



## Introduction au développement par modules Java (JDK 9+)

**Intervenant :** Chouki TIBERMACHINE

**Bureau :** LIRMM (E.311)

**Tél. :** 04.67.14.97.24

**Mél. :** Chouki.Tibermachine@lirmm.fr

**Web :** <http://www.lirmm.fr/~tibermacin/ens/hmin304/>

# Plan du cours

- Introduction : limites de Java 8 et <
- Généralités sur les modules Java et le JDK modulaire
- Créer ses propres modules
- Fournir et utiliser des services dans Java

# Plan du cours

- Introduction : limites de Java 8 et <
- Généralités sur les modules Java et le JDK modulaire
- Créer ses propres modules
- Fournir et utiliser des services dans Java

# Réutilisation en Java

- L'unité de réutilisation théorique en Java est la classe
- Mais, en pratique, l'unité de réutilisation est l'unité de release (principe du génie logiciel)
- L'unité de réutilisation en pratique est donc un **ensemble** de classes organisées en **plusieurs** packages (et empaquetées dans une archive JAR)

# Limites de la réutilisation en Java -1-

- Quand on réutilise des programmes, on doit gérer beaucoup de packages qui proviennent de JAR différents
- Gestion complexe des dépendances : *Classpath (JAR) Hell*
  - Dépendances non explicites : un JAR ne dit pas de quel(s) autre(s) JAR(s) il dépend
  - Dépendances transitives
- Des outils existent pour assister la gestion des dépendances : Maven ou Gradle, par exemple
- Mais ce sont des outils statiques et externes à l'application :
  - Dépendances explicitées en dehors de la définition des composants de l'application
  - A l'exécution, la notion de composant/dépendance disparaît

## Limites de la réutilisation en Java -2-

- A l'exécution, les frontières des JARs disparaissent (leurs contenus sont « fusionnés »)
- Impossible d'avoir des versions parallèles d'une même classe (classes avec le même nom qualifié complet, mais appartenant à des versions différentes d'une même librairie, par ex.) :
  - Cas fréquent dans les grosses applis
  - La première chargée par le class loader est celle qui sera utilisée (le class loader Java respecte l'ordre précisé dans le classpath)
  - Cette classe peut être incompatible (l'usage fait de sa « jumelle », qui a été masquée, peut ne pas correspondre partout)

## Limites de la réutilisation en Java -3-

- Modificateurs d'accès possibles dans Java :
  - *private* : permet l'encapsulation des données, mais trop restrictif pour la réutilisation de code
  - *protected* : privé « relâché », destiné à l'héritage
  - par défaut (package) : le type est lié au package (lié aux autres types qui partagent l'**espace de nom**). Il n'est pas destiné à être utilisé en dehors de ça
  - *public* : seul moyen de partager des types entre packages, mais : le type devient accessible à tout programme extérieur
- Ce dernier cas implique des :
  - Problèmes de conception d'API (types destinés à être utilisés par d'autres types « amis », qui ne partagent pas le même package)
  - Problèmes potentiels de sécurité (surface d'attaque plus large)

## Limites de la réutilisation en Java -4-

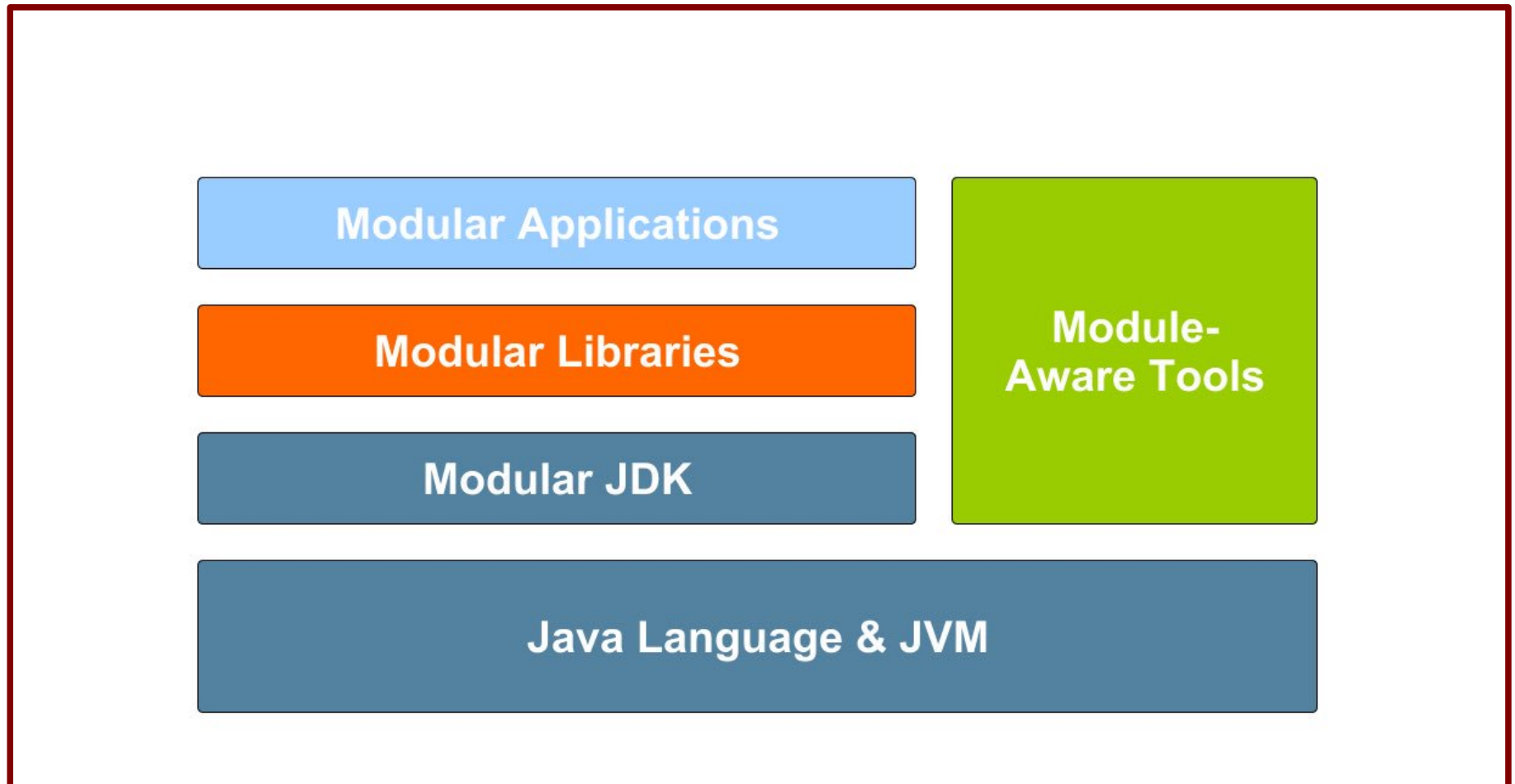
- Utiliser la librairie standard de Java dans votre code implique la présence à l'exécution d'un bloc monolithique de plus de 200 packages, qui composent l'implémentation de Java SE :
  - Le runtime JAR : rt.jar = ~64 Mo
- Même si votre application n'utilise pas tous ces packages, elle doit cohabiter avec
- C'est contraignant pour certains types d'applications (Web par exemple, qui n'utilisent ni AWT, ni Swing, ni ...) et dans certains environnements dédiés (conteneurs Cloud et environnements mobiles ou embarqués)



# Plan du cours

- Introduction : limites de Java 8 et <
- Généralités sur les modules Java et le JDK modulaire
- Créer ses propres modules
- Fournir et utiliser des services dans Java

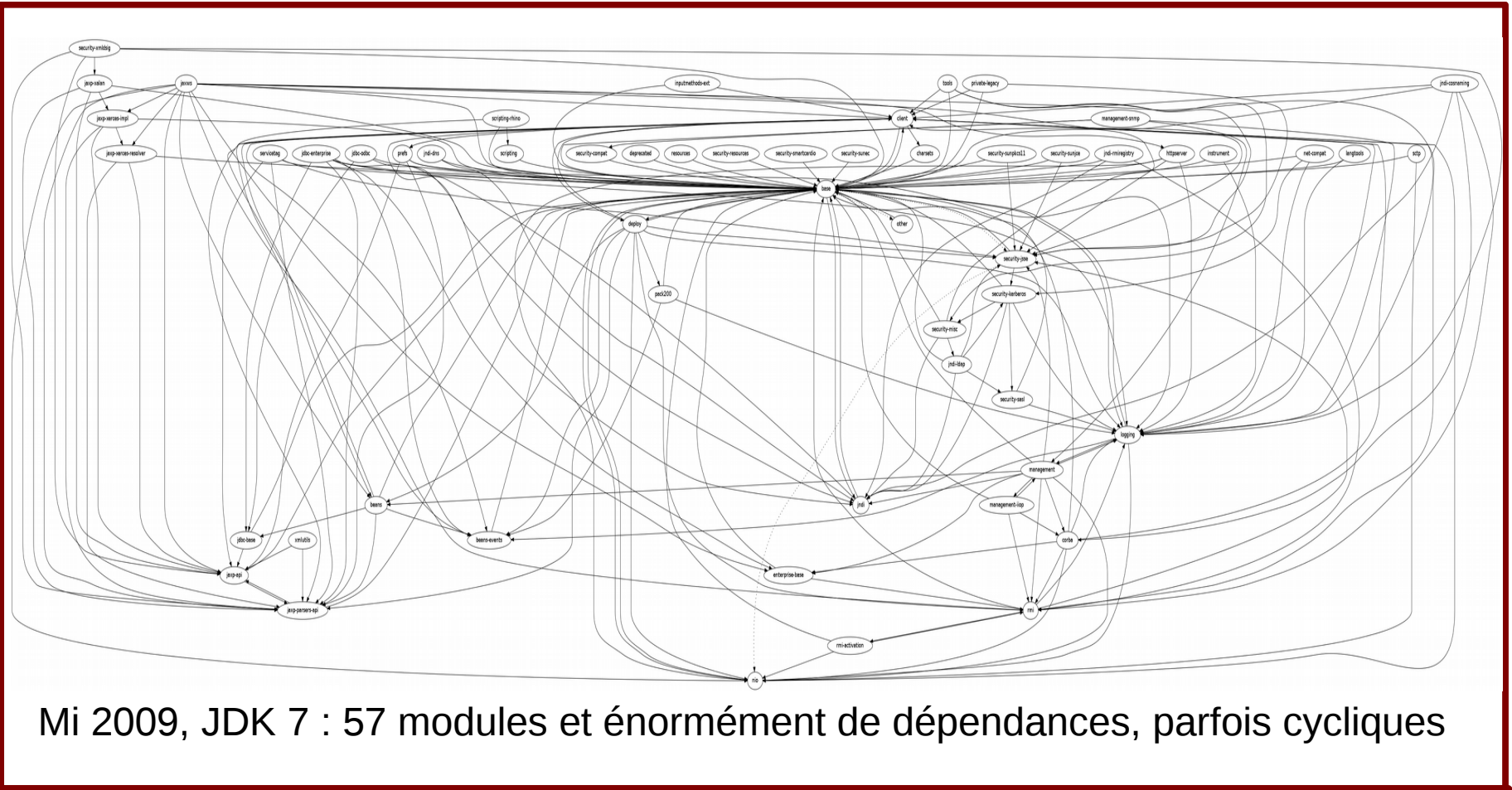
# Dans le JDK 9, un système de modules à tous les niveaux



# Module Java

- Un module est un ensemble de packages groupés au sein d'une unité destinée à la réutilisation
- Les packages qui ne sont pas destinés à la réutilisation sont masqués à l'intérieur du module
- Les programmes sont désormais (dans JDK9+) un ensemble de modules (qui contiennent un ensemble de packages/classes)
- Même la librairie standard Java a été restructurée en modules

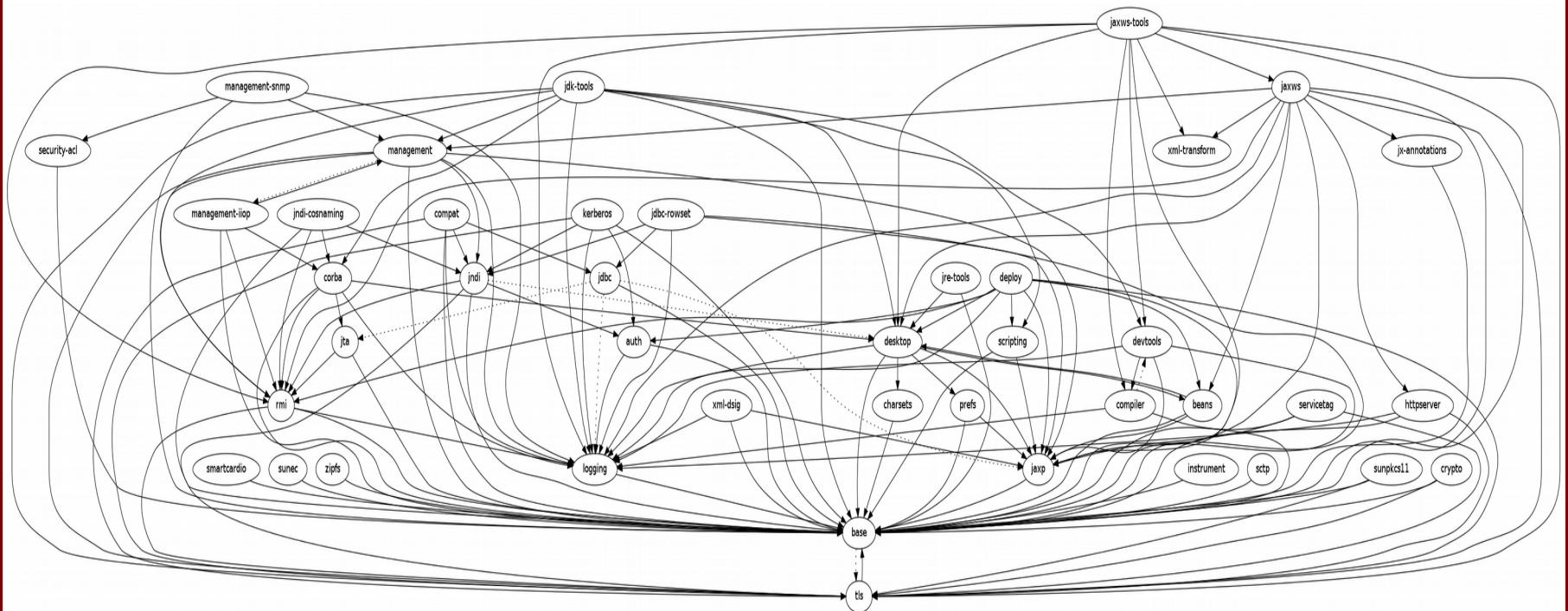
# Vue d'ensemble du JDK organisé en modules



# Vue d'ensemble du JDK organisé en modules

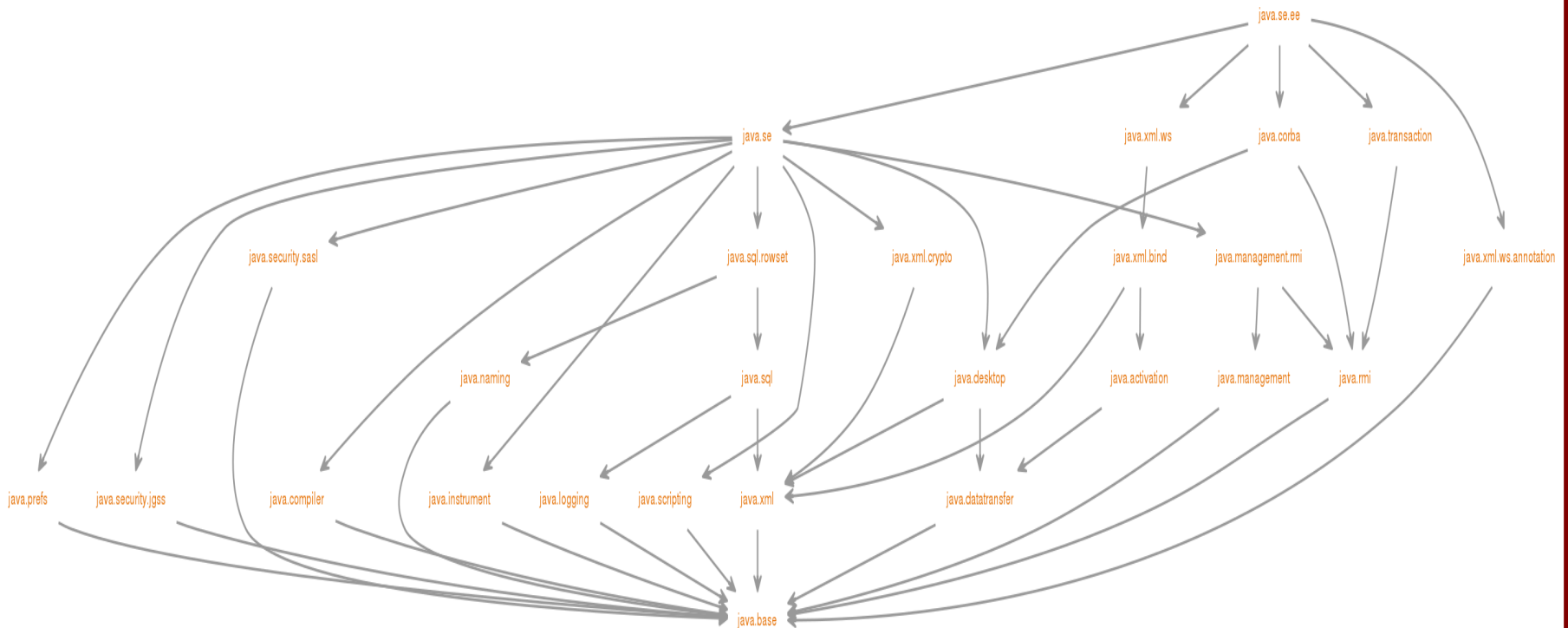
Après un gros travail de modularisation :

<http://cr.openjdk.java.net/~mchung/jigsaw/modularization-bugs.html>



JDK 8 : 44 modules (26 java SE et 18 outils JDK) et 134 dépendances

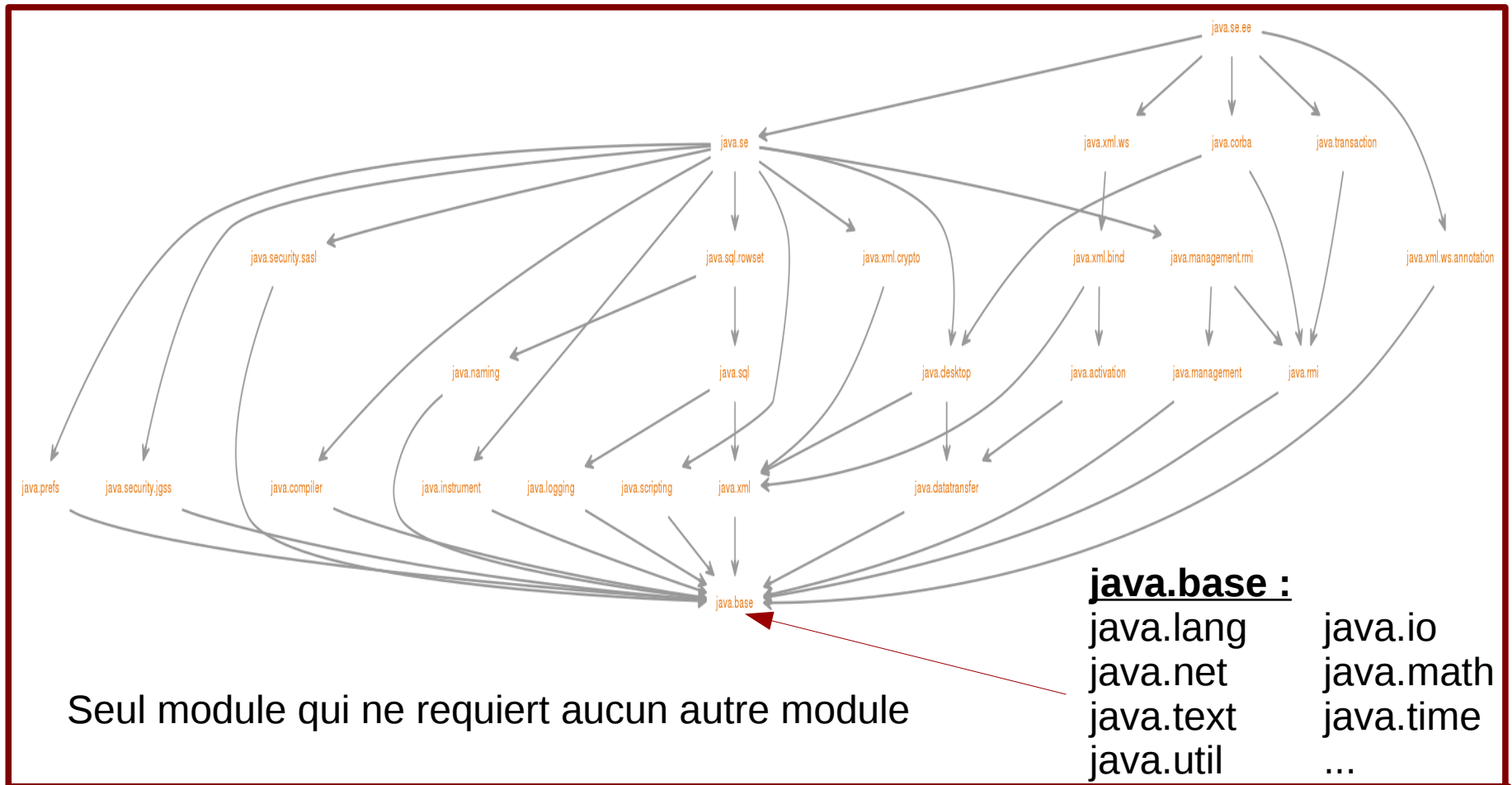
# Vue d'ensemble du JDK organisé en modules



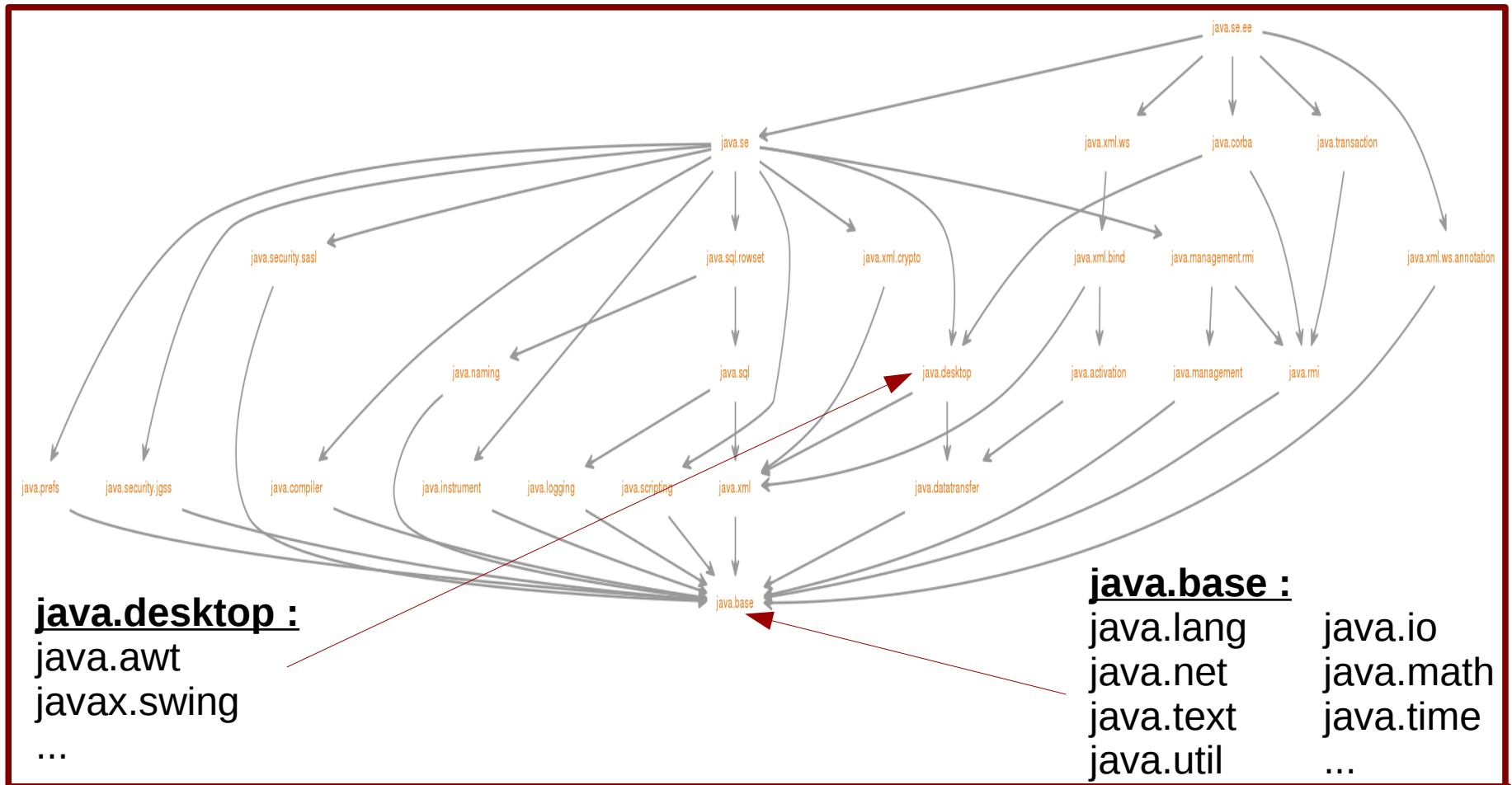
JDK 9 : les 26 modules de Java SE (java.\*)

Commande pour lister les modules : `java --list-modules`

# Vue d'ensemble du JDK organisé en modules

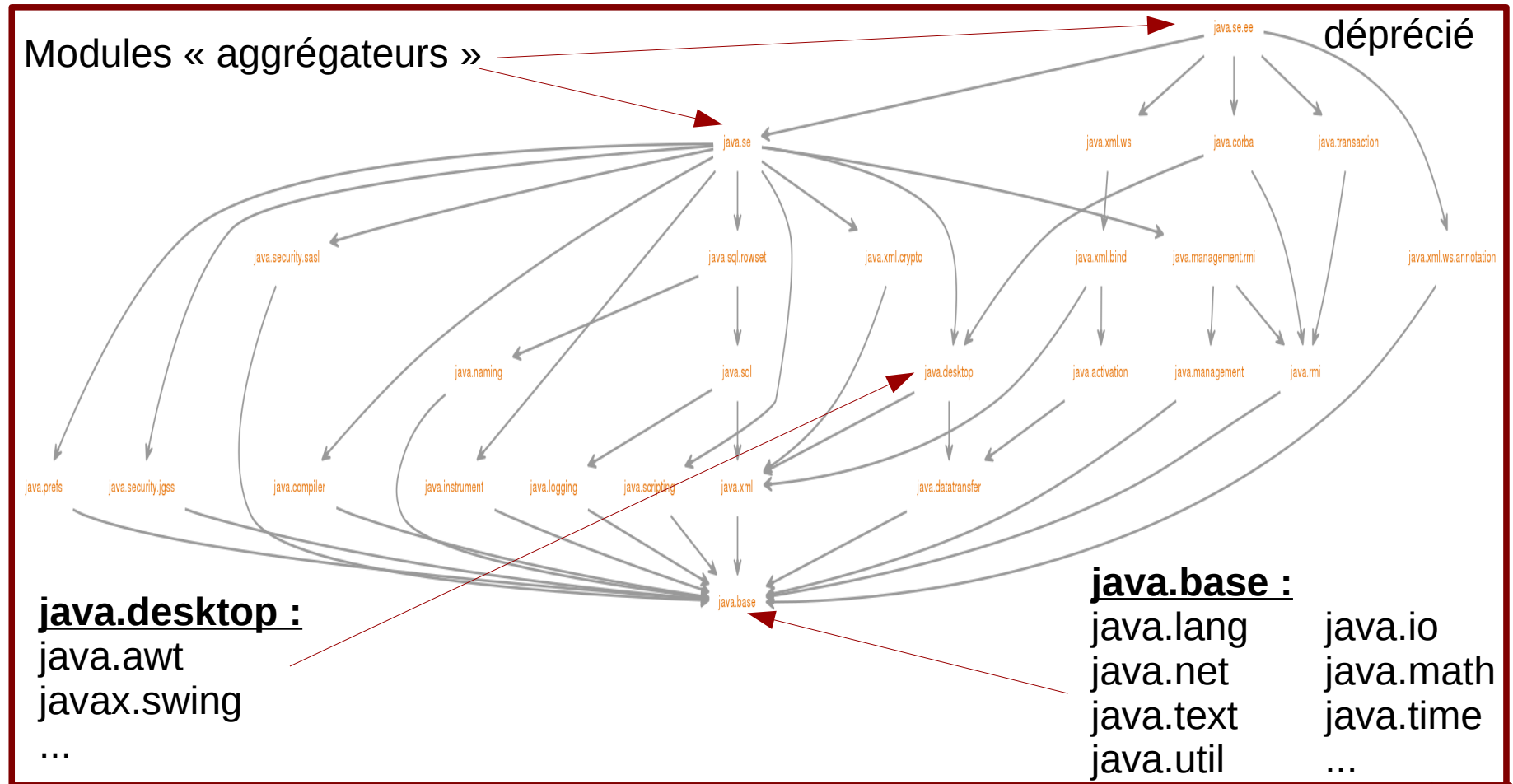


# Vue d'ensemble du JDK organisé en modules





# Vue d'ensemble du JDK organisé en modules



# Descripteur de module Java

- Un fichier module-info.java

- Exemple :

```
module java.prefs {  
    requires java.xml;  
    exports java.util.prefs;  
}
```

- Nom d'un module = identifiant unique (notation DNS inversée)  
com.uneEntreprise.unProjet.unModule

# Descripteur de module Java

- Un fichier module-info.java

- Exemple :

```
module java.prefs {  
    requires java.xml;  
    exports java.util.prefs;  
}
```

**Module** requis

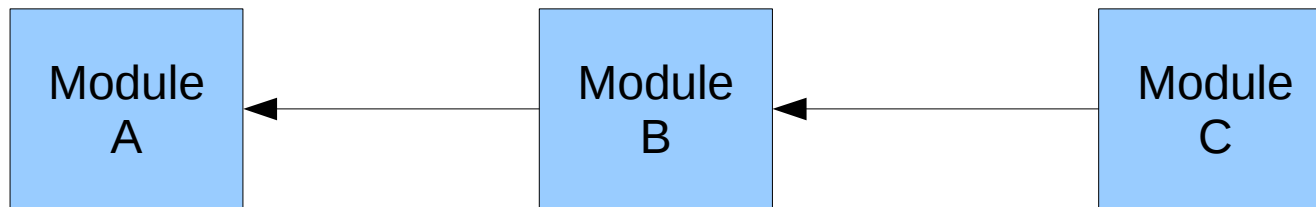
**Package** fourni/exporté  
(tout autre package dans  
le module est masqué)

- Nom d'un module = identifiant unique (notation DNS inversée)  
com.uneEntreprise.unProjet.unModule

# Sémantique changée pour le modificateur d'accès *public*

- Avant JDK9, un type (interface ou classe) publique est accessible pour tout autre type
- Avec le système de modules, un type publique est accessible pour tout autre type se trouvant à l'intérieur du même module :
  - Si son package est exporté par le module, il devient accessible partout (encapsulation renforcée si package non-exporté)

# Dépendances non transitives, par défaut



- Le module C, qui requiert B, n'a pas accès aux types qui se trouvent dans les packages exportés par A (requis par B)

## Dépendances non transitives, par défaut -suite-

- Par contre, il est possible parfois qu'un module, qui déclare exporter un type, voudra également exporter un type qu'il requiert d'un autre module et qui est utilisé par le type qu'il exporte

- Exemple :

Interface IA exportée par un module A déclare une méthode qui retourne un type TB exporté par un module B

// Dans un module A :

```
interface IA { public TB m() ; }
```

Un module C qui utilise (requiert) A est censé requérir B aussi pour pouvoir se servir de IA

# Dépendances transitives explicites

- Solution : dépendances transitives explicites  
module A { requires **transitive** B ; }
- Cela veut dire que le module A requiert B, et en quelque sorte « ré-exporte » les packages exportés par B aux autres modules qui requièrent A
- Le module « aggregateur » java.se comporte uniquement des clauses *requires transitive*

# Exports qualifiés

- Parfois, il est utile d'exporter des packages dans un module pour certains modules « amis » seulement

Exemple :

```
module java.xml {  
    ...  
    exports com.sun.xml.internal.stream.writers  
        to java.xml.ws ;  
    ...  
}
```

- Aucun autre module ne peut utiliser le package exporté
- Possibilité de mettre plusieurs noms de modules séparés par ,
- Solution déconseillée, sauf exceptions, par rapport à la modularité



# Résolution de modules

- *Module Path* : (autre chose que le classpath !!!)  
une liste de modules utilisée par le compilateur et la JVM  
pour résoudre les dépendances entre modules
- La résolution de modules suit un processus simple :
  1. charger le module racine (celui de l'application à compiler/exécuter)
  2. charger les modules qu'il requiert (si ce n'est pas déjà fait)
  3. refaire l'étape 2 pour chacun des modules chargésComme les dépendances entre modules ne sont pas cycliques  
ce processus se termine toujours
- Le chargement de modules est effectué à partir des indications  
dans le *module path*

# Rétro-compatibilité du système de modules

- Toute application écrite sans le système de module continue de fonctionner dans JDK9
- Raison : les classes sont compilées dans un module sans nom (*unnamed module*) qui requiert tous les autres modules  
Le compilateur et la JVM continuent d'utiliser le classpath
- Pour développer une nouvelle application, il est vivement conseillé d'utiliser le système de modules : rétro-compatibilité non-garantie sur le long terme

## Et par rapport à OSGi ?

- Les deux systèmes ont le même objectif : offrir une meilleure modularité dans la construction d'applications Java
- Différences : (qui font qu'OSGi continuera d'exister encore qlq temps)
  - Un bundle OSGi importe des packages (et non requiert des modules) : une meilleure modularité
  - OSGi offre des possibilités de gestion de versions au niveau des packages et des bundles (avec intervalles de versions)
  - Chargement dynamique des bundles OSGi (start-stop-update-...) avec possibilité de mettre en place des callbacks (cycle de vie)
  - OSGi offre un système de services dynamiques (annuaire central, des frameworks comme *declarative services*, ...)
- Privilégier OSGi dans les systèmes dynamiques (embarqués, ...)

# Plan du cours

- Introduction : limites de Java 8 et <
- Généralités sur les modules Java et le JDK modulaire
- Créer ses propres modules
- Fournir et utiliser des services dans Java

# Un premier module

- Une classe dans un package : HelloWorld.java

```
package hmin304.jdk9.cours.helloworld;
public class HelloWorld {
    public static void main(String... args) {
        System.out.println("Hello Modular World!");
    }
}
```
- Un descripteur de module : module-info.java

```
module helloworld {
}
```
- Sous le répertoire src : un répertoire additionnel pour le module **helloworld**  
hmin304/jdk9/cours/helloworld/HelloWorld.java  
helloworld/module-info.java  
(Le nom de ce répertoire additionnel = Le nom du module dans module-info.java)

# Nommer un module

- Le nom d'un module doit être unique dans une application
- Utiliser des noms courts pour des modules dans une application
- Si un module est publié comme librairie, il est important d'utiliser un nom qui ne doit pas rentrer en conflit avec les noms d'autres modules connus : notation DNS inversée par exemple
- Les mots *module*, *requires*, *exports*, ... sont réservés, mais uniquement dans les descripteurs de modules  
Ils peuvent être utilisés comme identifiants dans les classes

# Compiler ce premier module

- Compilation sans outils de build :  
javac -d out/helloworld  
    src/helloworld/hmin304/jdk9/cours/helloworld/HelloWorld.java  
    src/helloworld/module-info.java  
→ Compiler le descripteur de module aussi (cela déclenche la compilation en mode module)
- Ceci produit une structure de répertoires similaire à celle des sources avec des .class à l'intérieur (HelloWorld.class et module-info.class)

## Empaqueter ce premier module (dans un JAR modulaire)

- Créer une archive JAR avec le contenu du module en incluant module-info.class :

```
jar -cfe mods/helloworld.jar  
    hmin304.jdk9.cours.helloworld.HelloWorld  
-C out/helloworld .
```

Il faudra d'abord créer un répertoire mods

L'option e précise qu'on va indiquer la classe qui constitue le point d'entrée au JAR (HelloWorld)



## Exécuter ce premier module

- Exécuter la version non-empaquetée :

```
java --module-path out
```

```
--module helloworld/hmin304.jdk9.cours.helloworld.HelloWorld
```

ou bien :

```
java -p out
```

```
-m helloworld/hmin304.jdk9.cours.helloworld.HelloWorld
```

Nom du module

Nom qualifié complet de la classe à exécuter

- Exécuter la version empaquetée : utiliser le JAR modulaire

```
java --module-path mods --module helloworld
```

ou bien :

```
java -p mods -m helloworld
```

← Répertoire (on peut aussi mettre un JAR modulaire ou plusieurs valeurs séparées par « : » -Linux/Mac OS- ou « ; » -Windows)

- helloworld est le module racine pour la résolution de modules

## Lier des modules dans une *image*

- Dans l'exemple précédent, la JVM fait une résolution pour 2 modules seulement (helloworld et java.base)
- Grâce aux « *images* », on peut produire une distribution spéciale de l'environnement d'exécution, qui ne comporte que les modules nécessaires à l'exécution de l'application
- Très utile pour les environnements contraints (à ressources limitées) comme les systèmes embarqués ou les containers dans le Cloud
- Pour produire cette image, on utilise un outil du JDK qui s'intercale entre la compilation et l'exécution : jlink

## Lier des modules dans une *image*

- Pour l'exemple précédent :  
jlink --module-path mods/:\$JAVA\_HOME/jmods  
--add-modules helloworld  
--launcher hello=helloworld  
--output helloworld-image

- Cela produit :

```
helloworld-image
├── bin
│   ├── hello
│   ├── java
│   └── keytool
├── conf
│   └── ...
├── include
│   └── ...
├── legal
│   └── ...
├── lib
│   └── ...
└── release
```

Un script pour exécuter  
l'application

# Construire une application multi-modules

- Pour compiler :

```
javac -d out --module-source-path src -m module.racine.appli
```

↑  
Répertoire destination  
obligatoire (modules  
non-empaquetés)

↑  
Répertoire où se trouvent  
les sources des modules

↑  
Module racine  
(compilé en premier)

- Le compilateur lit les descripteurs de modules pour résoudre les dépendances, en commençant par celui du module racine

# Construire une application multi-modules -suite-

- Erreurs possibles si non-définition explicite des dépendances :
  - Si *requires* est manquant :
  - `java.lang.NoClassDefFoundError` si classe d'un module applicatif  
ou bien `package xxx is not visible`

```
src/gui.printer/printer/impl/GraphicalPrinter.java:5: error: package javafx.scene
.control is not visible
import javafx.scene.control.Alert;
                        ^
   (package javafx.scene.control is declared in module javafx.controls, but module
gui.printer does not read it)
```

si module de plate-forme (librairie standard)

- Si *exports* est manquant, `java.lang.IllegalAccessError`

```
Exception in thread "main" java.lang.IllegalAccessError: class helloer.Helloer (in
 module my.helloer) cannot access class printer.PrinterFactory (in module my.print
er) because module my.printer does not export printer to module my.helloer
    at my.helloer/helloer.Helloer.main(Helloer.java:8)
```

# Dépendances cycliques

- Si les modules déclarent des dépendances cycliques, erreur à la compilation (résolution statique) :

```
src/my.printer/module-info.java:3: error: cyclic dependence involving my.helloer  
requires my.helloer;  
           ^
```

- Souvent les dépendances cycliques sont indirectes : causées par des dépendances transitives qui impliquent beaucoup de modules (cas difficiles à détecter à la main)

# Dépendances « presque » cycliques (dynamiques)

- Lorsqu'on déclare dans un module l'export d'un package vers un module x (export mon.package to x;) et requérir x (requires x;)

- Exemple :

```
module gui.printer {  
    exports printer;  
(*) → exports printer.impl to javafx.graphics;  
    requires javafx.graphics;  
    requires javafx.controls;  
}
```

JavaFX instancie la classe  
GraphicalPrinter par *réflexion*  
alors que cette classe se trouve  
dans un package masqué (impl).  
D'où l'export **qualifié** (\*)

```
package printer.impl;  
  
import printer.IPrinter;  
  
import javafx.scene.control.Alert;  
import javafx.scene.control.Alert.AlertType;  
import javafx.application.Application;  
import javafx.stage.Stage;  
  
public class GraphicalPrinter extends Application implements IPrinter {  
    private static String msgToPrint;  
    public void print(String msg) {  
        msgToPrint = msg;  
        Application.launch(msg);  
    }  
    public void start(Stage stage) {  
        Alert alert = new Alert(AlertType.INFORMATION);  
        alert.setTitle("Information Dialog");  
        alert.setHeaderText(null);  
        alert.setContentText(msgToPrint);  
        alert.showAndWait();  
    }  
}
```

# Identifier dans quel module se trouve un package

```
package printer.impl;

import printer.IPrinter;

import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.application.Application;
import javafx.stage.Stage;

public class GraphicalPrinter extends Application implements IPrinter {
    private static String msgToPrint;
    public void print(String msg) {
        msgToPrint = msg;
        Application.launch(msg);
    }
    public void start(Stage stage) {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle("Information Dialog");
        alert.setHeaderText(null);
        alert.setContentText(msgToPrint);
        alert.showAndWait();
    }
}
```

```
module gui.printer {
    exports printer;
    exports printer.impl to javafx.graphics;
    requires javafx.controls;
    requires javafx.graphics;
}
```

OVERVIEW	MODULE	PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV CLASS	NEXT CLASS	FRAMES	NO FRAMES	ALL CLASSES				
SUMMARY: NESTED   FIELD   CONSTR   METHOD		DETAIL: FIELD   CONSTR   METHOD						

Module **javafx.graphics**  
Package **javafx.application**  
Class **Application**

- La Javadoc des classes permet de savoir quel module requérir  
<https://docs.oracle.com/javase/9/docs/api/javafx/application/Application.html>
- Sinon, lister les modules puis afficher leur descripteur :  
java --list-modules  
java --describe-module javafx.controls



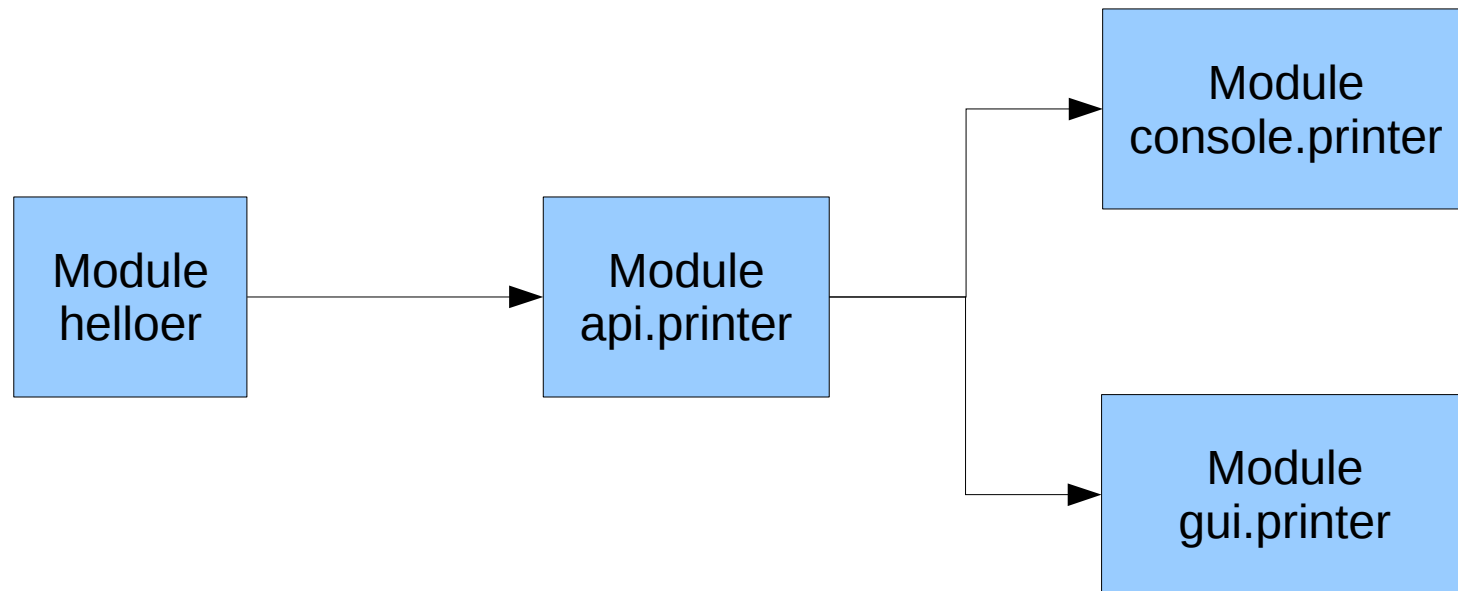
# Plan du cours

- Introduction : limites de Java 8 et <
- Généralités sur les modules Java et le JDK modulaire
- Créer ses propres modules
- Fournir et utiliser des services dans Java

# Organisation d'une API

- Reprenons notre exemple de helloer et printer et supposons que l'on dispose de **deux** printers (console.printer et gui.printer)
- Les deux printers fournissent l'interface IPrinter
- Problème : où mettre l'interface IPrinter ? (.java/.class)
  - Dans chacun des deux modules fournisseurs (console.printer et gui.printer) ?  
Duplication de code (mauvaise solution)
  - Dans le module helloer ? (client de l'interface)  
On sera obligé de déclarer une dépendance (requires) entre les deux printers et le module helloer parce que les classes d'implémentation importent l'interface (couplage indésirable)
  - Solution : dans un module à part (dédié à l'interface) - module API

# Organisation d'une API



# Utiliser le patron Factory pour « connecter » les modules

- Définir une classe Factory, qui crée des instances d'objets qui implémentent IPrinter :

```
public class PrinterFactory {  
    public static List<String> getSupportedPrinters() {  
        return List.of(ConsolePrinter.NAME, GraphicalPrinter.NAME);  
    }  
  
    public static IPrinter newInstance(String name){  
        switch(name) {  
            case ConsolePrinter.NAME : return new ConsolePrinter();  
            case GraphicalPrinter.NAME : return new GraphicalPrinter();  
            default : throw new IllegalArgumentException("Type  
                de Printer inconnu !");  
        }  
    }  
}
```

# Patron Factory et ses limites : cas d'API & +ieurs implems

- Problème : où mettre la classe Factory ?
    - Dans chaque module fournisseur d'API : mauvaise solution (duplication de code)
    - Dans un module dédié (factory) ou celui de l'API : problème de dépendances entre le module factory/API et les modules fournisseurs d'API qui doivent exporter les classes d'implémentation (mauvaise conception des modules : dépendances statiques entre l'API et ses implémentations)
- Tout ce que l'on a donc fait, c'est d'ajouter une couche d'indirection supplémentaire (complexité additionnelle)  
Malgré que l'on a gagné le fait que le module client (helloer) est découplé des implémentations de printer (mais le couplage existe plus loin : entre le factory et les implémentations)

## Solution : utiliser les services

- Idée de base : avoir un module qui requiert une API qui peut être implémentée par n'importe quel(le) classe/module
- Ce mécanisme permet d'éliminer le couplage statique (à la compilation) entre des clients d'API et ses fournisseurs
- Seul problème : il est un peu intrusif (dans le code, il faut passer par une API dédiée -ServiceLoader- pour utiliser ce mécanisme)

# Fournir un service

- Déclarer dans le **descripteur de module** (module-info.java) le fait de fournir un service (clause *provides-with*)
  - Rien à changer dans le code (mécanisme non-intrusif de ce côté)
- Exemple :

```
module console.printer {  
    requires api.printer;  
  
    provides api.printer.IPrinter  
        with console.printer.ConsolePrinter;  
}
```

Le module API  
où se trouve l'interface  
du service

L'interface du service

L'implémentation du service
- Remarquez l'absence d'un export du package de la classe d'implémentation du service (encapsulation)

## Fournir un service -suite-

- On fait la même chose pour le module gui.printer
- Le système de modules instancie automatiquement les classe d'implémentation du service et rend ces instances disponibles pour les clients du service
- Tout autre module (client) peut utiliser le service, sans qu'il n'y ait de dépendances statiques entre lui et les modules fournisseurs de service (seule dépendance = l'interface du service)



# Consommer un service

- Il suffit de déclarer dans le descripteur du module (client) le fait d'utiliser le service X en précisant son interface (clause *uses*)

- Exemple :

```
module my.helloer {  
    requires api.printer;  
    uses api.printer.IPrinter;  
}
```

Le module API  
où se trouve l'interface  
du service

L'interface du service


- A la compilation, aucune vérification n'est faite sur la présence obligatoire d'une implémentation du service (binding dynamique)  
Une résolution de module est tout de même effectuée en intégrant au graphe de modules tout module qui fournit une implémentation du service

## Consommer un service -suite-

- Le consommateur du service doit utiliser l'API ServiceLoader (qui existe depuis Java 6) pour accéder au service

- Exemple :

```
Iterable<IPrinter> printers = ServiceLoader.load(IPrinter.class);  
for (IPrinter printer : printers) {  
    printer.print(message);  
}
```



A chaque invocation de load  
un objet ServiceLoader  
est instancié (il est responsable  
de l'instanciation du service)

## Consommer un service -suite-

- Le consommateur du service doit utiliser l'API ServiceLoader (qui existe depuis Java 6) pour accéder au service

- Exemple :

```
Iterable<IPrinter> printers = ServiceLoader.load(IPrinter.class);  
for (IPrinter printer : printers) {  
    printer.print(message);  
}
```

Un objet ServiceLoader  
itérable (ServiceLoader<IPrinter>  
aussi)

A chaque invocation de load  
un objet ServiceLoader  
est instancié (il est responsable  
de l'instanciation du service)

## Consommer un service -suite-

- Le consommateur du service doit utiliser l'API ServiceLoader (qui existe depuis Java 6) pour accéder au service

- Exemple :

```
Iterable<IPrinter> printers = ServiceLoader.load(IPrinter.class);  
for (IPrinter printer : printers) {  
    printer.print(message);  
}
```

A ce moment, instantiation de(s)  
la classe(s) d'implémentation  
connue(s) du service

A chaque invocation de load  
un objet ServiceLoader  
est instancié (il est responsable  
de l'instanciation du service)

# Cycle de vie d'un service

- A quel moment la (les) classe(s) d'implémentation du service connue(s) est (sont) instanciée(s) ?  
Lorsqu'on utilise l'interface du service la première fois après l'invocation de `load(...)`
- Accéder une deuxième fois aux services (en utilisant le même `ServiceLoader`) donne accès aux mêmes objets (en cache : état des objets maintenu)
- C'est une fois après qu'on invoque une autre fois la méthode `load` qu'on obtient d'autres objets (nouvellement instanciés) ou bien :  
**`ServiceLoader`**<IPrinter> printers = `ServiceLoader.load(IPrinter.class)`;  
printers.**`reload()`**;
- Pas de singleton : chaque module aura des instances différentes

# Méthodes fournisseurs de services

- Dans la classe d'implémentation du service, il faut définir un constructeur public sans paramètres
- Parfois, on ne veut pas exposer un tel constructeur
- On met dans la classe d'implémentation une méthode publique et statique qui retourne un objet typé par l'interface du service
- Cette méthode est recherchée en premier par le ServiceLoader
- Aucun changement dans le descripteur du module
- Possibilité d'avoir un service (objet) singleton si dans cette méthode on implémente le patron « à la main »

# Sélectionner une implémentation de service

- Souvent, pour un même service on dispose de plusieurs implémentations disponibles. Comment choisir la meilleure ?
- Ce choix appartient au client du service, car cela dépend du domaine métier de l'application
- Souvent, on ajoute dans le service des méthodes qui retournent une description (caractéristiques) de l'implémentation du service :
  - le nom de l'algo implémenté, par exemple :  
"Naïve Bayes", "SVM", "Deep Neural Net", pour des algos d'apprentissage automatique qui implémentent tous de la classification de données par exemple (même interface)
  - des attributs de qualité de l'implémentation (complexité de l'algo)
- On peut aussi utiliser des annotations qu'on fournit dans l'API

# Dépendances optionnelles

- Parfois, on est amené à déclarer des dépendances optionnelles : Si le module dont on dépend existe, il sera utilisé, sinon, quelque chose d'autre va se passer (on a prévu un code pour le faire)
- Cela est possible grâce aux services (requérir une interface), qui peut ne pas être implémentée (et gérer ça avec ServiceLoader)
- Mais, parfois on ne veut pas imposer l'utilisation d'un ServiceLoader
- On peut donc utiliser une dépendance statique :  
`requires static un.module.optionnel;`  
Dans ce cas, la dépendance est résolue uniquement à la compil.  
A l'exécution, si le module n'existe pas aucune erreur n'est signalée



# Références bibliographiques

- Paul Bakker et Sander Mak  
**Java 9 Modularity**: Patterns and Practices for Developing Maintainable Applications. O'Reilly Media. Septembre 2017
- Alex Buckley.  
Tutoriel : **Modular Development with JDK 9**  
JavaOne (TM). Oracle. Octobre 2017  
<https://www.youtube.com/watch?v=gtcTftvj0d0>

# Questions

