Java 9 - 12

Leuville - Objects

Versions

- Oracle sort une version bénéficiant du LTS (long-term support) tous les 3 ans, pour une durée de 3 ans.
  - Ainsi, le support commercial de Java 8 (qui date de mars 2014) se termine :
    - en janvier 2019 officiel
    - décembre 2020 pour le non commercial
    - septembre 2022 pour OpenJDK.

- Java 11 est la nouvelle version LTS jusqu'à septembre 2022 pour OpenJDK,
  - Oracle n'a pas encore annoncé de date pour les autres,

- il faut migrer !

J9 - *Méthodes privées dans les interfaces*

- Pour alléger les méthodes par défaut des interfaces depuis Java 8,

- Pour éviter la duplication de code, il possible d'implémenter des méthodes `private` dans une interface :

```
public interface Logger {

  default void info(String s) {
    log(message, "INFO");
  }

  default void debug(String s) {
    log(message, "WARN");
  }

  private void log(String s, String visibility) {
    // Do something
  }
}
```

J9 - *Try-with-resources*

- Arrivé au JDK 7, le `try-with-resources` permet d'instancier des objets implémentant `java.lang.AutoCloseable`
  - et donc tous les objets qui implémentent `java.io.Closeable` sans avoir à explicitement appeler la méthode `close()`.

- Grace à Java 9, on peut utiliser des variables `Closeable` instanciées en dehors d'un bloc `try-with-resources` :

```
FileReader reader
 = new FileReader("/lo/csv/file.csv");


try (reader) {
  // Do something
}
```

J9 - *Instanciation de collections immuables*

- Aide à l'instanciation de collections immuables :

```
List<String> list = List.of("Leuville", "love", "Java");

Set<Integer> set = Set.of(1, 2, 3, 4);

Map<Integer, String> map = Map.of(1, "Leuville", 2, "love", 3, "Java");

// Peut prendre jusqu'à 10 entrées
```

J9 - *L'annotation `@Deprecated` enrichie*

- 2 nouveaux attributs pour cette annotation :

```
@Deprecated(since="4.2"
forRemoval=true)
```

J9 – *API Flow*

- Java 9 intègre l'API Flow, vous permettant d'implémenter les propres flux réactifs :

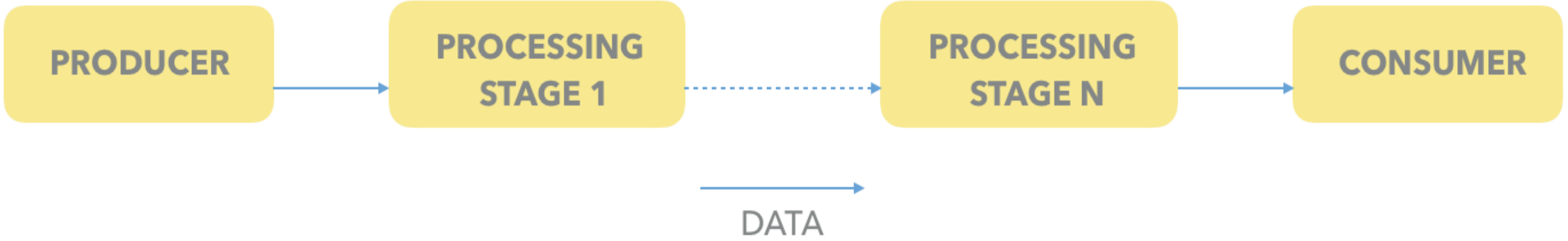
```
public class MySubscriber<T> implements Subscriber<T> {
  private Subscription subscription;

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
//a value of Long.MAX_VALUE may be considered as effectively unbounded
  }

  @Override
  public void onNext(T item) {
    System.out.println("Got : " + item);
    subscription.request(1);
//a value of Long.MAX_VALUE may be considered as effectively unbounded
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    System.out.println("Done");
  }
}
```

J9 – *API Flow*

- Reactive Streams est une norme pour le traitement asynchrone des données d'une manière de streaming avec un non-blocage.
  - À partir de Java 9, ils appartient auJDK sous la forme des interfaces `java.util.concurrent.Flow`.

- Dans une architecture de traitement de flux généralisée, on nomme les concepts principaux:
  - la source des données, parfois appelée le producteur,
  - la destination des données, parfois appelées le consommateur,
  - une ou plusieurs étapes de traitement qui font quelque chose avec les données.

- Dans un tel pipeline, les données circulent du producteur, à travers les étapes de traitement, au consommateur :

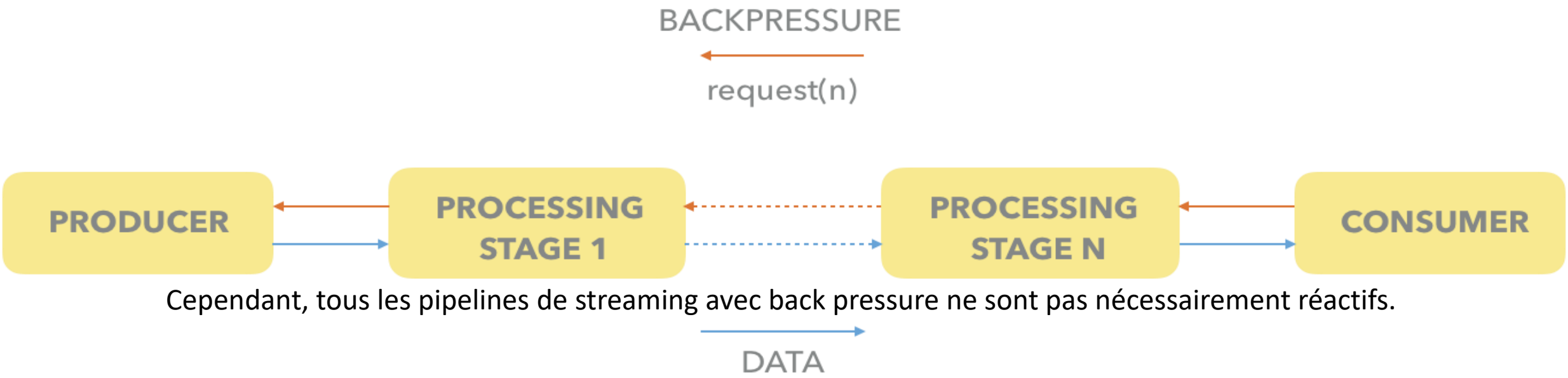


J9 – *API Flow*

- Maintenant, si on considère que les composants peuvent avoir des vitesses de traitement différentes, il y a 2 scénarios possibles:
  - Si l'aval (c'est-à-dire le composant qui reçoit des données) est plus rapide que l'amont (le composant qui envoie des données), c’est très bien car le pipeline devrait fonctionner en douceur.
  - Si, cependant, l'amont est plus rapide, alors l'aval devient inondé de données et les choses commencent à empirer. Il existe quelques stratégies pour traiter les données excédentaires :
    1. Bufferiser - mais les tampons ont une capacité limitée et vous allez manquer de mémoire tôt ou tard.
    2. Laissez tomber - mais alors on perd des données, mais peut avoir un sens dans certains cas - par exemple, c'est ce que le matériel de réseau fait souvent.
    3. Bloquez jusqu'à ce que le consommateur en ait fini, mais cela pourrait entraîner un ralentissement de l'ensemble du pipeline.

J9 – *API Flow*

- La meilleure façon de traiter ces différentes capacités de traitement est une technique appelée back pressure
  - Un consommateur plus lent demande une quantité fixe de données au producteur plus rapide, mais seulement un montant que le consommateur est en mesure de traiter à ce niveau.
- Pour le diagramme de pipeline de streaming, on considère la rétropression comme un type spécial de données de signalisation qui circulent dans la direction opposée (par rapport aux données régulières qui sont traitées :



Cependant, tous les pipelines de streaming avec back pressure ne sont pas nécessairement réactifs.

J9 – *API Flow*

- Le concept clé de Reactive Streams est le traitement des flux infinis de données d'une manière asynchrone et non-blocage, de sorte que les ressources informatiques (cœurs de processeur ou hôtes de réseau) peuvent être utilisés en parallèle.

- Il y a trois facteurs clés qui rendent un flux réactif :
  - les données sont traitées de manière asynchrone,
  - le mécanisme de back pressure est non-bloquant,
  - le fait que l'aval peut être plus lent que l'amont est en quelque sorte représenté dans le modèle de domaine.

- Des exemples : l'API en streaming Twitter, où on peut être déconnecté si la consommation trop lente, ou l'une des étapes intégrées dans Akka Streams, qui permet explicitement de planifier pour une période de temps en aval.

J9 – *API Flow*

- Les 4 interfaces sont:
  - un `Publisher<T>` est responsable de la publication d'éléments de type T et fournit une méthode d'abonnement aux abonnés pour s'y connecter,
  - un `Subscriber<T>` se connecte à un `publisher`, reçoit une confirmation via `onSubscribe`, puis reçoit des données via les rappels `onNext` et des signaux supplémentaires via `onError` et `onComplete`,
  - un `Subscription` représente un lien entre un `publisher` et un `subscriber`, et permet de faire reculer le `publisher` sur demande ou de mettre fin au lien avec annuler,
  - un processeur combine les capacités d'un `publisher` et d'un `subscriber` dans une seule interface.

J9 - Subscriber

- 4 méthodes dans cette interface:
  - `onSubscribe`: il s'agit de la 1ère méthode à être invoquée lorsque le subscriber est abonné pour recevoir des messages par le publisher. Habituellement, nous appelons subscription.request pour commencer à recevoir des éléments du processeur.
  - `onNext`: cette méthode est appelée lorsqu'un élément est reçu de le publisher. C'est ici que nous implémentons notre logique métier pour traiter le flux, puis que nous demandons plus de données à au publisher.
  - `onError`: Cette méthode est appelée lorsqu'une erreur irrécupérable se produit. Nous pouvons effectuer des tâches de nettoyage dans cette méthode, telles que la fermeture d'une connexion à une base de données.
  - `onComplete`: Cela ressemble à la méthode finally et est invoqué lorsqu'aucun autre élément n'est produit par l'éditeur et que l'éditeur est fermé. Nous pouvons l'utiliser pour envoyer une notification du traitement réussi du flux.

J9 - Une implémentation d'un publisher qui publie un itérateur arbitraire d'entiers

```
public class SimplePublisher implements
Flow.Publisher<Integer> {

  private final Iterator<Integer> iterator;

  SimplePublisher(int count) {
    this.iterator = IntStream.rangeClosed(1, count)
                             .iterator();
  }

  @Override
  public void subscribe(Flow.Subscriber<? super Integer>
subscriber) {
    iterator.forEachRemaining(subscriber::onNext);
    subscriber.onComplete();
  }
}
```

```
public static void main(String[] args) {
  new SimplePublisher(10).subscribe(new Flow.Subscriber<>() {
    @Override
    public void onSubscribe(Flow.Subscription subscription) {}

    @Override
    public void onNext(Integer item) {
      System.out.println("item = [" + item + "]");
    }

    @Override
    public void onError(Throwable throwable) {}

    @Override
    public void onComplete() {
      System.out.println("complete");
    }
  });
}
```

un abonné factice qui imprime juste les données reçues

```
item = [1]
item = [2]
item = [3]
item = [4]
item = [5]
item = [6]
item = [7]
item = [8]
item = [9]
item = [10]
complete
```

J9 - *JShell*

- Il s'agit du read-eval-print loop (REPL) de Java.
- Il permet ainsi de tester des instructions Java sans avoir à démarrer tout un programme.
- Il est possible de :
  - Importation de librairies
  - Déclaration de Classes
  - Déclaration d'Interfaces
  - Déclaration de méthodes
  - Déclaration de variables
  - Déclaration de constantes
  - Déclaration d'expressions

```
jshell> int i = 10;
i ==> 10

jshell> i
i ==> 10

jshell> i=20;
i ==> 20

jshell> int j = 30;
j ==> 30

jshell> /vars
|    int i = 20
|    int j = 30
```

J9 - *JShell*

```
jshell> int sum (int a, int b) {
   ...> return a+b;
   ...> }
|  created method sum(int,int)

jshell> /methods
|    int sum(int,int)

jshell> sum(2,2)
$6 ==> 4

jshell> /list sum

1 : int sum (int a, int b) {
   return a+b;
   }
```

- Pour éditer le code: `/edit sum`
- Pour changer d'éditeur:
  `/set editor "C:\\Program Files\\Sublime Text 3\\sublime_text.exe`
- Pour charger du code extérieur:
  `/open c:\\temp\\demo.java`

J9 - *Améliorations de l'API Stream*

- 4 nouvelles méthodes ont vu le jour dans l'API Stream :

```
Stream.of("a", "b", "c", "d").takeWhile(s ->
!s.equals("c")).forEach(System.out::println); // Affiche "ab"


Stream.of("a", "b", "c", "d").dropWhile(s ->
!s.equals("c")).forEach(System.out::println); // Affiche "cd"


Stream.ofNullable(null);
// Retourne un Stream vide, sans NullPointerException évidemment


IntStream.iterate(1, i -> i < 10, i -> i + 1);
// Équivalent à une boucle "for" de 0 à 9
```

J9 - *Jigsaw, le système modulaire*

- `Jigsaw` a pour objectif de rendre modulaire votre application.

- Le but est de réduire la taille des applications en n'embarquant que les dépendances nécessaires,
  - afin de déployer le logiciel sur des environnements où l'espace de stockage peut coûter cher, comme l'IoT ou le cloud.

- Pour en profiter, il faut ajouter un fichier nommé `module-info.java` à la racine du JAR :

```
module fr.lo.jva.infra {
    requires fr.lo.jva.domain;
    exports fr.lo.jva.infra.api;
}
```

J9 - *Jigsaw, le système modulaire*

- Un module est créé avec le fichier `module-info.java` et le mot-clé `module` est utilisé pour spécifier le nom du module.

- Le module de base est `java.base` qui est automatiquement résolu par tous les modules. Le module `java.base` contient les packages tels que `java.lang, java.io, java.net, java.util`, etc.

- Tous les packages exportés par le module `java.base` sont automatiquement disponibles pour tous les modules.

J9 - *Jigsaw, le système modulaire*

- Un module est un ensemble de packages conçus pour être réutilisés.
  - En Java 9, les programmes Java sont des modules.
  - Le module Java est la principale fonctionnalité introduite dans la version 9 de Java.

- En Java, il y a des classes, des packages et maintenant des modules.
  - Avant le module Java 9, les programmes Java sont composants OSGi.
  - Maintenant, en utilisant Module, on contrôle l'accessibilité entre les packages pour les méthodes publiques des classes.
  - Un package est l'ensemble des classes et un module est l'ensemble des packages.

- Les méthodes publiques de packages ne sont disponibles que pour les packages du module. On exporte les packages à partir de son module pour le rendre disponible aux packages d'autres modules.
  - Ainsi, seuls les packages exportés d'un module seront disponibles pour les packages d'autres modules.
  - On peut exporter des packages uniquement vers des modules amis et pas pour tous.

J9 - *Jigsaw*

- Soit un module nommé com.greetings qui imprime simplement «Greetings!».

- Le module se compose de 2 fichiers sources : la déclaration du module (`module-info.java`) et la classe principale.

- Par convention, le code source du module se trouve dans un répertoire qui est le nom du module.
  - `src/com.greetings/com/greetings/Main.java`
  - `src/com.greetings/module-info.java`

`$ cat src/com.greetings/module-info.java`

```
module com.greetings { }
```

`$ cat src/com.greetings/com/greetings/Main.java`

```
package com.greetings;
public class Main {
  public static void main(String[] args) {
    System.out.println("Greetings!");
  }
}
```

Le code source est compilé dans le répertoire `mods/com.greetings` à l'aide des commandes suivantes:

```
$ mkdir -p mods/com.greetings

$ javac -d mods/com.greetings \
      src/com.greetings/module-info.java \
      src/com.greetings/com/greetings/Main.java
```

J9 - *Jigsaw*

- Le code source est compilé dans le répertoire `mods/com.greetings` à l'aide des commandes suivantes:

```
$ mkdir -p mods/com.greetings

$ javac -d mods/com.greetings \
      src/com.greetings/module-info.java \
      src/com.greetings/com/greetings/Main.java
```

- Maintenant, pour lancer l'exemple avec la commande :

```
$ java --module-path mods -m
com.greetings/com.greetings.Main
```

- --module-path est le chemin du module, sa valeur est un ou plusieurs répertoires contenant des modules.
- -m spécifie le module principal, la valeur après le '/' est le nom de la classe de la classe principale du module.

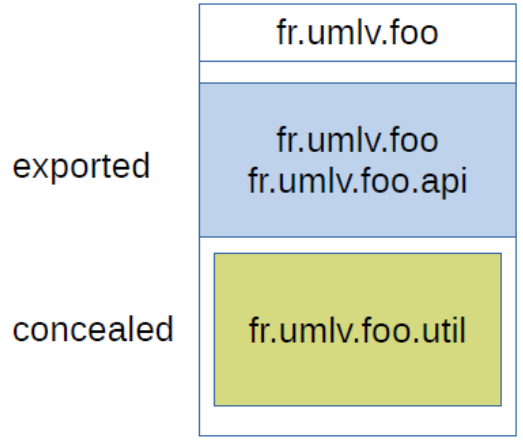
J9 - Modules

- 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)

J9 - Modules

- 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;
   }
```



J9 - module-info sur le disque

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

monsuperprojet

src/main/java

fr.umlv.foo — module fr.umlv.foo

module-info.java

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 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 { … }
```

J9 – package-info sur le disque

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
```

monsuperprojet

```
src/main/java
    fr.umlv.foo
        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
```

J9- Prototyper

- 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

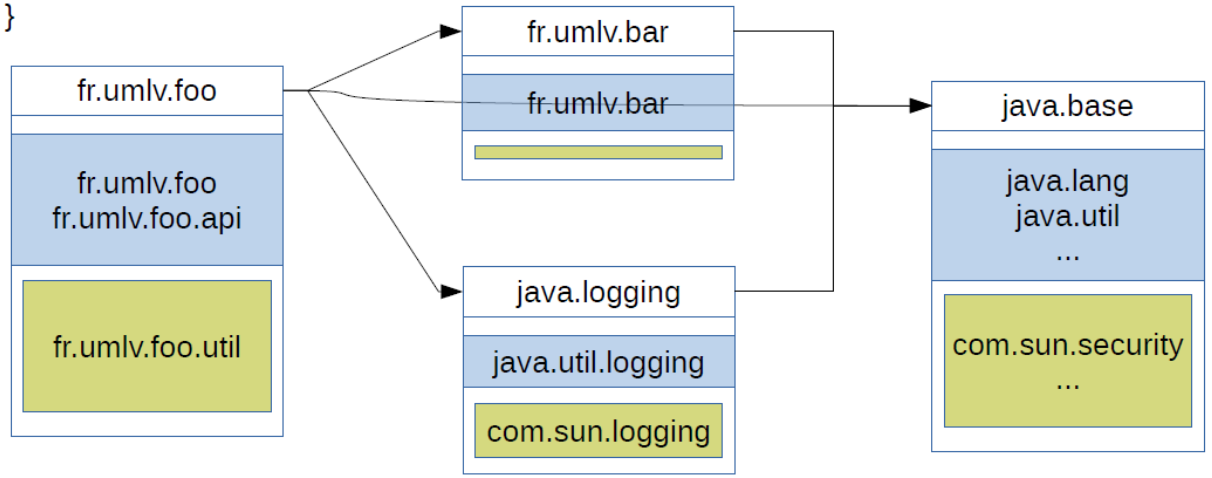
- On peut ne pas déclarer de package
  - Les classes sont alors dans le package par défaut (*default package*)
  - Mais on ne peut pas utiliser *import*

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

J9 – Spécification des dépendances

On utilise la directive `requires`

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

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



- 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 vérifie récursivement que tous les modules sont présents avant de démarrer !

J9 – Compilation d'un projet

- 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 module

```
javac --release 9
-d output/modules/fr.umlv.foo/
--module-path other/modules
$(find src/main/java/fr.umlv.foo/ -name "*.java")
```

- Compiler plusieurs modules

```
javac --release 9
--module-source-path src
-d output/modules/
--module-path other/modules
$(find src/main/java -name "*.java")
```

J9 – Exécution d'un projet

- Si le module déclare une classe Main

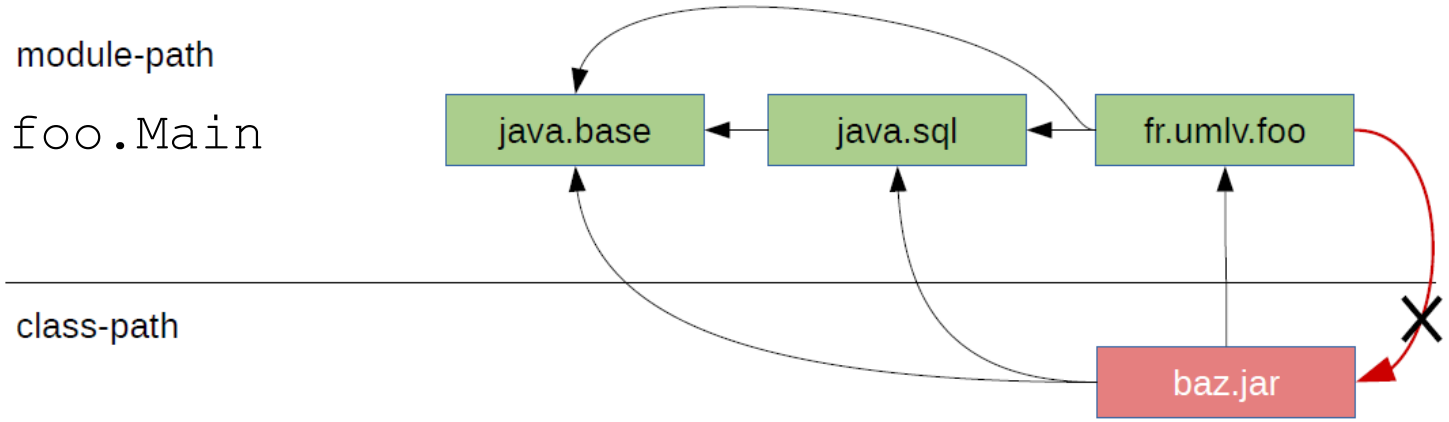
```
java --module-path mlibs
--module fr.umlv.foo
```

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

- Si on veut spécifier une classe Main

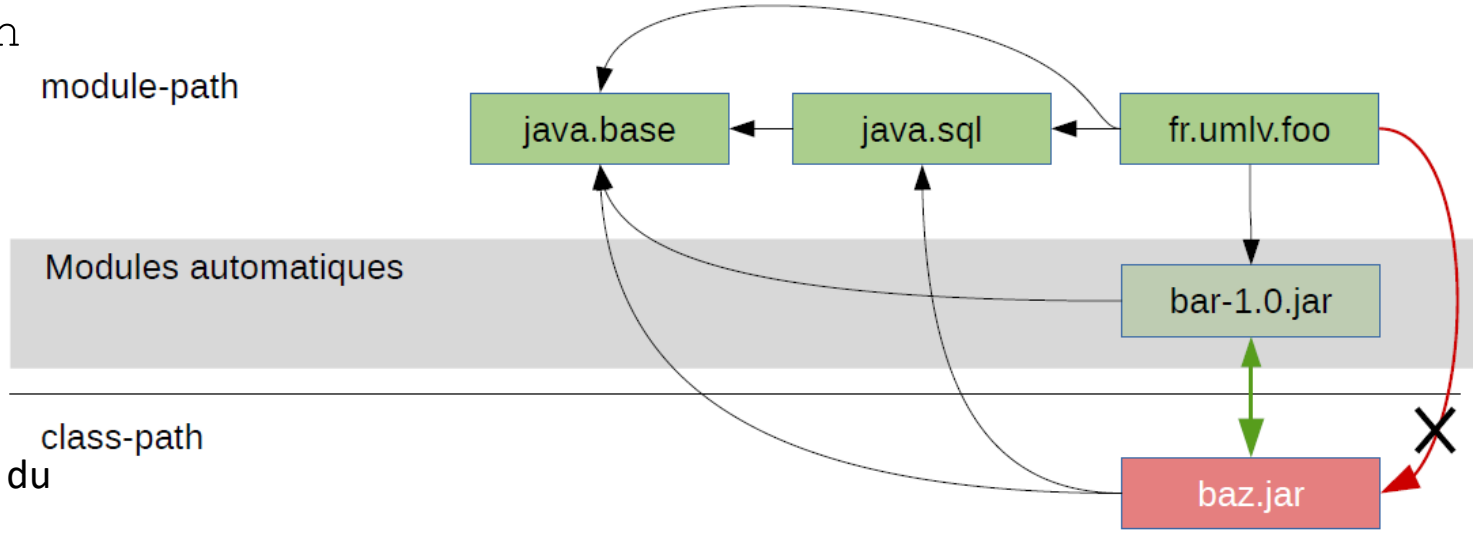
```
java --module-path mlibs
--module fr.umlv.foo/fr.umlv.foo.Main
```

- 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



J9 – 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
- Relation entre les jar modulaires, les jar automatiques et les jar classiques



J9 – dépendances transitives

- 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** ...

- 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
}
```

J9 - 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

- 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;
}
```

J9 - Déclaration de Service

- `provides` interface with implementation
  - Définit une implémentation 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`

```
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();
    ...
  }
}
```

J9 – export restreint

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

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

- En plus des classes du module `fr.lo.foo`, les classes du package `fr.lo.util` seront aussi visibles pour les classes du module `fr.lo.bar`
  - Permet de partager des packages d'implantations entre modules créer par les mêmes personnes

- 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

J9 - 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()`

- 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.lo.foo {
   ...
   open fr.lo.foo.api;
// deep reflection allowed
}
```

J9 – Open module

- Au lieu de déclarer l'ensemble des packages "open" on peut déclarer le module "open"
- * n'est pas autorisé pour les directives des modules

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

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

J9 – compatibilité J8

- 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> …
```

- 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

J9 - jlink

- 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` 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
```

J9 - Exemple

```
$ ls -R image
image:
bin conf lib release
image/bin:
fr.lo.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.lo.foo
```

Executer en utilisant le 'wrapper java'

```
$ ./image/bin/java --module fr.lo.foo
```

- Jaotc Génère une librairie partagée pour chaque module
  - Mode normal:
    - On génère le code pour tout module (le code n'est pas modifié à l'exécution)
  - Mode tiered:
    - On génére un code qui profile l'application et peut être remplacépar les JITs

```
jaotc --modulepath mlibs
--compile-for-tiered # AOT + JIT
--module fr.umlv.foo
--output foo.so
```

J10 - *Inférence de type*

- Différent des variables non typées, Java 10 intègre l'inférence de type afin de :
  - gagner en lisibilité
  - éviter ainsi la redondance de code

```
List<String> list =
List.of("Leuville", "love", "Java");

// Peut aussi s'écrire

var otherList = List.of("Leuville",
"love", "Java");
```

J10 - *Instanciation de collections immuables*

- La copie de `List` avec Collections#unmodifiableList :

```
var original = new ArrayList<>(List.of("Leuville", "love", "Java"));
var unmodifiable = Collections.unmodifiableList(original);
original.set(2, "Scala");
System.out.println(unmodifiable.get(2));

// Affiche Scala, et non Java car Collections#unmodifiableList retourne une
VUE non modifiable de la liste d'origine
```

- Avec Java 10, pour copier une `List` sans craindre de modifier la copie quand la `List` d'origine est modifiée, la méthode `copyOf` est apparue :

```
var original = new ArrayList<>(List.of("Leuville", "love", "Java"));
var copy = List.copyOf(original);
original.set(2, "Scala");
System.out.println(copy.get(2)); // Affiche Java
```

- De plus, de nouveaux `Collectors` ont été crée : `toUnmodifiableList`, `toUnmodifiableSet`, et `toUnmodifiableMap`.

J11 - *Inférence de type pour les paramètres de lambdas*

- Java 10 a apporté le mot clé `var`,

- mais on ne pouvait pas les utiliser dans les paramètres des expressions lambda en J10.
  C'est maintenant corrigé avec Java 11 :

- Certes, avec J8 il est inutile de spécifier le type. Mais il est utile si l'on souhaite rajouter une annotation (`@NonNull` par exemple) sur le paramètre.

```
var original = List.of("Leuville", "love", "Java");


original.stream().filter((var s) ->
s.contains("X")).forEach(System.out::println);
```

J11 - *Nouveau client HTTP*

- Initialement prévu avec J9, ce nouveau client HTTP est sorti de son incubateur avec Java 11,
- `HttpClient` est compatible avec la version 2 du protocole HTTP ainsi que les WebSocket.

- De plus, il permet d'exécuter des requêtes de manière non bloquante avec `BodyPublisher` et un `BodySubscriber`, tous 2 implémentant les interfaces de l'API `Flow` mise en place avec J9) :

```
HttpRequest request = HttpRequest.newBuilder().uri(new
URI("https://lo.fr/endpoint")).GET().build();


HttpResponse<String> response = HttpClient.newHttpClient().send(request,
HttpResponse.BodyHandler.asString());


httpClient.sendAsync(request, BodyHandlers.ofString()).thenAccept(response ->
System.out.println(response.body()));
```

J12 - Switch Expressions

- les switch expressions permettent de définir des switchs en tant qu'expression pour en récupérer le résultats dans une variable,
- Le switch ne va donc plus être juste une structure de contrôle (ensemble de if/else) mais permettre de calculer un résultats.

- Une syntaxe plus pratique d'utilisation et plus concise, qui utilise l'opérateur arrow déjà utilisé dans les lambda : '->'.
- On peut utiliser cette nouvelle syntaxe aussi bien dans un switch classique que dans un switch expression.

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

J12 - Switch Expressions

- Une switch expression qui permet de calculer un entier représentant le numéro de jour,
  - notez la présence du ‘;’ en fin d’expression qui indique bien que ce n’st plus une structure de contrôle mais bien une expression :

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

- Chaque `case` étant sur une ligne, on a un `break` implicite de la variable (ici un entier littéral).

```
int j = switch (day) {
    case MONDAY  -> 0;
    case TUESDAY -> 1;
    default      -> {
        int k = day.toString().length();
        int result = f(k);
        break result;
    }
};
```

- Il est possible d’écrire des `case` sur plusieurs lignes, on doit alors écrire un `break` à la fin du bloque de code avec la valeur que doit retourner l’expression,
- Ce nouveau switch est en mode preview, pour l’utiliser il faut ajouter l’option **`–enable-preview`** à la ligne de commande Java,

Migration

- Mise à jour:
  - De l'IDE aux dépendances externes, en passant par less plugins de build et même Maven ou Gradle, pour minimiser de problèmes lors du passage à Java 11.

- Rajoutez les dépendances manquantes
  - Si suite à la mise à jour d'une dépendance, `ClassNotFoundException` apparait, c'est qu'une classe n'est plus accessible suite à la modularisation d'une librairie tierce. Il faut rajouter une nouvelle dépendance ou le module à au projet.

- Pas besoin de tout rendre modulaire
  - Java 9 et Jigsaw n'ont pas supprimé le Classpath de Java. Il n'est pas nécessaire de rendre modulaire une application, à moins de développer une librairie pour laquelle on doit assurer une compatibilité pour les versions 8 à 11 de Java.
    Cependant, le faire va réduire grandement le taille du livrable et augmenter la vitesse de démarrage de l'application.