# JAVA 9

JÉRÉMY PERROUAULT

# INTRODUCTION

## INTRODUCTION

# INTRODUCTION

- Depuis Java9, la stratégie de nommage des versions a changé
  - Par exemple, Java 8 pouvait s'appeler « 8 » ou « 1.8 »
  - Ce n'est plus le cas, Java 9 s'appelle « 9 »

# INTRODUCTION

- A partir de Java9, une release tous les 6 mois
- Pour vérifier les nouveautés, comparer entre versions
  - https://javaalmanac.io/

# INTRODUCTION

- Java9 a apporté les changements suivants
  - Modularité (*jlink*, *jdeps*)
  - L'outil *jshell*
  - Améliorations **`CompletableFuture`**
  - Améliorations **`Process`** et pipelines sur **`ProcessBuilder`**
  - Améliorations **`Stream`** et **`Optional`**
  - **`Streams`** réactives
  - Nouveau client HTTP
  - Améliorations sur les Collections
  - Méthodes privées d'interface

# MODULES

## MODULARITÉ JAVA 9

# MODULES

- Java Platforme Module System (JPMS)
- La modularité introduite dans Java9 permet
  - D'avoir des classes non-accessibles en dehors du module
  - De modulariser la JRE (fabrication d'une nouvelle JRE avec *jlink*)
  - De vérifier la hiérarchie de dépendances avec *jdeps*
- Puisqu'en Java 8 et avant, la JVM recherche les classes utilisées dans
  - Les classes de la plateforme Java sont stockées dans un seul fichier rt.jar
  - Le classpath
  - Des répertories contenant des arborescences en packages contenant des fichiers « .class »
  - Des fichiers .jar
- C'est également le cas à la compilation

# MODULES

- Le fichier rt.jar est un énorme monolithe de plus de 53 Mo
- Le classpath peut être différent à la compilation et à l'exécution
- La JVM ne fait aucun contrôle, au lancement de l'application, des classes / jar présents
  - Les erreurs peuvent survenir pendant l'exécution

# MODULES

- Le projet JIGSAW à l'origine de la modularité Java
  - Un classpath sous forme d'arbre de dépendances
  - Un contrôle de la présence des modules nécessaires à l'application dès le démarrage
    - S'il manque un module, l'application ne démarre pas
  - Une sécurité renforcée : seuls les packages exportés explicitement sont visibles par les autres modules
  - La JVM elle-même est modulaire
    - java --list-modules pour avoir la liste des modules présents dans Java

# MODULES

- La JVM modulaire permet de n'inclure que les modules nécessaires
  - Fabrication d'une JRE spécifique pour chaque application
  - Réduit la taille de la JRE
  - Les classes internes à la JRE sont vraiment privées
  - Une sécurité renforcée car moins de classes chargées
    - Ne s'expose plus à une faille potentielle sur des classes non utilisées
  - Utilisation de l'outil *jlink* pour créer la JRE

# MODULES

- Un projet Java peut être un module
  - Avec le fichier module-info.java dans le répertoire source, dans lequel on précise
    - Le nom du module
    - Le(s) package(s) à exporter (accessibles depuis les autres modules – tous cachés par défaut)
      - Possible de ne rendre visible qu'à certains modules avec le mot-clé « to »
    - Le(s) module(s) requis
      - De façon « transitive » pour que ceux qui utiliseront notre module n'aient pas besoin d'importer le module
        - Permet donc de « masquer » l'arborescence de dépendances
      - De façon « static » pour rendre optionnelle la présence d'un module à l'exécution (obligatoire pour la compilation)
    - L'implémentation d'une ou plusieurs interfaces avec « provides » et « with »
      - Et précision de son utilisation dans le module avec « uses »

# MODULES

```
module fr.formation {
  requires fr.formation.model;

  uses fr.formation.ifaces.IDemo;
}
```

```
module fr.formation.model {
  exports fr.formation.model.brique1 to fr.formation;

  requires transitive fr.formation.ifaces;

  provides  fr.formation.ifaces.IDemo
    with    fr.formation.model.impl.DemoV1,
            fr.formation.model.impl.DemoV2;
}
```

```
module fr.formation.ifaces {
    exports fr.formation.ifaces;
}
```

# MODULES

```
public static void main(String[] args) {
  Demo.show();
  ServiceLoader<IDemo> demos = ServiceLoader.load(IDemo.class);

  demos.forEach(IDemo::demo);

  // HiddenClass hidden = new HiddenClass(); -> Pas accessible
}
```

# MODULES

- Avec la modularité, il n'est plus possible de faire ce qu'on veut grâce à la réflexivité
  - Utiliser `setAccessible` par exemple …
  - La sécurité de la JVM a été renforcée en ce sens
- Il faut donc « ouvrir » un package avec le mot-clé « opens » ou « open »

# MODULES

```
open module fr.formation.model {
    exports fr.formation.model.brique1 to fr.formation;
}
```

```
module fr.formation.model {
    exports fr.formation.model.brique1 to fr.formation;
    opens fr.formation.model.brique1;
}
```

```
Demo demo = new Demo();
Field field = Demo.class.getDeclaredField("test");
field.setAccessible(true);

System.out.println(field.get(demo));
```

# MODULES

- Java 9 peut mixer l'utilisation des modules et la non-utilisation
  - Dans ce cas, les librairies non modularisées sont placées dans un module « unamed »

# MODULES

- Pour voir les dépendances d'un jar
  - jdeps fichier.jar
  - jdeps --module-path "libs" -m nom.module
  - jdeps --module-path "libs" -recursive -summary fichier.jar
- Pour voir les dépendances d'un module
  - jdeps --module-path "libs" -m nom.module -recursive -summary

- Pour fabriquer une JRE spécifique
  - jlink --output dossier-jre --add-modules module1, module2, etc.
  - jlink --output dossier-jre --add-modules java.base,java.logging,java.naming
  - jlink --module-path lib --output dossier-jre --add-modules java.base,java.logging,java.naming

# JSHELL

## OUTIL JAVA

# JSHELL

- JShell est un outil de manipulation de Java, en dehors d'un projet Java
  - Un éditeur est proposé, avec historique
  - Il est possible d'exécuter un traitement pour tester, apprendre
  - L'instruction qui commence par « / » est une commande à JShell (/help, /history, etc.)
  - L'auto-complétion est possible avec Tabulation
  - L'import est possible avec Shift+Tab puis « i »

# JSHELL

- Il existe des plugins sous Eclipse, VSCode, etc. pour prendre en main JShell directement
- Ou bien utiliser la console classique

# JSHELL

- Les commandes
  - /save fichier.txt
  - /open fichier.txt

  - permettent de sauvegarder et ouvrir des fichiers contenant du code

# JCMD

## OUTIL JCMD

# JCMD

- jcmd est un outil qui permet d'obtenir des informations sur les processus qui utilisent la JVM actuellement sur la machine
  - jcmd
  - jcmd PID
  - jcmd PID help

# ASYNCHRONE

## NOUVEAUTÉS JAVA 9

# COMPLETABLEFUTURE

- De nouvelles fonctionnalités sont ajoutées
  - Nouvelles méthodes Factory
  - Support des délais et timeouts

```
CompletableFuture<String> completableFuture = helloAsync()
  .completeOnTimeout("Oups", 2000, TimeUnit.MILLISECONDS)
  .thenApply(s -> s + " World")
  .thenApply(s -> s + " !");

System.out.println(completableFuture.get());
```

# PROCESS

## NOUVEAUTÉS JAVA 9

# PROCESS – PROCESSHANDLE

- Nouvelle classe **`ProcessHandle`**
  - Permet de manipuler des processus système

```
ProcessHandle self = ProcessHandle.current();
ProcessHandle.Info procInfo = self.info();

Optional<Instant> startTime = procInfo.startInstant();
Optional<Duration> cpuUsage = procInfo.totalCpuDuration();

System.out.println(startTime);
System.out.println(cpuUsage);
```

# PROCESS – PROCESSHANDLE

- Récupérer les informations sur tous les processus

```
ProcessHandle.allProcesses().forEach(System.out::println);
```

# PROCESS – PROCESSBUILDER

- Possibilité d'ajouter des Pipelines dans l'exécution de processus (« | »)
  - En créant une liste de **`ProcessBuilder`**
  - En utilisant `startPipeline` de **`ProcessBuilder`**

# STREAMS

## NOUVEAUTÉS JAVA 9

# STREAMS

- De nouvelles fonctionnalités sont ajoutées
  - `takeWhile` Constitue un nouveau **`Stream`** avec les éléments, jusqu'à …
  - `dropWhile` Constitue un nouveau **`Stream`** avec les éléments depuis …
  - `ofNullable` Constitue un **`Stream`** vide si l'élément est vide

# STREAMS – TAKEWHILE

- Attend un **Predicate**

```
Stream<String> stream = Stream.of("A", "B", "", "D");

stream
  .takeWhile(s -> !s.isBlank())
  .forEach(System.out::println);
```

# STREAMS – DROPWHILE

- Fait l'inverse de `takeWhile`
- Attend un **`Predicate`**

```
Stream<String> stream = Stream.of("A", "B", "", "D");

stream
  .dropWhile(s -> !s.isBlank())
  .forEach(System.out::println);
```

# STREAMS – DROPWHILE

- Retourne à partir de l'élément qui répond à la condition
  - Contraitement à `takeWhile`, il garde cet élément

```
Stream<String> stream = Stream.of("A", "B", "", "D");

stream
  .dropWhile(s -> !s.isBlank())
  .skip(1)
  .takeWhile(s -> !s.isBlank())
  .forEach(System.out::println);
```

# STREAMS – OFNULLABLE

- Ressemble à `ofNullable` de **`Optional`**

```
System.out.println(Stream.ofNullable("42").count());
System.out.println(Stream.ofNullable(null).count());
```

# STREAMS – RÉACTIVE

- L'idée est de pouvoir parcourir un flux de données dans un état asynchrone
  - Flux éventuellement infini
  - Fait parti des nouvelles interfaces `java.util.concurrent.Flow`

- Un **`Flow.Subscriber<T>`** s'abonne à un **`Flow.Publisher<T>`**
  - Le publisher publie le flux
  - Le subscriber effectue les actions à la réception / à la fin du flux, ou lors d'une erreur par exemple

# STREAMS – RÉACTIVE

- Possibilité également d'utiliser un **SubmissionPublisher**
  - Qui implémente l'interface **`Flow.Publisher`**
  - Qui possède toute une mécanique d'appels, notamment avec la méthode `submit`

# OPTIONAL

## NOUVEAUTÉS JAVA 9

# OPTIONAL

- De nouvelles fonctionnalités sont ajoutées
  - `stream` Transforme un Optional en Stream
  - `or` Récupère un nouvel optional si l'élément n'existe pas
  - `isPresentOrElse` Permet d'accomplir les 2 actions : présent, et absent

# OPTIONAL – OR

- Attend un **`Supplier`**
  - Et il est possible de chainer, jusqu'à récupérer un élément

```
Optional<String> opt = Optional.empty();

System.out.println(
  opt
    .or(() -> Optional.empty())
    .or(() -> Optional.of("Défaut 2"))
);
```

# OPTIONAL – IFPRESENTORELSE

- Attend un **`Consumer`** et un **`Runnable`**
  - Combine les 2 actions à réaliser en fonction des 2 possibilités : présent et absent

```
Optional<String> opt = Optional.of("test");

opt.ifPresentOrElse(
  System.out::println, // Action si présent
  () -> System.out.println("Pas de valeur") // Action si absent
);
```

# CLIENT HTTP

## NOUVEAUTÉS JAVA 9

# CLIENT HTTP

- Supporte le protocole *HTTP/2* et *WebSocket*
  - Avec des performances similaires aux librairies *Apache*
- Sera disponible dans le package `java.net.http` à partir de Java 11

# CLIENT HTTP

- Constituer la requête HTTP
- Constituer les éventuels en-têtes, corps, etc.
- Fabriquer le client HTTP **`HttpClient`** avec ces informations, puis envoyer la requête

# COLLECTIONS

## NOUVEAUTÉS JAVA 9

# COLLECTIONS

- Possibilité de créer différents types de collections non-modifiables avec les méthodes `of`
  - Également disponible avec **`Stream`**, **`List`**, etc.

```
Set<String> fruits = Set.of("Orange", "Banane", "Citron");
fruits.forEach(System.out::println);
```

```
Map<String, Integer> map = Map.of("clé1", 42, "clé2", 104);
map.forEach((key, val) -> System.out.println(key + " => " + val));
```

# INTERFACES

## NOUVEAUTÉS JAVA 9

# INTERFACES

- Après les méthodes par défaut et les méthodes statiques
  - Qui devaient être publiques
- On peut maintenant faire la même chose en changeant leur portée en privée