## Après Java 8, Java 9 et 10

Speakers : Jean-Michel Doudoux (Oxiane)
Format : Conférence
Date : 20 avril 2018

## Introduction

Beaucoup de choses bougent en ce moment au niveau du langage Java.
La version 8 de Java a apporté beaucoup de nouveautés.
Il y'a eu un réel engouement pour Java 8. Preuve en est, d'après l'index TIOBE, Java est redevenu le langage de l'année en 2015.

## Java 9

Java 9 est la version la plus controversée de l'histoire de Java car le JCP a voté non pour la première fois.

Les principales fonctionnalités

- Le système de module JPMS (Java Platform Module System) issu du projet Jigsaw
- 89 JEPs : évolutions mineures, des API, des outils

Java 9 vient avec un nouveau modèle de releases : 2 releases majeures de Java par an via OpenJDK.
Java 11 arrive en septembre 2018.
Oracle propose un modèle LTS : 1 version tous les 3 ans à partir de Java 11.
Java 9 et 10 ne sont pas LTS.

**Les a priori**

- Classpath et module-path cohabitent
- Le code d'une application ne doit pas obligaoirement être modularisé
- La classe sun.misc.Unsafe n'est finalement pas retirée
- Une application Java 8 compile en Java 9 : çà dépend
- Une application Java 8 s'exécute en Java 9 : çà dépend

**Améliorations**

- Méthode private dans les interfaces
- Variables finales
- …

**API et outils**

- Process API
- Flow API
- Fabriques pour collection immuable
- Var Handles (remplace partiellement Unsafe)
- Enrichissement de CompletableFuture

- JavaDoc en HTML 5 avec recherche
- JShell

**Améliorations de la JVM**

- Globalement de meilleurs performances :
  - Compact String
  - Concaténation de String avec InvokeDynamic
  - G1 comme GC par défaut
  - Code compilé réparti dans du cache séparé
- Unification du logging de la JVM

**Les modules**

- Les modules sont perçus comme contraignants
  - Pas de split package
  - Pas d'accès par défaut aux classes publiques d'un module
  - Nécessité de définir les dépendances dans le module-info ET dans les outils de build (Maven)
- Avantages des modules :
  - Encapsulation forte
    - Amélioration de la sécurité et de la maintenabilité
  - Une configuration fiable
    - Réduit les problèmes de classpath hell. La JVM peut détecter les modules manquant (mais pas la version …)
  - Occasion de remettre de l'ordre
    - Dans le design des livrables
    - Dans les dépendances
    - Dans les API utilisées
- 4 types de modules
  - Platform modules : modules du JRE
  - Unnamed module : modules regroupant toutes les classes/JAR du classpath
  - App Named modules : jar modulaire dans le module path
  - Automatic modules : JAR non modulaire dans le module path

**Runtime personnalisé**

L'outil Java Linker (jlinker) permet de créer un JRE personnalisé à partir des modules utilisés par l'application.
Pour fonctionner, cet outil nécessite que tous les JARs doivent être modulaires, ce qui n'est pas pour tout de suite.

## Migration vers Java 9

**Faire un état des lieux**

Faire une cartographie des dépendances pour vérifier s'il existe une version modulaire.
Utilisation de l'outil Jdeps du JDK pour cartographier les dépendances.
Jdeps analyse statiquement le bytecode (travaille sur les .class et les .jar)
Permet également de détecter les split packages.

Jdeps permet de créer le module-info d'un JAR donné, avec possibilité de le retoucher manuellement après.

**Migration d'une bibliothèque**

2 stratégies :

1. Conversion en module d'un JAR
   - Ajout d'un fichier module-info
2. Sans conversion en module
   - Nécessité de figer son nom via l'attribut Automatic-Module-Name. Sinon, le nom est déterminé automatiquement à partir du nom du JAR.

Quel que soit la stratégie adoptée, bien tester avec classpath et module path. Raison : leurs comportements peut différer.

**Pourquoi ne pas rester en Java 4, 5, 6, 7 ou 8 ?**

- Plus de support d'Oracle
- Plus de mise à jour gratuites de JRE
- Ne permet pas toujours d'upgrader des librairies

Pourquoi ne pas migrer vers une autre techno ? Pour quel ROI ? Le cout de migration vers Java 9 restera bien inférieure.

Stratégie viable : attente la prochaine LTS

A l'avenir, il va falloir choisir entre le modèle gratuit ou le support payant

- Modèle LTS : support (non gratuit) longue durée proposée par Oracle

**Migrer vers les modules**

Solution cible : les modules.
Nécessité d'ajouter un module-info à chaque JAR.
Problème des dépendances et des dépendances transitives utilisées.

On peut migrer à Java 9 sans les modules.
Il est possible de n'utiliser que le classpath comme en Java 8

Solution intermédiaire : migration incrémentale.
Modulariser complètement ou partiellement l'application. Continuer à utiliser des bibliothèques non modulaires le temps qu'elles se mettent à jour.

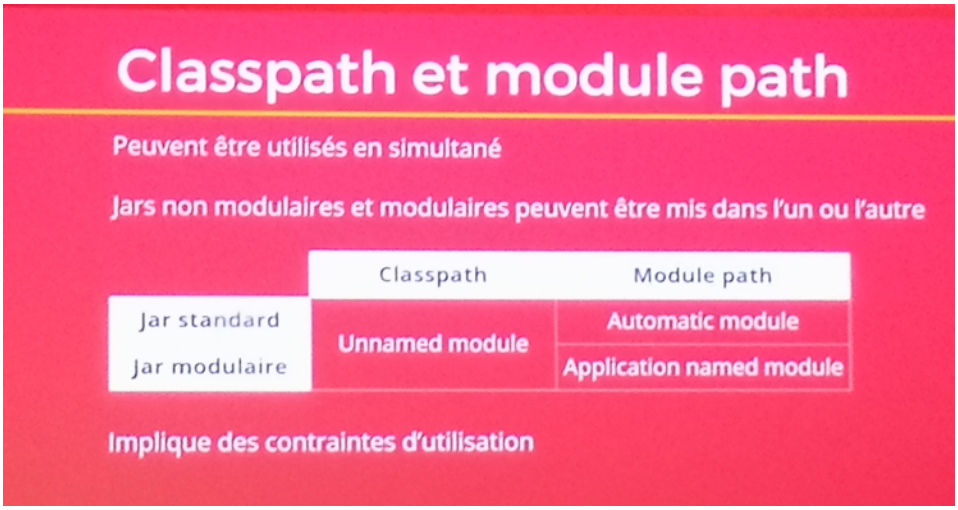
L'outil jar permet de savoir si un JAR est modulaire ou pas avec l'option --describe-module. Sur un JAR modulaire, il donne le nom automatique.

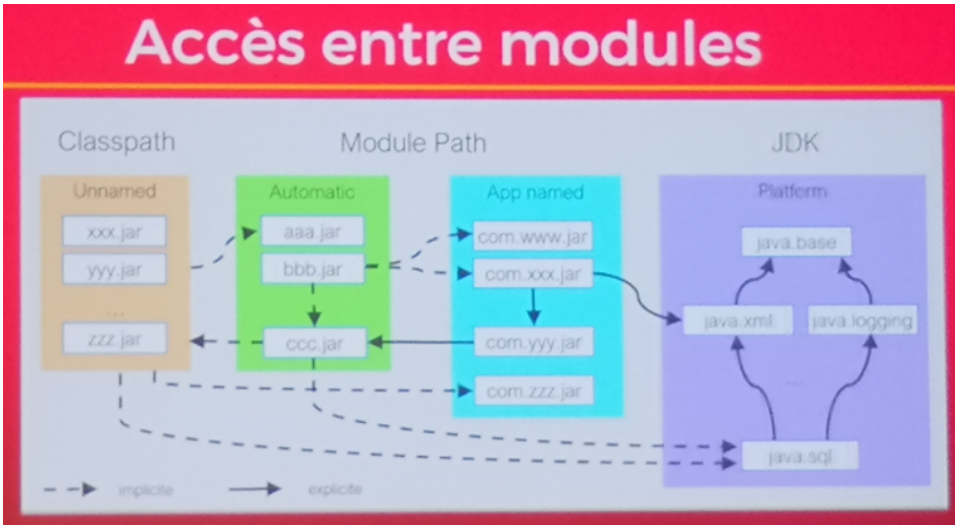
L'option --illegal-access permet d'autoriser ou non l'accès par introspection par le unnamed module. Par défaut, l'option est à permit (tous les accès sont autorisés). Fait perdre une partie des bénéfices de JPMS. A termes, la valeur par défaut sera passée à deny. Jean-Michel nous conseille de ne pas ignorer le warning.
4 valeurs possibles : warning, debug, deny, permit

Pour assouplir JPMS, il y'a 5 options au niveau de la JVM ou du compilateur :

1. –add-module
2. –add-reads
3. –add-exports
4. –add-open
5. –patch-module : permet d'ajouter des classes dans un module

Ces options ne modifient pas le module-info.
Attention à ne pas abuser de leur utilisation.




**Multi-Release JAR (MR JAR)**

Permet d'étendre le format JAR : inclure des classes spécifiques de .class pour d'autres versions de Java dans un même fichier jar
Un jar peut contenir plusieurs .class pour différentes versions

**Outil jdeprscan**

Analyse statique des .class pour rechercher les API deprecated
Java 11 va retirer des méthodes. Ce qui n'était pas le cas jusque-là.
L'application ne fonctionnera plus.

**Les incompatibilité**

L'identifiant _ n'est plus valide.
Les mécanismes d'endorsed et d'extension sont retirés.
Le JRE et le JDK ont désormais la même structure de répertoire.

**Les API internes**

La plupart des API internes sont encapsulées. Le développeur n'y a plus accès. Certaines sont remplacées (ex : sun.misc.BASE64Decoder par java.util.Base64). D'autres vont disparaitre (ex : Unsafe)
L'outil jdeps avec l'option --jdkinternals propose une solution de remplacement si elle existe.

**Les dépendances cycliques**

- Les JARS non modulaires peuvent avoir des dépendances cycliques
    - Pas une bonne pratique
    - Mais courant dans le classpath
- Il va être nécessaire de les éliminer.
- Ne fonctionne ni à la compilation ni au runtime

**Split packages**

- Un même package Java présent dans plusieurs JAR
- Courant dans le classpath
- Interdit par Java 9 pour fiabilisé la configuration
- S'applique pour les packages dans les modules des modules exportés ou NON

**Nouveau format de version**

- La version de Java est parfois utilisée pour de mauvaises raisons
- Le format de version change en Java 9
- Ça a rechangé un peu en Java 10
- Solutions : utiliser l'API Runtime.Version ou le MR JAR

**Les modules Java EE**

- Les modules Java EE ne sont plus chargés par défaut : JAXB, JAX-WS
- De préférence, utilisez une librairie externe car elles seront retirées de Java 11

## Java 10

Arrivé 6 mois après Java 9.
Seulement 12 JEPs.
A rendu Java 9 obsolète

L'inférence de type pour la déclaration de variables locales avec var
Attention : var n'est pas un mot clé
Code qui compile :
*var var = "Bonjour";*
Par contre, une classe appelée var ne compilera plus.

Préconisation : utiliser var avec discernement. Le code peut être plus ou moins lisible. Le nommage des variables est primordial.

Meilleur support de Docker :

- Option UseContainerSupport (Linux uniquement)

## Conclusion

- Java évolue, nous devons (devrons) suivre
- La migration vers JPMS et les modules sera obligatoire tôt ou tard
    - Elle va être délicate et longue
    - Plusieurs stratégies sont possibles
- Oracle propose un guide de migration vers Java 9