Préparez-vous à la modularité selon Java 9

Speaker : Alexis Hassler et Rémi Forax
Format : University
Date : 5 avril 2017
Slides : http://prez.sewatech.fr/devoxxfr17-uni/

Rémi a participé à la spécification sur les modules Java 9 (groupe d'expert).
Rémi a mis du code dans Java 9 (Open JDK) : utilisation du + entre des chaines de caractères.
Alexis a lu la doc et a interrogé Rémi. Jigsaw ayant débuté il y'a 10 ans, regarder les dates des documentations trouvées sur Internet : beaucoup de choses ont changé.
Java a plusieurs gros problèmes qui trainent depuis très longtemps. Initialement Jigsaw était prévu pour Java 7. Le problème principal est que la façon dont on déploie des applications a changé entre la création de Java en 1995 et aujourd'hui avec des outils de build Maven ou Gradle. L'autre problème à résoudre : tout est très ouvert (utilisation de l'introspection pour setter des propriétés privées).

Les outils de build résolvent des dépendances à la compilation (cf. photo arbre).
Pour la Machine Virtuelle, on a plein de JAR linéaires scannés les uns après les autres. En moyenne, une application Java comporte une centaine de JAR.
Problème lorsque 2 Jar de la même librairies sont dans le classpath : la JVM prend le JAR de la 1ière classe trouvée.
Une classe fille nouvellement introduite dans ASM 3.1 peut hériter d'une classe parent présente dans ASM 2.3.
Objectif de JigSaw : utiliser l'arbre (DAG) vu à la compilation lors du runtime.

Autre problème : taille du RT.jar : 66 MB (guava.jar 2.3 Mb et spring-context.jar de 1Mb).
Les classes de RT.jar ne sont pas signées : problèmes potentiels de sécurité.

L'un des objectifs de Jigsaw est de réduire la taille de la JVM.
Exploit CVE-2015-6420 sur un bug dans Commons Collection => impacte de nombreux serveurs d'applications (Websphere, JBoss, WebLogic, Jenkins). Ces derniers n'utilisent ni OSGI ni JBoss Modules. Commons Collection n'est pas accessible juste en passant le réseau.
Objectif de Jigsaw.

Modularisation du JDK

Dans Java 9, rt.jar et tools.jar disparaissent. Ils sont remplacés par des modules.
Gros module : **java.base**
Tous les modules ont des dépendances vers d'autres modules. Ils s'appuient tous sur java.base (qui contient le package java.lang).
Il y'a des modules présents mais marqués comme deprecated : java.activation, corba … Ces modules dépréciés sont tous apparus dans Java 6 et viennent du monde JavaEE. Ces modules dépréciés vont redescendre dans javaEE 9. Ils seront retirés dans Java 10.

Modules non-standards : sun.* et com.sun.*
Depuis 1996, il est précisé de ne pas utiliser les classes présentes dans ces modules.
Techniquement, avant java 9, la plateforme n'empêche pas de les utiliser.
Dans Java 9 : des classes ont été supprimées, des packages ne sont plus visibles.
Les modules commençant par jdk (ex : jdk.net, jdk.attach, jdk.httpserver) sont propres à une implémentation de JDK. Ces modules jdk.* pourront être utilisés dans d'autres versions de Java, de la même manière qu'un JAR avec module.

Descripteur de module

Classe module-info.java vu par le compilateur.
Chaque module dispose d'un Nom et d'une liste de dépendances.
Un module est un JAR (contient plusieurs packages) et un descripteur de modules module-info.java

La compilation ne change pas avec javac.
La création d'un module passe par l'utilisation de la commande jar.
L'exécution passe par l'utilisation du module-path. Le classpath existera toujours afin d'assurer une transition en douceur des frameworks. A termes, le classpath existera toujours car de vieilles librairies ne seront pas mises à jour.

java –module-path modules –classpath dependencies/dependency.jar –m app.main/com.app.Main

Exemple d'application

Application d'enregistrement d'image utilisant les classes du JDK (pas de lib externes).
Application Java 8 à migrer vers Java 9.

On commence par migrer module-info.java. Ajout de dépendances vers :

- ~~java.xml.bind~~ (JAXB est déprécié en Java 8, il faudra le tiré avec un outil de build à partir de Java 10)

Le compilateur javac accepte les double tirets -- (exemple -sourcepath en --source-path).

Une nouvelle version de JDK 9 sort toutes les semaines.
En Java 9, utilisation massive de @Deprecated.

A l'exécution, on spécifie uniquement le module-path
Java 9 casse lorsque plusieurs fois le même JAR sont présents dans le module-patH. Java 9 effectue des vérifications au démarrage.

Erreur au runtime : utilisation de Method.setAccessible. Remédiation : utilisation du **open** module dans le descripteur.

Une fois l'application migrée, on cherche à modulariser l'application.
Pas de convention de nommage sur le nom des modules.
Il va falloir spécifier ce que l'on exporte à l'extérieur. Utilisation du mot clé **exports**

Avec Java 9, on peut avoir plusieurs modules dans la même arborescence : src/main/java. Javac sait le faire. Les outils de build comme Maven et Graddle doivent évoluer. Les IDE également (notion de module différente dans IntelliJ). Par contre, il faut appeler autant de fois la commande jar qu'il y'a de modules.

The Jigsaw sandwich

Pourquoi Jigsaw a-t-il mis autant de temps à sortir ?
Présence d'outils de build avec leurs systèmes de modules à la compilation.
JBoss, JavaEE, OSGI disposent également de leurs propres systèmes de module.
On ne pouvait pas choisir une techno existante. Sinon la compatibilité avec les autres outils était rompue.
Ce n'est pas au JDK de résoudre les dépendances des modules. Cette tâche est déléguée aux outils de build et aux serveurs. Le JDK va simplement indiquer que cela ne va pas marcher s'il trouve 2 fois le même module.
La 1ière version de Jigsaw utilisait les classloaders. Mais plus aucune application ne fonctionnait.
Les modules Java n'ont pas de version. Plus exactement, ils peuvent en avoir un (--module-version=1.0), mais la JVM ne les exploite pas. Cette version est pratique pour debugger (présent dans la stacktrace). Les outils de build pourront exploiter la version, la JVM non.

Découpage en modules

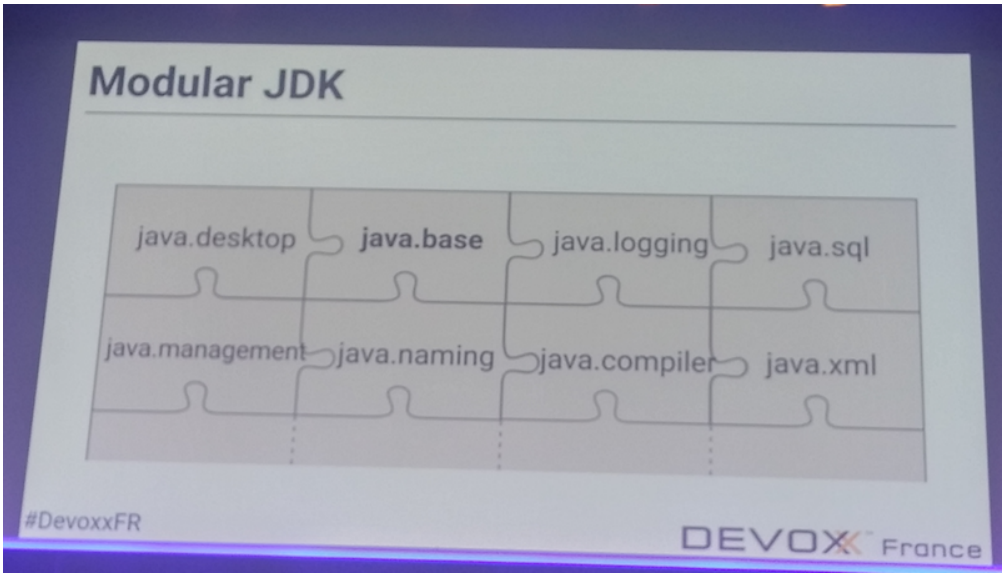
Dans l'application exemple, 4 modules ont été identifiés à partir du graphe de packages Java : image.backend, image.data, image.show et sw.common.

Lors du découpage du JDK, certains choix de répartition ont dû être faits. Exemple : classes de manipulation d'image dans **java.desktop**.

Javac n'indique pas si une dépendance ajoutée au descripteur de modules n'est pas utilisée.

Par défaut, lors de la création d'un module, rien n'est exporté vers l'extérieur. Une classe publique doit être exportée (ou plus exactement son package). L'utilisation de la syntaxe * est interdite.

Au run java, on précise la main class (précédemment précisée lors de la création du module avec jar).



Visibilité

L'**export** a un impact sur le niveau de visibilité des classes.

Un package ne peut pas être splitté sur plusieurs JAR. Un package Java doit forcément être dans un seul et unique module.

On peut exporter un package à une liste déterminée de modules. Sorte d'export restreint. exports com.app.internal to app.api

Application : le module image.backend est redécoupé en 3 modules : image.backend, image.inmem et image.db. Les modules image.inmem et image.db ne sont exportés que pour image.backend. La classe ImageInMemory (de image.inmem) n'est plus visible depuis image.show.

La classe sun.misc.Unsafe

Cette classe est tellement utilisée par de nombreux frameworks, qu'elle existe même sur Androïd. On ne pourra pas supprimer tout de suite sun.misc.Unsafe. Ce qui est regretable,

car cette classe ne profite pas des optimisations du JIT. D'après Rémi, pour les développeurs de librairies, l'usage de classe Unsafe est avant tout une question de simplicité. Mais il existe tout de même des cas où l'on ne peut pas s'en passer. Il faudra proposer des alternatives avant de pouvoir la supprimer.
En attendant, pour les développeurs d'API, La classe sun.misc.Unsafe a été mis dans un module à part : jdk.unsupported. En interne, le JDK continue à l'utiliser. Mais elle n'est exposée qu'à des modules bien ciblés.

Deep reflection

Deep reflection : changement de propriétés privées ou final.
Consiste à utiliser la méthode field.setAccessible(true). Le framework Hibernate l'utilise massivement.
Par défaut, setAccessible ne fonctionne plus. Pour que des classes d'un module puisse être utilisée par deep reflection à l'extérieur du module, il faut utiliser le mot clé **opens** dans le descripteur de modules (à la place de export).
Un intérêt est que le JIT sait à présent que des propriétés peuvent être considérées comme final.

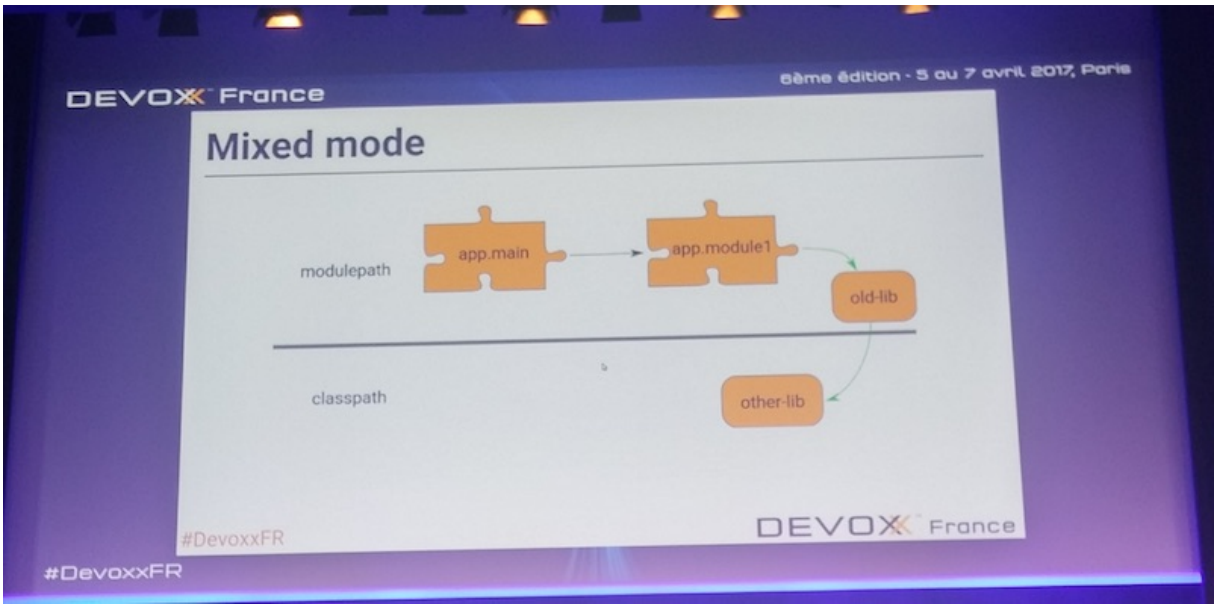
Autres formes de dépendances

Il existe d'autres formes de dépendances :

- **requires** : pas de transitivité (pour rappel, par défaut, Maven est transitif)
- **requires transitive** : permet d'ajouter de la transitivité mais uniquement de niveau 1.
- **requires static** : dépendance vérifiée à la compilation, mais pas au runtime (la dépendance est facultative)

Mixed mode

On ne peut pas faire de requires d'un JAR présent dans le classpath.
Les modules ne voient pas les JAR du classpath. Les JAR du classpath peuvent voir les modules.
Pour faciliter la transition, il a fallu trouver un moyen : on ajoute le JAR dans le modulepath. C'est ce qu'on appelle les **modules automatiques**. Un algorithme permet de trouver les JAR en retirant leur numéro de version. Cet artifice permet d'utiliser les JAR en attendant qu'ils soient transformés en modules. Cette fonctionnalité disparaitra peut-être en Java 10 (en fonction de son temps de développement).
A noter qu'un module automatique peut voir les JAR du classpath.
En discussion : Maven Central pourrait prendre tous les JAR et générer un nom de module. Cela permet d'avoir un nom stable.



Mise en pratique sur l'application exemple

On repart de l'application d'exemple.

Les modules sw.common et image.data sont désormais exposés en transitive depuis image.backend. Permet d'alléger un peu l'usage des requires.

On configure l'application pour utiliser H2 database. H2 est ajouté dans le modulepath.

Ensuite, on souhaite remplacer le logger maison par SLF4J. Dans le module-info.java, on ajoute le nom du module : slf4j.api (déduit à partir de slf4j-api-1.7.25.jar).
SLF4J a besoin d'une implémentation pour fonctionner (et non uniquement de l'API).
SLF4J API étant un module automatique, il n'importe pas le slf4J-simple-1.7.25.jar. Nécessité de passer en mode mixte avec un classpath

Message : on n'est pas obligé d'attendre que toutes les librairies deviennent des modules afin de pouvoir utiliser des modules.

Le système de service

Introduit dans Java 6, la classe java.util.ServiceLoader permet de charger des implémentations à partir d'une interface. Mécanisme très light d'injection de dépendances. Toutes les injections sont décrites dans le META-INF/services.

Dans l'application, le module image-backend ne tire plus image.inmem et image.db.
Dans image.db/module-info.java, on ajoute :
provides fr.sw.fwk.dao.DAO with fr.sw.img.db.ImageDB
Le ServiceLoader est désormais utilisé dans ImageService. La 1ière implémentation de l'interface DAO est utilisée. Lorsqu'on lance plusieurs fois l'appli, on n'obtient pas le même ordre.

Migrer vers Java 9

Pour la 1ière fois, pas de compatibilité ascendante totale. N'importe quelle grosse application ne pourra pas être migrée sans changement.
Comment utiliser dans Java 9 des librairies écrites en Java 8 ? Jigsaw permet de débrayer certaines restrictions pour faciliter l'adoption de Java 9.

Option de la JVM : **--permit-illegal-access** ajouté en Java 9 mais retiré en Java 10. La JVM va logger tous les accès. Cela permet de savoir ce qui va être corrigé.

Sans utiliser la modularisation Java, il va falloir spécifier les modules dépréciés au compilateur. Plusieurs lignes de commandes pour java et javac : --add-reads, --add-exports, --add-opens. L'application continuera à fonctionner sans warning.

Outil d'analyse statique jdeps : la commande « **jdeps –jdk-internals** » sort un rapport contenant la liste des dépendances vers des classes du JDK internes et donne des solutions. Commande d'ores et déjà disponible sur Java 8. Ne fonctionne pas sur .setAccessible() qui nécessite une analyse dynamique.

Tests

Comment tester unitaire une application ?
Les classes de tests sont généralement placées dans le même package que la classe testée mais dans des répertoires différents : src/main/java vs src/test/java. Problème : c'est désormais interdit par la modularisation Java 9.

Possibilité d'utiliser un autre package pour la classe de TU avec JUnit en module automatique et hamcrest dans le classpath.

L'autre solution plus couramment utilisée avec utilisation du même package Java. Nécessité

de créer un seul module. Solution simple : créer un JAR avec code de prod + TU. Pas propre. Au runtime, il est nécessaire d'indiquer à javac et java que les classes de tests doivent avoir accès à JUnit. Cela revient à créer un module contenant des classes se trouvant dans des répertoires différents. On doit injecter les classes de tests dans le module à tester. A la compilation, utilisation de « **--patch-module** app.module1=src/test/java » et « **--add-reads** app.module1=junit », « **--add-modules** junit ». Ligne de commande encore plus verbeuse pour l'exécution des tests. Très laborieux.

Heureusement pour nous, Maven supporte dès à présent Java 9 et connaît toutes ces options de ligne de commande.

Intérêt d'une application modulaire

Change la façon de concevoir une application. L'application devient fermée. On devient capable de prendre que ce qui nous intéresse et retirer tout le reste. Autre possibilité : pré-compiler le code, ce qui permet d'accélérer le démarrage des applications.

Nouvel outil : **jlink**. Permet de créer son propre JDK (ex : sur Rasberry PI ou pour Docker). Intéressant également pour éviter d'avoir des failles de sécurité sur des modules non utilisés. Jlink va créer des jmod. Un jmod est similaire au jar à l'exception que l'on peut ajouter du code natif (.so, .dll). jlink refuse les modules automatiques à cause du classpath. Possibilité d'utiliser la VM minimale de 3 Mo. Au moment où l'image est créée, la vérification des modules est effectuée. Cette étape n'est donc plus réalisée au démarrage de la JVM.

Nouvel outil expérimental : **jaotc** (Ahead of Time compilation). Le code va moins vite, mais reproductible car pas d'optimisation du JIT. Option de la JVM : –XX :+UseAOT

Le JDK 9 contient les classes en double : avec .jmod et sans. D'où l'augmentation de la taille du JDK entre JDK 8 et 9 (360 Mo vs 494 Mo).

Resources :

- https://cfp.devoxx.fr/2017/talk/KHV-4031/Preparez-vous_a_la_modularite_selon_Java_9