Les modules : Java $\geq$ 9

Plan

Pourquoi les modules en Java 9 (2017)

Les limites des JDK < 8

- Java 8 : 4240 entrées (Javadoc) -> **rt.jar 53 Mo**
- Difficultés liées au *classpath* : compil vs. exe, fichiers uniquement, pas de vérification au lancement
- `Classloader` : ne vérifie pas le *namespace*, ne gère pas les dépendances entre jars (l'ordre de chargement prime)
- $\Rightarrow$ beaucoup de problèmes apparaissent uniquement au *runtime*

Java 9 (JIGSAW) $\Rightarrow$ Modularisation

- Modularisation de la JVM $\Rightarrow$ déploiement / sécurité
- *classpath* et dépendances arborescentes
- vérification des dépendances au démarrage
- introduction d'un niveau de visibilité supérieur, entre modules

Définir un module : `module-info.java`

`v1.HelloModule.java`

```
package v1;

public class HelloModule {

  public static void main(String[] args) {
    System.err.println("hello world");
  }
}
```

`module-info.java` : à la racine des packages (`src`)

```
module fr.iut.modularization {
}
```

Remarques

Sur le nommage d'un module

- le nom `module-info.java` est standardisé et invariable
- c'est le seul nom de fichier `.java` qui peut contenir un '-'
- on nomme généralement le module du même nom que le package principal
- l'exemple précédent montre que (1) ce n'est pas obligatoire et (2) qu'il faut faire la différence entre le nom du module et le nom du package principal

Remarques

Sur l'exécution

- `java -cp bin v1.HelloModule` marche encore
- mais la bonne syntaxe est maintenant :

```
java --module-path bin
--module fr.iut.modularization/v1.HelloModule
```

Ajout d'une dépendance : module `requires`

`v2.HelloModule.java`

```
package v2;

import java.util.logging.Logger;

public class HelloModule {

  public static void main(String[] args) {
    Logger.getLogger(HelloModule.class.getName()).info("Hello world");
    System.err.println(System.getProperty("user.dir"));
  }
}
```

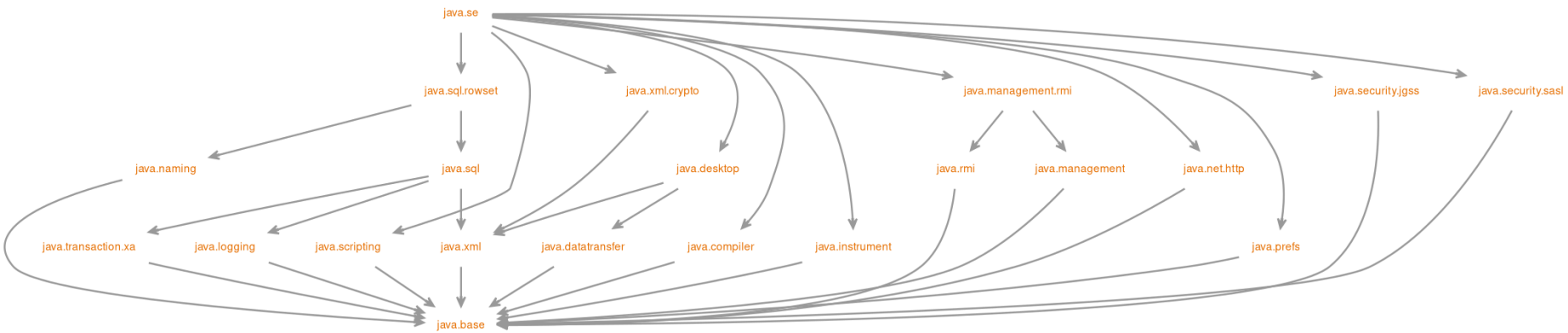
`module-info.java` : ajout d'un `requires` (dépendance)

```
module fr.iut.modularization {
  requires java.logging; //declaring the dependancy
}
```

- comprendre avec la Javadoc

image

- Module `java.se`



Ajout d'un module hors Java SE

v3.HelloModule.java

```
package v3;

import v2.HelloModule; // requires fr.iut.modularization

public class HelloDependancy {

  public static void main(String[] args) {
    HelloModule.main(null);
  }
}
}
```

module-info.java

```
module fr.iut.other {
  requires fr.iut.modularization;
}
```

keyword : `exports`

Ne marche pas immédiatement

- même avec la dépendance entre projets (v3 dépend de v2)
- *error : The type v2.HelloModule is not accessible*
- ? $\Rightarrow$ Il faut explicitement exporter les packages du module v2 !

`module-info.java` : modification de la V2

```
module fr.iut.modularization {

  //rend visible les classes de ce package (public/protected) pour ce module
  exports v2;
}
```

requires n'est pas transitif par défaut

v4.HelloDependancy.java

```
package v4;

import java.util.logging.Logger;

import v2.HelloModule; // requires fr.iut.modularization;

public class HelloDependancy {

  public static void main(String[] args) {
    HelloModule.main(null);
    Logger.getLogger(HelloDependancy.class.getName()).info("Hello world");
  }
}}
```

module-info.java

```
module fr.iut.other {
  requires fr.iut.modularization;

  //required even if defined in V2
  requires java.logging;
}
```

keyword : `transitive`

`module-info.java` : modification dans V2

```
module fr.iut.modularization {
  requires transitive java.logging;
  exports v2;
}
```

`module-info.java` : V4 OK maintenant

```
module fr.iut.other {
  requires fr.iut.modularization;
}
```

Mais ce n'est pas le bon cas d'utilsiation de `transitive` !

keyword : `transitive` ⇒ use case

v5.HelloModule.java

```
package v5;

import java.util.logging.Logger;

public class HelloModule {
  public static Logger getLogger() {//this return requires to read java.logging module
    return Logger.getLogger(HelloModule.class.getName());
  }
}
```

v6.HelloDependancy.java

```
package v6;

import java.util.logging.Logger;
import v5.HelloModule;

public class HelloDependancy {
  public static void main(String[] args) {
    Logger l = HelloModule.getLogger();//cannot be compiled without importing Logger
  }
}
```

keyword : `transitive` ⇒ use case

La V5 justifie l'utilisation de `transitive` :

`module-info.java` V5

```
module fr.iut.modularization {
  requires transitive java.logging;
  exports v5;
}}
```

keyword : `exports ...to`

`exports ...to` permet de spécifier quel(s) module(s) peut utiliser les packages exportés

`module-info.java` : V5 modifiée

```
module fr.iut.modularization {
  requires transitive java.logging;

  exports v5 to fr.iut.other;//only readable by this module
}
```

Services

Objectif et intérêt

- Définir un service (une interface Java) fournit par un module
- Définir son implémentation dans un autre module
- $\Rightarrow$ permet plusieurs implémentations d'un même service
- Permettre au module client du service d'utiliser différentes versions sans casser le code

Services ⇒ `provides ...with`

`module-info.java` : module définissant l'interface du service

```
module  com.example {

    //contains a service for using a codec :  com.example.CodecFactory
    exports  com.example
}
```

`module-info.java` : module où le service est implémenté

```
module iut.awesome {
  requires  com.example;

//the AwesomeCodecImplementation class implements the CodecFactory interface

  provides com.example.CodecFactory with
       iut.awesome.AwesomeCodecImplementation
}
```

Services ⇒ uses

module-info.java : module utilisateur du service

```
module com.moi.myservicelient {
    requires  com.example;

    uses  com.example.CodecFactory;
}
```

La déclaration ci-dessus ne définit pas où trouver l'implémentation. Ceci est résolu à l'exécution grâce à

- java.util.ServiceLoader

Dans une classe du module utilisateur par ex. :

```
ServiceLoader<CodecFactory> loader = ServiceLoader.load(CodecFactory.class);
for (CodecFactory factory : loader) {
    Encoder enc = factory.getEncoder("PNG");
    if (enc != null)
        ... use enc to encode a PNG file
        break;
    }
```

`open, opens, opens ...to`

À propos de réflexivité

- Avant Java 9, toutes les classes/attributs/méthodes sont accessibles par réflexivité (voir `java.lang.Class`)
- Java 9 permet une *strong encapsulation*
- $\Rightarrow$ rien n'est accessible sauf mentionné explicitement dans la déclaration du module

open, opens, opens ...to

open : réflexivité pour toutes les classes du module

```
open module fr.iut {//only at runtime
}
```

opens : ouvre uniquement le package spécifié

```
module fr.iut.modularization {//does not work with open

  opens iut.monpackage;
}
```

opens ...to : uniquement à certains modules

```
module fr.iut.other {

  opens iut.monpackage to fr.iut.modularization;
}
```

jlink : distribution autonome

Objectif et intérêt

- Les classes d'un programme Java, seules, nécessite d'être exécutées sur le bon JRE
- `jlink` permet de construire une distribution complète avec la JVM et les seuls modules nécessaires
- plus besoin d'avoir Java installé sur le système cible
- `jlink` est disponible dans le répertoire d'installation du JDK
- ▸ jlink

jlink : utilisation

Ligne de commande

```
jlink
--module-path bin:../modularizationV2/bin:/usr/lib/jvm/java-11-openjdk/jmods
--add-modules fr.iut.other
--output release
--launcher prog=fr.iut.other/modules.v4.HelloDependancy}
```

Fonctionnement

- `--module-path` : indique où se trouve les modules nécessaires à la construction des *root modules*
- `--add-modules` : indique les *root modules* à ajouter au runtime (et leurs dépendances par transitivité)
- `--output` : crée un répertoire `release` où tout sera stocké
- `--launcher` : crée un script `prog` win ou linux dans `release/bin` qui lancera la commande `java` appropriée

Conclusion

Sur les modules

- *Strong encapsulation* (visibilité module) : POO / sécurité
- déploiement : "petits exécutables" / `jlink`

écosystème Java

- Java 8 LTS jusqu'en 2030 ▸ Roadmap
- Java 11 LTS jusqu'en 2026