Module, Packaging

Rémi Forax

Classe, Package et Module

Il y a 3 niveaux d'encapsulation en Java

- La classe

  contient des membres (champ, methode, classe interne)

  - 4 niveaux de visibilité (private, *default*, protected, public)

- Le package

  contient des classes

  - 2 niveaux de visibilité (*default*, public)

- Le module

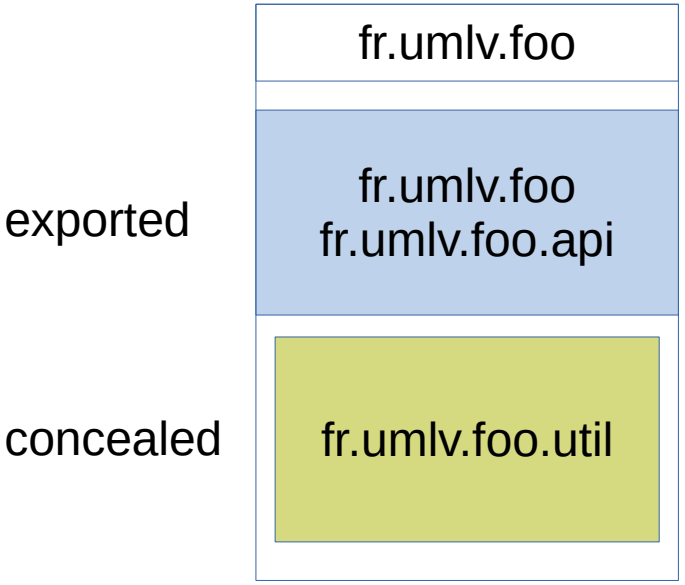
  contient des packages

  - 2 niveaux de visibilité (*default*, exports)

Encapsulation forte

Le fichier module-info.java décrit le contenu d'un module

Seuls les packages exportés sont définis



module-info.java

```
module fr.umlv.foo {
  exports fr.umlv.foo;
  exports fr.umlv.foo.api;
}
```

module-info sur le disque

Le module-info.java est dans le dossier qui contient les packages

monsuperprojet

src/main/java

module-info.java — module fr.umlv.foo

/fr

/umlv

/foo — package fr.umlv.foo

Main.java

/api — package fr.umlv.foo.api

Fizz.java

Buzz.java

/util — package fr.umlv.foo.util

Helper.java

module-info.java

module-info.java

```
module fr.umlv.foo {            // on peut mettre des annotations
  exports fr.umlv.foo;
  exports fr.umlv.foo.api;
}
```

Fizz.java

```
package fr.umlv.foo.api;

public class Fizz { … }
```

package-info.java

module-info.java

```
module fr.umlv.foo {            // on peut mettre des annotations
  exports fr.umlv.foo;
  exports fr.umlv.foo.api;
}
```

Fizz.java

```
package fr.umlv.foo.api;
public class Fizz { … }
```

package-info.java (dans fr.umlv.foo.api)

```
/** on met la doc du package ici !
  */
package fr.umlv.foo.api;    // on peut mettre des annotations
```

package-info sur le disque

monsuperprojet

src/main/java

module-info.java

/fr/umlv/foo

Main.java

/api

package-info.java descripteur du package

Fizz.java

Buzz.java

/util

package-info.java descripteur du package

Helper.java

Si on ne fait pas une vrai appli

On peut ne pas déclarer de module

- Les packages sont alors dans le module sans nom (*unamed module*)
- Mais tous les packages sont alors exportés

On peut ne pas déclarer de package

- Les classes sont alors dans le package par défaut (*default package*)
- Mais on ne peut pas utiliser *import*

Si on déclare un module, alors il faut obligatoirement mettre les classes dans un package

Packaging de module

Le format JAR permet de packager des .class

- Utilise le format ZIP
- Un seul module par jar

La commande `jar` permet aussi de spécifier des méta-données

- Version du jar
- La classe Main
- Version de l'OS pour les librairies native

Les méta-données sont stockés dans le module-info.class

ClassPath

Avant Java 9, l’ensemble des jars composant l’application est indiquer dans le CLASSPATH

```
java -classpath lib/fr.umlv.foo-1.0.jar:lib/bar.jar:etc ...
```

L’ordre est important car la VM cherche dans les jars de façon linéaire

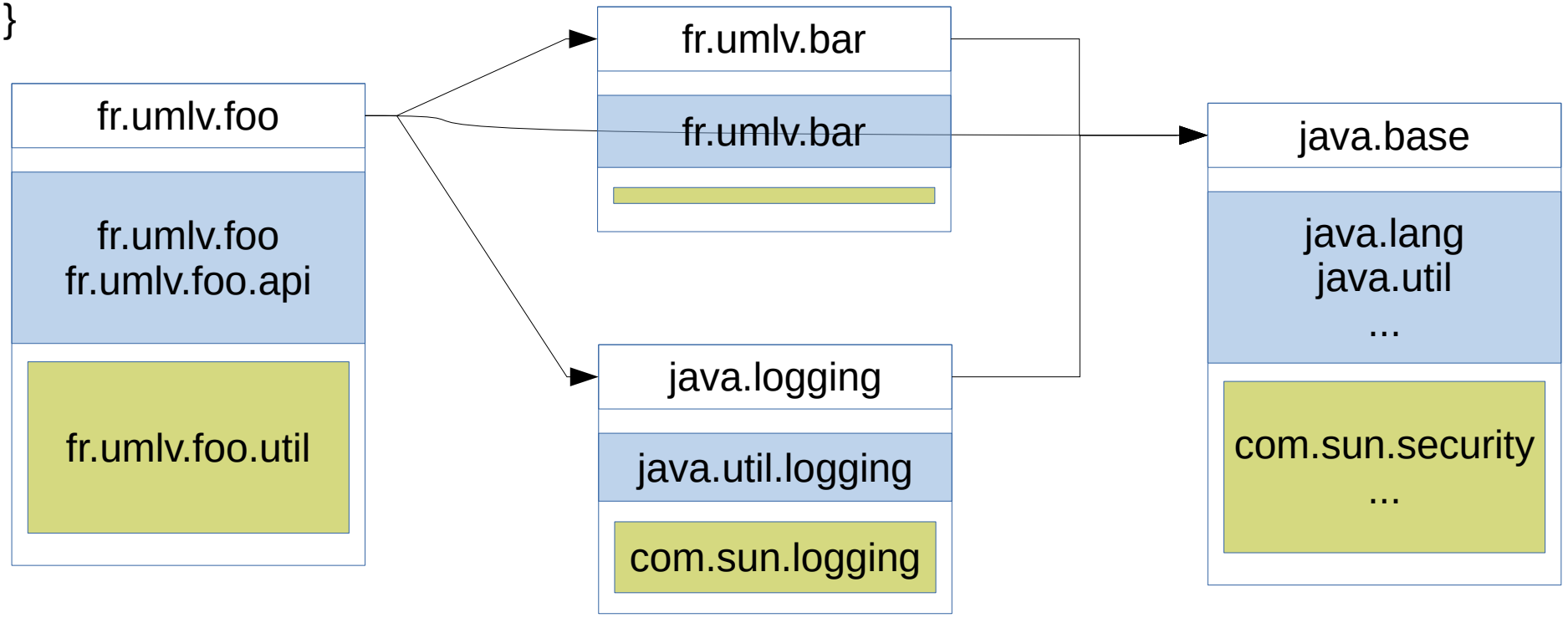
- Si il manque un jar
  - NoClassDefFoundError à l’exécution
- On peut facilement avoir des conflits si deux applis ont besoin de jar de version différentes
  - mais ce n’est pas detecté avant l’exécution

Specifier les dépendances

On utilise la directive requires

```
module fr.umlv.foo {
  exports …;

  requires java.base;      // pas nécessaire
  requires java.logging;
  requires fr.umlv.bar;
}
```



Specifier les dépendances

On utilise la directive requires

```
module fr.umlv.foo {
  exports …;

  requires java.logging;
  requires fr.umlv.bar;
}
```

A la compilation, pour utiliser une classe d'un package, il faut faire un requires du module qui exporte ce package

A l'exécution, la VM verifie récursivement que tous les modules sont présents avant de démarrer !

ModulePath

Le ModulePath contient des répertoires contenant les jar modulaires

Le compilateur et la VM vont chercher les modules en fonction des `requires`

- Pour utiliser un package, le package doit être dans un module `requires`

Compiler un projet

Compiler un module

```
javac --release 17
      -d output/modules/fr.umlv.foo/
      --module-path other/modules
       $(find src/main/java/fr.umlv.foo/ -name "*.java")
```

Compiler plusieurs modules

```
javac --release 17
      --module-source-path src
      -d output/modules/
      --module-path other/modules
       $(find src/main/java -name "*.java")
```

Exécuter un projet

Si le module déclare une classe Main

```
java --module-path mlibs
     --module fr.umlv.foo
```

La VM utilise les `requires` pour trouver l'ensemble des modules recursivement

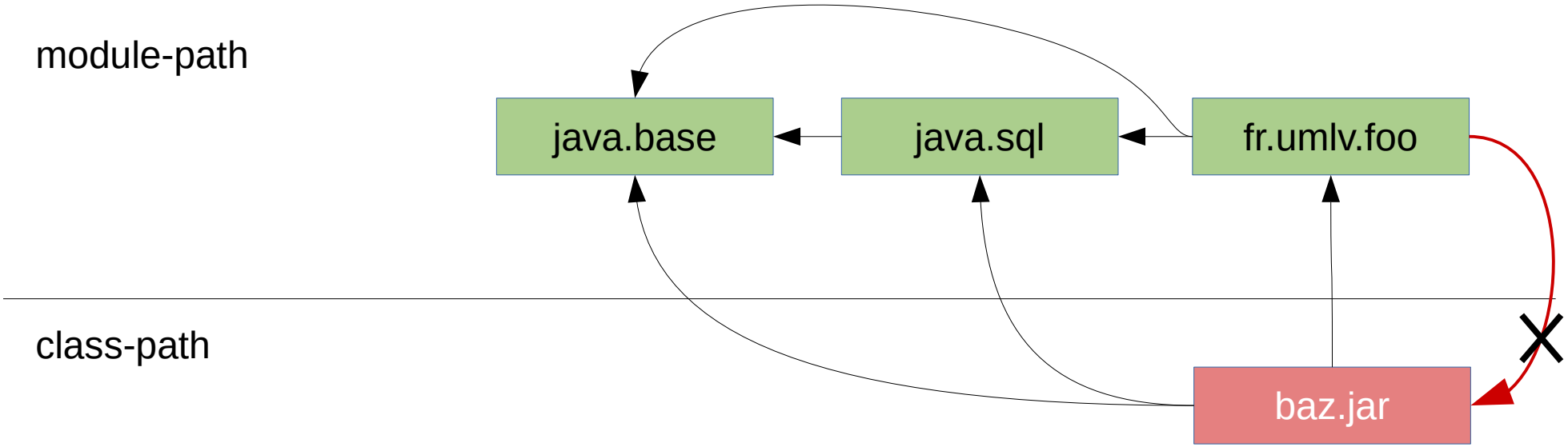
Si on veut spécifier une classe Main

```
java --module-path mlibs
     --module fr.umlv.foo/fr.umlv.foo.Main
```

ModulePath + ClassPath

Par compatibilité avec la java 8

- Un module ne peut référencer (requires) que des modules
- Le module non-nommé (qui contient tous les jars du class-path) référence tous les modules



Module Automatique

Et si un jar ne contient pas de module-info.class

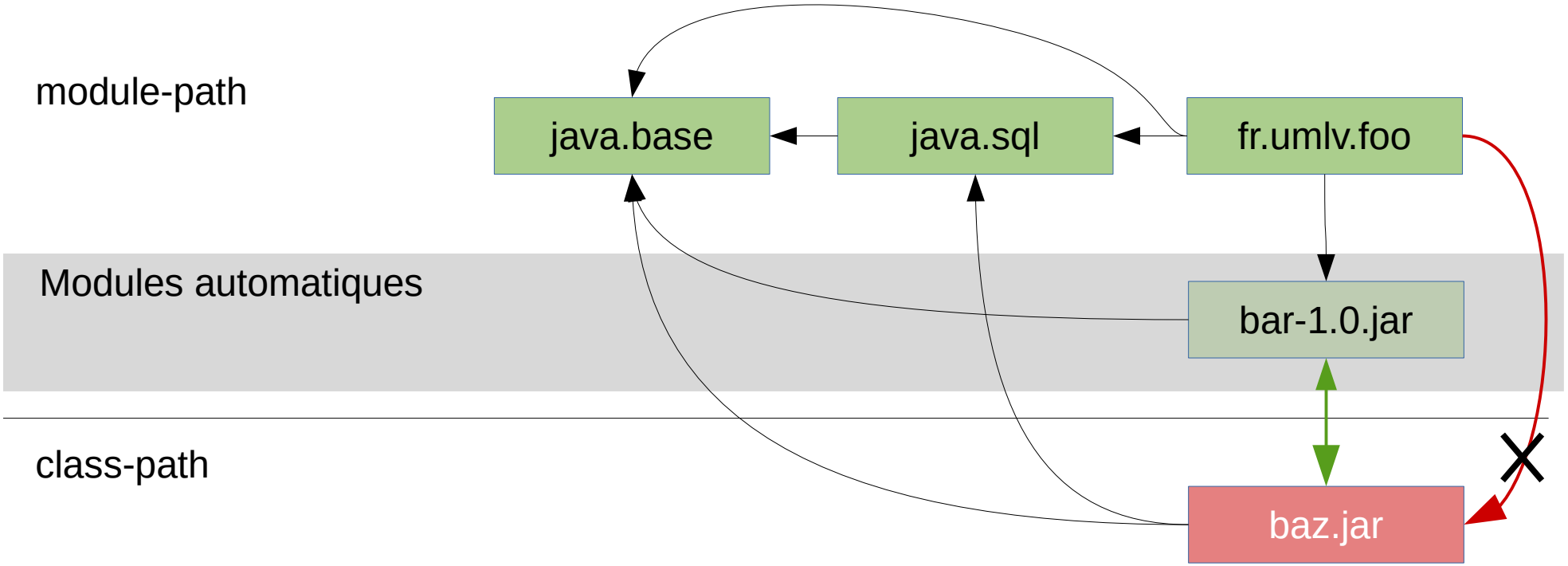
On peut le mettre dans le class-path mais il ne sera pas accessible pour les modules

On met le jar dans le module-path, dans ce cas, la VM crée un module automatique

- Le nom du module est extrait
  - du champ Automatic-Module-Name du fichier MANIFEST.MF
  - sinon du nom du jar *-version.jar
- Tous les packages contenus dans le jar sont exportés
- On peut utiliser requires sur un module automatique

Module automatique

Relation entre les jar modulaires, les jar automatiques et les jar classiques



Requires

On fait un requires pour 2 raisons

- Car l'implementation utilise des classes d'un module
- Car l'**API publique** utilise des classes d'un module

```
interface Fizz {
  Logger getLogger();
}
```

  - Fizz est déclarée dans fr.umlv.foo;
  - Logger est déclarée dans java.logging

pour le code qui veut utiliser Fizz, il faut faire un requires fr.umlv.foo et un requires java.logging **ou** ...

Requires Transitive

Si l'API publique d'un package exporté d'un module utilise une classe d'un autre module, on va utiliser un `requires transitive`

```
module fr.umlv.foo {
  exports fr.umlv.foo.api;  // contient Fizz

  requires transitive java.logging;
                            // car l'API de Fizz utilise Logger
  requires fr.umlv.bar;
                            // pour l'implantation de Fizz
}
```

Services

Injection de dépendance

Les modules possède déjà un mécanisme de *plugin* automatique

On peut demander

- la ou les implantations d'une interface (service) sans spécifier le nom de l'implantation
- Un module peut déclarer une ou plusieurs implantations du service

Avant l'exécution on vérifie qu'il existe au moins une implantation pour chaque service

Exemple - Drivers de base de donnée

Le module java.sql est déclaré comme ceci

```
module java.sql {
  requires ...
  exports …
  uses java.sql.Driver;
}
```

Le module com.mysql.jdbc créé par les developpeurs de MySQL

```
module com.mysql.jdbc {
    requires ...
    exports com.mysql.jdbc;
    provides java.sql.Driver with com.mysql.jdbc.Driver;
}
```

Déclaration de Service

`provides` interface `with` implementation

- Définie une implementation pour un service

`uses` interface

- Indique que l'implantation du code du module nécessite une/des implantations de l'interface

La classe java.util.ServiceLoader permet dans un code du module qui fait `uses` Foo de demander les implantation de Foo

Exemple - ServiceLoader

```
module fr.umlv.foo {
  requires java.sql;

  uses java.sql.Driver;
}
```

```
public class Fizz {
  public static void main(String[] args) {
    ServiceLoader<Driver> loader =
        ServiceLoader.load(Driver.class, Fizz.class.getClassLoader());
    Driver driver = loader.findFirst().get();
    ...
  }
}
```

Sécurité

Exports restreint

Il est possible d'exporter des packages uniquement à certain module

```
module fr.umlv.foo {
  requires …;
  export fr.umlv.foo.util to fr.umlv.bar;
}
```

En plus des classes du module fr.umlv.foo, les classes du package fr.umlv.util seront aussi visibles pour les classes du module fr.umlv.bar

- Permet de partager des packages d'implantations entre modules créer par les mêmes personnes

Exemple

Le package jdk.internal.misc contient entre autres la classe Unsafe

```
$ javap --module java.base module-info
module java.base {
  exports jdk.internal.misc to
      java.rmi,
      java.sql,
      jdk.charsets,
      …;
}
```

Le package jdk.internal.misc est visible que par certain modules du jdk

Deep reflection

Il est possible de faire de la “reflection profonde”

- Appeler des méthodes privées,
- Changer les champs finals
- Etc…

Doit être demandé explicitement dans le code en appelant setAccessible()

Open package

Si le code est dans un module, setAccessible ne marche pas

l'exception InaccessibleObjectException est levée

Sauf si on déclare le package open

```
module fr.umlv.foo {
  ...
  open fr.umlv.foo.api;  // deep reflection allowed
}
```

Open module

Au lieu de déclarer l'ensemble des packages "open" on peut déclarer le module "open"

* n'est pas alloué pour les directives des modules

```
open module fr.umlv.foo {
  ...
}
```

Cela revient à déclarer tous les packages du module (exportés ou non) open.

Backward Compatibility

Incompatible avec le jdk8

Le code accède à une classe du JDK qui est dans un package qui n'est pas exporté

- Par ex. les classe des package sun.* ou com.sun.* ou jdk.internal.*
- on peut utilise --add-export

  java --add-exports <module>/<package>=<target-module> …

  (ALL-UNNAMED si pas dans un module)

Le code utilise la deep-reflection

- on peut utilise --add-open

  java --add-open <module>/<package>=<target-module> …

Incompatible avec le jdk8

Il existe aussi une option globale qui ne marche que si un code d'une classe du class-path veut accéder à un package non-exporté

java --illegal-access=warn

Attention, cette option sera supprimée dans le future

jlink
(conteneur docker)

java pré-linker

Si une application est composée uniquement de modules, il est possible de générer une image unique pour le JDK + l'application

- L'image est spécifique à une plateforme (Linux x64, macOs, Windows x64, etc)

Permet de ne pas dépendre d'une version pré-installée du JDK

- Pratique pour IoT ou le Cloud
- Attention aux patchs de sécurité !

jlink

jlink crée une image particulière contenant

- les classes du jdk et de l'application mélangées
- une machine virtuelle
- un executable qui lance le tout

Il faut spécifier les modules de l'application et les modules du jdk

```
jlink --modulepath /usr/jdk/jdk-9/jmods:mlib
      --add-modules fr.umlv.foo
      --strip-debug
      --output image
```

Exemple (sur le disque)

$ ls -R image
image:
bin conf lib release

image/bin:
fr.umlv.foo java keytool

image/lib:
amd64 classlist jexec **modules** security tzdb.dat

…

image/lib/amd64/server:
classes.jsa libjsig.so libjvm.so Xusage.txt

Executer par l’executable
$ ./image/bin/fr.umlv.foo

Executer en utilisant le ‘wrapper java'
$ ./image/bin/java --module fr.umlv.foo

En résumé

Configuration fiable

- A terme, plus de ClassPath
- Fidèlité entre la compilation et l’exécution

Diminuer la taille de la plateforme

- Modularisation des applications et du JDK

Vrai Encapsulation

- Notion de visibilité de package
- Meilleure sécurité

Monde fermé

- Optimizations/Compilation statique ?