implémentée par Intel, qui fournit aussi une implémentation pour ses cartes flash haute performance (NV-DIM 3D XPoint), permet de démarrer une JVM en utilisant une heap (zone d'allocation mémoire dynamique) placée en dehors de la RAM. C'est très bien, mais n'oubliez pas qu'il vaut mieux améliorer les performances avec un meilleur code.

## Application Class-Data Sharing

Cette autre nouveauté permet de partager les métainformations des classes – ou class metadata en Java, une partie du « metastore » – entre plusieurs JVM (Java Virtual Machine, voyons...) tournant sur le même appareil. Cela permet d'optimiser le démarrage



et surtout l'empreinte mémoire de ces JVM. En cette ère de Cloud et de containerisation à outrance, cette fonctionnalité peut s'avérer très efficace.

## Et le Java 11 dans tout ça ?

Tout est-il dit, ou plutôt fait, dans les Java 10, ou 11, est-il un peu plus que la version LTS du jdk 10 ? C'est bien le presque-équivalent de Java 10 mais en version LTS (Long Term Support, ou version avec un support à long terme), et il y a quand même quelques nouveautés. Sa sortie est prévue dès septembre 2018. Cette version devrait avoir un support Oracle de premier niveau jusqu'en septembre 2023 et un support étendu incluant des correctifs et des alertes de sécurité jusqu'en 2026. Seules trois nouvelles fonctionnalités sont annoncées à ce jour, bien qu'il soit encore possible que d'autres viennent s'y greffer d'ici là. Le format de fichier de classe Java va être étendu afin de prendre en charge une nouvelle forme de pool de constante. Un nouveau type – immuable – apparaît, le type Constant Dynamic. Le mot clé var pourra aussi être

### La structure du heap à la sauce du G1GC, le garbage collector de Java.

utilisé lors de la déclaration des paramètres formels d'une expression lambda typée implicitement.

Avant Java 11 :  
`ITest op = (@ATest  
NomDeClasseVraimentTropLongAEcrire  
x, final  
NomDeClasseVraimentTropLongAEcrire  
y, final  
NomDeClasseVraimentTropLongAEcrire  
z,) -> .... ;`

En Java 11 :  
`ITest op = (@ATest var x, final y, final  
z) -> .... ;`

Adieu Corba, JEE et JavaFx ? Parallèlement à ces ajouts, certaines fonctionnalités vont purement et simplement disparaître. C'est notamment le cas de Java FX ainsi que des modules Corba et Java JEE. Dépréciés depuis le Java 9, le retrait des modules Java EE et Corba était une mesure annoncée qui devient effective avec le Java 11.

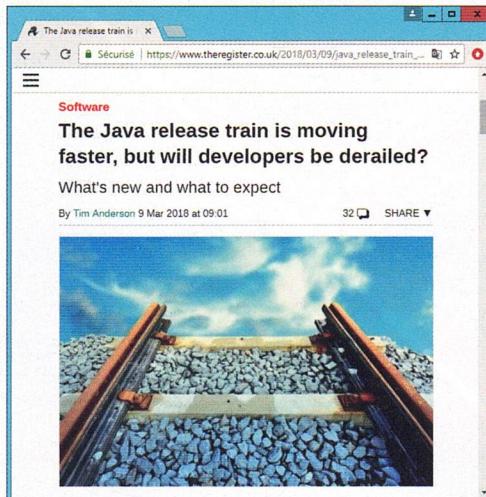
### CORBA

La suppression de Corba se justifierait par deux raisons. Tout d'abord, ce module datant des années 90 ne recèle rien d'autre que des classes et interfaces de communication entre objets. Deuxièmement, il n'a pas été mis à jour pour prendre en compte les meilleures pratiques de développement logiciel, telles que l'interface first-class et l'annotation-based configuration.

développement d'applications Java modernes (sic). Ensuite – ou bien est-ce la première et unique raison ? – les coûts de maintenance de ce module seraient trop élevés comparés aux bénéfices rapportés. Néanmoins, cette suppression risque fort d'entraîner des dysfonctionnements sur des implémentations existantes. Celles-ci risquent de ne pas fonctionner du tout si elles n'incluent qu'une partie de l'API CORBA actuelle. Elles s'attendront certainement à ce que le JDK leur fournit le reste et elles pourront attendre longtemps. Qui plus est, il n'existe pas de versions alternatives pour CORBA et il est très peu probable qu'un tiers se décide à en prendre la charge à l'heure qu'il est, et surtout aussi rapidement.

#### JEE

Concernant la plate-forme Java EE, sa suppression serait due cette fois à des difficultés de maintenance dans Java SE au fil de ses évolutions. Sortie en décembre 2011, Java SE6 était accompagnée d'une large panoplie des services web nécessaires aux développeurs. Quatre étaient dédiés à Java JEE : JAX-WS (Java API for XML-based Web Services), JAXB (Java Architecture for XML Binding), JAF (JavaBeans Activation Framework) et Common Annotations for Java. Ces ajouts de fonctionnalités pour Java EE ont entraîné des complications sévères pour Java SE, notamment parce qu'ils incluaient des fonctionnalités sans rapport avec cette dernière... D'après Oracle, avec des versions « stand-alone » disponibles sur des sites tiers, il ne serait plus nécessaire de l'inclure dans Java SE ou dans le JDK. Il n'empêche que certaines applications pourraient ne pas compiler si elles s'appuient sur un support



prêt à l'emploi du JDK pour les API et outils Java EE. Des incompatibilités binaires et de source risquent de se produire lors de la migration de JDK 6, 7 et 8 vers une version ultérieure. Oracle recommande aux développeurs concernés par ces risques de déployer sur des versions alternatives des technologies Java EE. En résumé, après avoir négligé et même méprisé JEE, après avoir un temps pensé sérieusement à en faire une version propriétaire, Oracle balance JEE à la poubelle et dit gentiment aux développeurs de se débrouiller avec des versions JEE d'autres éditeurs ou

**The register émet des doutes sur ce fameux train de la nouveauté. Fera-t-il dérailler les développeurs ?**

communautés. Tant pis pour la rétrocompatibilité et l'investissement passé par de très nombreux développeurs. Apparemment, Oracle n'en a cure.

#### JAVAFX

JavaFX ne sera pas totalement supprimé, mais disponible comme module séparé. Enfin, au moins au départ. Prudence tout de même pour les futures versions. Le Java FX permet pourtant la création d'applications graphiques de type client lourd d'une manière simple en respectant bien l'architectural pattern MVC. Bizarrement, Swing est préservé alors qu'il est bien moins adapté que le Java FX pour coder des applications « modernes ». Il est sans doute condamné au même sort à plus ou moins long terme. Le principe est le même que pour Java EE – ou plutôt devrait-on dire désormais Jakarta EE. Oracle s'en désolidarise complètement et le renvoie à son niveau de module séparé du JDK. Le projet étant open source, il sera géré par l'OpenJFX qui est en train de s'organiser autour de Johan Vos et de Gluon. ○

THIERRY THAUREAUX

## Experimental Java-Based JIT Compiler

**Le JIT est la partie de la JVM qui optimise le code au fur et à mesure de son exécution. Il est, à l'heure actuelle, écrit en C++ et sa maintenance serait de plus en plus ardue. Graal est un tout nouveau projet d'Oracle faisant partie intégrante d'un projet plus important, Metropolis, dont le but est de faire du Java on Java. Il faut donc pour cela faire tourner les programmes Java avec une JVM écrite en Java, et non en C++. Graal est ainsi un compilateur écrit en Java et qui, à partir du bytecode Java, est capable de générer du code natif optimisé. Son implémentation en Java en faciliterait l'écriture et la maintenance. Il n'est encore qu'en phase bêta, ne l'utilisez surtout pas en production. Si vous voulez le tester depuis Java 10, utilisez ces options pour la JVM :**

**-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler**