Java - Nouveautés des versions 8 à 16

Christopher Loisel

Le système de modules (JDK 9 et plus)

- Architecture et modularité dans le JDK 9
- La déclaration des modules
- Exemples des classes d'applications modulaires
- Le fichier module-info.java
- Le graphe de dépendances
- Lancer une application à partir des modules
- Le packaging des modules et les JAR
- Exporter un package pour l'utiliser ailleurs
- Encapsulation forte et accessibilité

Architecture et modularité dans le JDK 9

Java 9 introduit un nouveau niveau d'abstraction au-dessus des packages, connu officiellement sous le nom de Java Platform Module System (JPMS), ou "Modules" en abrégé.

Dans ce cours, nous allons passer en revue le nouveau système et discuter de ses différents aspects.
Nous allons également construire un projet simple pour démontrer tous les concepts que nous allons apprendre dans ce guide.

Classe, Package et Module

Il y a 3 niveaux d'encapsulation en Java

1. – La classe contient des membres (champ, méthode, classe interne)
    - 4 niveaux de visibilité (private, default, protected, public)
2. – Le package contient des classes
    - 2 niveaux de visibilité (default, public)
3. – Le module contient des packages
    - 2 niveaux de visibilité (default, exports)

Qu'est-ce qu'un module ?

Tout d'abord, nous devons comprendre ce qu'est un module avant de pouvoir comprendre comment les utiliser.

Un module est un groupe de paquets et de ressources étroitement liés, accompagné d'un nouveau fichier descripteur de module.

En d'autres termes, il s'agit d'une abstraction de type "paquet de paquets Java" qui nous permet de rendre notre code encore plus réutilisable.

Packages

Les packages d'un module sont identiques aux packages Java que nous utilisons depuis la création de Java.

Lorsque nous créons un module, nous organisons le code en interne dans des packages, comme nous l'avons fait précédemment pour tout autre projet.

En plus d'organiser notre code, les packages sont utilisés pour déterminer quel code est accessible publiquement en dehors du module.

Resources

Chaque module est responsable de ses ressources, comme les médias ou les fichiers de configuration.

Auparavant, nous placions toutes les ressources au niveau de la racine de notre projet et gérions manuellement celles qui appartenaient aux différentes parties de l'application.

Avec les modules, nous pouvons envoyer les images et les fichiers XML nécessaires avec le module qui en a besoin, ce qui rend nos projets beaucoup plus faciles à gérer.

Descripteur de module

Lorsque nous créons un module, nous incluons un fichier descripteur qui définit plusieurs aspects de notre nouveau module :

1. Nom - le nom de notre module
2. Dépendances - une liste des autres modules dont ce module dépend.
3. Paquets publics - une liste de tous les paquets que nous voulons rendre accessibles depuis l'extérieur du module.
4. Services offerts - nous pouvons fournir des implémentations de services qui peuvent être consommés par d'autres modules.
5. Services consommés : permet au module actuel d'être consommateur d'un service.
6. Reflection Permissions - permet explicitement aux autres classes d'utiliser la réflexion pour accéder aux membres privés d'un paquet.

Les règles de dénomination des modules sont similaires à la façon dont nous nommons les paquets (les points sont autorisés, les tirets ne le sont pas). Il est très courant de faire des noms de style projet (my.module) ou de style Reverse-DNS (com.christopher.mymodule). Nous utiliserons le style projet dans ce guide.

Nous devons lister tous les paquets que nous voulons rendre publics car par défaut, tous les paquets sont privés de module.

Il en va de même pour la réflexion. Par défaut, nous ne pouvons pas utiliser la réflexion sur les classes que nous importons d'un autre module.

Plus loin dans l'article, nous verrons des exemples d'utilisation du fichier descriptif de module.

Types de modules

Il existe quatre types de modules dans le nouveau système de modules :

- Modules système - Ce sont les modules listés lorsque nous exécutons la commande *list-modules* ci-dessous. Ils comprennent les modules Java SE et JDK.
- Modules d'application - Ces modules sont ceux que nous voulons généralement construire lorsque nous décidons d'utiliser les modules. Ils sont nommés et définis dans le fichier compilé module-info.class inclus dans le JAR assemblé.
- Modules automatiques - Nous pouvons inclure des modules non officiels en ajoutant des fichiers JAR existants au chemin du module. Le nom du module sera dérivé du nom du JAR. Les modules automatiques auront un accès complet en lecture à tous les autres modules chargés par le chemin.
- Module sans nom - Lorsqu'une classe ou un JAR est chargé dans le classpath, mais pas dans le chemin du module, il est automatiquement ajouté au module sans nom. Il s'agit d'un module fourre-tout qui permet de maintenir une compatibilité rétroactive avec le code Java précédemment écrit.

Distribution

Les modules peuvent être distribués de deux manières : sous forme de fichier JAR ou de projet compilé "éclaté". Il s'agit bien entendu de la même chose que pour tout autre projet Java, ce qui n'est donc pas surprenant.

Nous pouvons créer des projets multi-modules composés d'une “application principale” et de plusieurs modules de bibliothèque.

Nous devons cependant faire attention car nous ne pouvons avoir qu'un seul module par fichier JAR.

Lorsque nous configurons notre fichier de construction, nous devons nous assurer de regrouper chaque module de notre projet dans un JAR distinct.

Default Modules

Lorsque Java 9 est installé, vous constaterez que le JDK a une nouvelle structure. Ils ont pris tous les packages d'avant et les ont divisés en modules. Les modules sont faciles à trouver en tapant 'java -list module' ou en les parcourant avec 'java -list-all-modules'.

Ce cours est divisé en quatre modules : java, javafx, jdk et Oracle.

Les modules Java sont les implémentations de classe de la spécification du langage SE de base

Les modules javafx sont les bibliothèques de l'interface utilisateur FX.

Tout ce qui est nécessaire au JDK lui-même est conservé dans les modules jdk.

Et enfin, tout ce qui est spécifique à Oracle est dans les modules oracle.

Déclarations de modules

Pour configurer un module, nous devons placer un fichier spécial à la racine de nos paquets, nommé module-info.java.

Ce fichier est connu sous le nom de descripteur de module et contient toutes les données nécessaires pour construire et utiliser notre nouveau module.

Nous construisons le module avec une déclaration. Le corps de cette déclaration peut être vide ou contenir des directives de module.

```
module monModuleNom {
    // toutes les directives sont facultatives
}
```

Pour démarrer un module, nous devons d'abord utiliser le mot-clé module, puis intituler le module.

Le module fonctionnera avec cette déclaration, mais nous aurons généralement besoin de plus d'informations.

C'est là qu'interviennent les directives de module.Le module fonctionnera avec cette déclaration, mais il y a généralement plus de détails qui doivent être spécifiés.

La déclaration des modules

- Requires
- Exports
- Uses
- Provides
- With
- Opens

Requires

Notre première directive est requise. Cette directive nous permet de déclarer les dépendances des modules :

```
module my.module {

    requiert le nom du module ;

}
```

Maintenant, my.module a une dépendance à la fois à l'exécution et à la compilation de module.name.

Et tous les types publics exportés par une dépendance sont accessibles par notre module lorsque nous utilisons cette directive.

Requires Static

Parfois, nous écrivons du code qui fait référence à un autre module, mais que les utilisateurs de notre bibliothèque ne voudront jamais utiliser.

Par exemple, nous pouvons écrire une fonction utilitaire qui imprime notre état interne lorsqu'un autre module de journalisation est présent. Mais, tous les utilisateurs de notre bibliothèque ne voudront pas de cette fonctionnalité, et ils ne veulent pas inclure une bibliothèque de journalisation supplémentaire.

Dans ces cas, nous voulons utiliser une dépendance optionnelle. En utilisant la directive statique requires, nous créons une dépendance à la compilation uniquement :

```
module my.module {

    requires static module.name ;

}
```

Requires Transitive

Nous travaillons couramment avec des bibliothèques pour nous faciliter la vie.

Mais nous devons nous assurer que tout module qui intègre notre code intègre également ces dépendances "transitives" supplémentaires, sinon il ne fonctionnera pas.

Heureusement, nous pouvons utiliser la directive transitive requires pour forcer les consommateurs en aval à lire également nos dépendances requises :

```
module my.module {

    requires transitive module.name ;

}
```

Désormais, lorsqu'un développeur exige mon.module, il n'aura pas besoin de dire aussi exige module.name pour que notre module fonctionne toujours.

Exports

Par défaut, un module n'expose aucune de ses API aux autres modules. Cette forte encapsulation a été l'une des principales motivations pour créer le système de modules en premier lieu.

Notre code est nettement plus sûr, mais nous devons maintenant ouvrir explicitement notre API au monde entier si nous voulons qu'elle soit utilisable.

Nous utilisons la directive exports pour exposer tous les membres publics du paquet nommé :

```
module my.module {

    exports com.my.package.name ;

}
```

Désormais, lorsque quelqu'un demande mon.module, il aura accès aux types publics de notre paquet com.my.package.name, mais à aucun autre paquet.

Exports … To

Nous pouvons utiliser les exportations... pour ouvrir nos classes publiques au monde entier.
Mais que faire si nous ne voulons pas que le monde entier ait accès à notre API ?
Nous pouvons restreindre les modules qui ont accès à nos API à l'aide de la directive exports...to.
Comme pour la directive exports, nous déclarons qu'un paquet est exporté.
Mais nous listons également les modules que nous autorisons à importer ce paquet en tant que requis.
Voyons à quoi cela ressemble :

```
module my.module {
    export com.my.package.name to com.specific.package ;
}
```

Uses

Un service est une mise en œuvre d'une interface spécifique ou d'une classe abstraite qui peut être consommée par d'autres classes.

Nous désignons les services que notre module consomme avec la directive uses.

Notez que le nom de la classe que nous utilisons est l'interface ou la classe abstraite du service, et non la classe d'implémentation :

```
module my.module {

    uses class.name ;

}
```

Nous devons noter ici qu'il existe une différence entre une directive requires et la directive uses.

Nous pouvons exiger un module qui fournit un service que nous voulons consommer, mais ce service implémente une interface de l'une de ses dépendances transitives.

Au lieu de forcer notre module à requérir toutes les dépendances transitives, juste au cas où, nous utilisons la directive uses pour ajouter l'interface requise au chemin du module.

Provides … With

Un module peut également être un fournisseur de services que d'autres modules peuvent consommer.

La première partie de la directive est le mot clé provides. C'est ici que nous mettons le nom de l'interface ou de la classe abstraite.

Ensuite, nous avons la directive with où nous fournissons le nom de la classe d'implémentation qui implémente l'interface ou étend la classe abstraite.

Voici à quoi cela ressemble une fois assemblé :

```
module my.module {

    provides MyInterface with MyInterfaceImpl ;

}
```

Open

Nous avons mentionné plus haut que l'encapsulation était un élément moteur de la conception de ce système de modules.

Avant Java 9, il était possible d'utiliser la réflexion pour examiner chaque type et membre d'un paquetage, même ceux qui étaient privés. Rien n'était vraiment encapsulé, ce qui pouvait poser toutes sortes de problèmes aux développeurs de bibliothèques.

Comme Java 9 impose une encapsulation forte, nous devons désormais autoriser explicitement les autres modules à réfléchir sur nos classes.

Si nous voulons continuer à autoriser la réflexion complète comme le faisaient les anciennes versions de Java, nous pouvons simplement ouvrir le module entier :

```
open module my.module {

}
```

Opens

Si nous devons autoriser la réflexion de types privés, mais que nous ne voulons pas que tout notre code soit exposé, nous pouvons utiliser la directive opens pour exposer des paquets spécifiques.

Mais n'oubliez pas que cela ouvrira le paquet au monde entier, alors assurez-vous que c'est bien ce que vous voulez :

```
module my.module {

  ouvre com.mon.paquet ;

}
```

Opens … To

D'accord, la réflexion est parfois géniale, mais nous voulons toujours autant de sécurité que ce que nous pouvons obtenir de l'encapsulation. Nous pouvons ouvrir sélectivement nos paquets à une liste pré-approuvée de modules, dans ce cas, en utilisant la directive opens...to :

```
module my.module {

    ouvre com.my.package à moduleOne, moduleTwo, etc.. ;

}
```

Le fichier module-info.java

Les métadonnées du module sont fournies dans un descripteur de module. Les métadonnées de module sont des données qui définissent les modules eux-mêmes, y compris ce qu'ils peuvent faire, d'où ils viennent et quelles dépendances ils ont. Ces méta-données doivent répondre à plusieurs questions :

- quel est le nom du module ?
- quelles sont les dépendances requises ? Il existe une dépendance implicite à java.base par défaut.
- Aucun package n'est exporté par défaut. Lorsqu'une classe publique est accessible en dehors du module, vous devez soit exporter le package auquel elle appartient, soit utiliser une instruction d'importation complète comme from future_module.other_package .

Le fichier descripteur de module contient toutes les informations concernant la description d'un module ; en particulier

- le nom du module
- les modules dont dépend ce module : les modules requis par ce module
- Les modules qu'il expose permettent à leurs classes publiques d'être utilisées par d'autres.
- paquets avec lesquels l'introspection des classes dans d'autres modules est autorisée
- les services qu'il expose et/ou consomme

Cette description peut être complétée par des détails supplémentaires sur notre service de rédaction. Par exemple, nous fournissons

Chaque module a un descripteur de module. Il s'agit d'un fichier source Java qui doit passer par la compilation pour générer le fichier .class, c'est-à-dire Module-info.

Un descripteur de module utilise une syntaxe spécifique utilisant des mots clés contextuels de la forme :

[**open**] **module** <nom-module>

{

[**export** <nom-package> [**to** <nom-module>]]*

[**requires** [**transitive**] <nom-module>] *

[**opens** <nom-packa> [**to** <nom-module]]*

[**provides** <type-service> **with** <nom-classe>]*

[**uses** <type-service>]*

}

Les limitations du format jar

les inconvénients des emballages JAR

- Le nom du fichier JAR n'est pas utilisé par la JVM
- Un JAR ne permet pas de déclarer des dépendances
- Il n'y a pas de mécanisme de versioning proposé

Ces inconvénients entraînent divers problèmes, tels que :

- Il est impossible pour la JVM d'identifier tous les JAR requis dans un classpath. Donc si une classe n'est pas trouvée, une exception de type NoClassDefFoundError sera levée à sa première utilisation
- Un fichier jar est le type de conteneur le plus simple. Il n'est cependant pas possible d'utiliser l'encapsulation entre les fichiers jar.
- Conflit de version si plusieurs versions d'un jar sont présentes dans le classpath.

Pas d'encapsulation dans un jar ou entre les jars

Il n'est pas possible d'avoir des éléments visibles, par exemple uniquement par ceux du fichier JAR qui les contiennent, ou en dehors de celui-ci. Le mécanisme de contrôle d'accès du langage de programmation Java et de la machine virtuelle Java ne permet à aucun composant d'empêcher d'autres composants d'accéder à ses packages.

Historiquement, l'encapsulation est uniquement proposée au travers des modificateurs de visibilité. Les modificateurs de visibilité de Java peuvent être utilisés pour implémenter l'encapsulation entre les classes d'un même package. Mais entre plusieurs packages, il n'y a que la visibilité publique qui soit utilisable.

Toutes les classes publiques sont accessibles par toutes les autres du classpath. Cela limite les possibilités d'encapsulation notamment au niveau du JAR.

En conséquence de nombreuses classes de l'API Core de Java sont public mais ne devrait pas être utilisée. Mais comme elles sont publiques, il est possible de les utiliser directement ou indirectement si c'est une dépendance qui les utilise. C'est notamment le cas des classes des packages sun.misc, jdk.internal, ... La plus connues d'entre-elles est la classe sun.misc.Unsafe. Malgré son nom explicite, elle est largement utilisée notamment par des frameworks couramment utilisés.

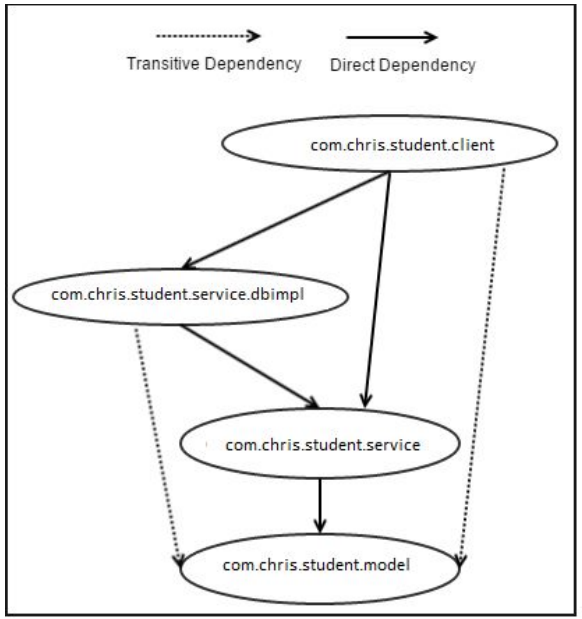
Les API internes du JDK, principalement dans les packages sun.* sont encapsulées dans des modules qui ne les exportent pas : ces API ne sont donc plus utilisables.

La vocation de ces API n'était pas d'être utilisable en dehors du JDK mais comme elles étaient public et proposaient des fonctionnalités puissantes, elles ont été largement utilisées par des bibliothèques.

Certaines de ces API ont été remplacées et certaines sont encore accessibles pour le moment mais elles devront être retirées à terme.

Exemples des classes d'applications modularisées

Créons une application modulaire simple avec des modules et leurs dépendances comme indiqué dans le graphique ci-dessous :



Le module com.chris.student.model est le module racine. Il définit la classe de modèle com.chris.student.model.Student, qui contient les propriétés suivantes :

```
public class Student {
    private String registrationId;
    //other relevant fields, getters and setters
}
```

Il fournit aux autres modules les types définis dans le paquetage com.chris.student.model. Pour ce faire, il est défini dans le fichier module-info.java :

```
module com.chris.student.model {
    exports com.chris.student.model;
}
```

Le module com.chris.student.service fournit une interface com.chris.student.service.StudentService avec des opérations abstraites :

```
public interface StudentService {
    public String create(Student student);
    public Student read(String registrationId);
    public Student update(Student student);
    public String delete(String registrationId);
}
```

Il dépend du module com.chris.student.model et met les types définis dans le paquet com.chris.student.service à la disposition des autres modules :

```
module com.chris.student.service {
    requires transitive com.chris.student.model;
    exports com.chris.student.service;
}
```

Nous fournissons un autre module com.chris.student.service.dbimpl, qui fournit l'implémentation com.chris.student.service.dbimpl.StudentDbService pour le module ci-dessus :

```
public class StudentDbService implements StudentService {

    public String create(Student student) {
        // Creating student in DB
        return student.getRegistrationId();
    }

    public Student read(String registrationId) {
        // Reading student from DB
        return new Student();
    }

    public Student update(Student student) {
        // Updating student in DB
        return student;
    }

    public String delete(String registrationId) {
        // Deleting student in DB
        return registrationId;
    }
}
```

Il dépend directement de com.chris.student.service et transitivement de com.chris.student.model et sa définition sera :

```
module com.chris.student.service.dbimpl {
    requires transitive com.chris.student.service;
    requires java.logging;
    exports com.chris.student.service.dbimpl;
}
```

Le dernier module est un module client - qui s'appuie sur le module de mise en œuvre du service com.chris.student.service.dbimpl pour effectuer ses opérations :

```
public class StudentClient {

    public static void main(String[] args) {
        StudentService service = new StudentDbService();
        service.create(new Student());
        service.read("17SS0001");
        service.update(new Student());
        service.delete("17SS0001");
    }
}
```

Et sa définition est la suivante :

```
module com.chris.student.client {
    requires com.chris.student.service.dbimpl;
}
```

Exportations de modules Java

Lorsque les paquets doivent être accessibles au code d'autres modules, nous devons exporter ces paquets à l'aide de la directive export. Lorsqu'un paquet est exporté, le module qui utilise le paquet exporté ne peut accéder qu'aux types publics de ce paquet. Lorsque le paquet est rendu accessible par l'exportation, tous ses sous-paquets sont également accessibles.

Syntaxe : `exports package;`

Exemple d'exportation de paquets à partir d'un module Java 9

```
module com.java4coding.app {
    exports com.java4coding.util ;
    }
```

Nous pouvons exporter un paquet vers un module particulier et seul le module spécifié peut accéder à ce paquet. Voici la syntaxe pour exporter un module vers des modules sélectionnés.

```
exports package to modulename;
```

Compiling and Running the Sample

Nous avons fourni des scripts pour compiler et exécuter les modules ci-dessus pour les plateformes Windows et Unix. Ils peuvent être trouvés dans le projet core-java-9 ici. L'ordre d'exécution pour la plateforme Windows est le suivant

compile-student-model

compile-student-service

compile-student-service-dbimpl

compile-student-client

run-student-client

L'ordre d'exécution pour la plateforme Linux est assez simple :

compile-modules

run-student-client

Dans les scripts ci-dessus, les deux arguments de ligne de commande suivants vous seront présentés :

-module-source-path

-module-path

Java 9 supprime le concept de classpath et introduit à la place le module path. Ce chemin est l'emplacement où les modules peuvent être découverts.

Nous pouvons le définir en utilisant l'argument de ligne de commande : -module-path.

Pour compiler plusieurs modules à la fois, nous utilisons l'argument -module-source-path. Cet argument est utilisé pour fournir l'emplacement du code source du module.

L'encapsulation forte et accessibilité (strong encapsulation)

Un des intérêts des modules est de permettre de renforcer l'encapsulation. Par défaut, aucune classe d'un module n'est accessible en dehors du module même si la classe est public. Pour permettre un accès à d'autres modules, il est nécessaire d'exporter le ou les packages concernés. Tous les autres packages non exportés ne sont accessibles uniquement que par le module lui-même.

Si un package est exporté à partir d'une classe, les règles de visibilité qui s'appliquent à la classe seront toujours en vigueur pour celle-ci.

Un module rend le contenu plus accessible et plus visible en même temps.

Parfois, la signification du modificateur public change. Par exemple, par défaut, une classe publique dans un module n'est accessible qu'aux autres classes de ce même module.

Cette classe publique est utilisée dans d'autres modules. Pour exporter ce package, vous devez exporter le package. Toutes les classes publiques d'un package exporté sont alors accessibles par d'autres modules qui déclareront le module en dépendance.

Ainsi pour permettre l'utilisation d'une classe d'un module par un autre module, trois contraintes doivent être respectées :

- la classe doit être public
- le package contenant la classe doit être exporté
- le module qui souhaite utiliser la classe doit déclarer le module de la classe en tant que dépendance

La mise en œuvre de ces trois règles permet un accès à une classe par un autre module.

Le fait de ne pas permettre par défaut un accès aux classes publiques d'un package permet de renforcer l'encapsulation. Il est par exemple possible de ne pas exposer des packages qui contiennent une implémentation et d'exposer les packages des interfaces.

L'encapsulation solide des modules renforcera la sécurité et assurera leur maintenance.

- le code important peut être enveloppé dans un module et rendu invisible même s'il se trouve dans une classe publique
- L'API publique d'un module peut être rationalisée
- Une application concrète consiste à utiliser JDK. Les API internes non standard sont encapsulées et ne sont donc plus accessibles par les applications.

Exercice

Créer un programme modulaire qui possède deux modules distincts , le premier sera le main, le second sera la classe Dog.

Cette classe possède une méthode Makenoise(), le main instancie la classe Dog et appellera la méthode makenoise.

Vous devrez respecter l'arborescence des fichiers propres à la programmation modulaire.

JShell: concepts

JShell est l'utilitaire Shell (ou **REPL**, pour Read Eval Print Loop) du langage de programmation Java.

Java manquait d'un outil adapté à cette fonction jusqu'à la version 9, mais il est maintenant disponible.

Le langage JShell est un moyen beaucoup plus rapide d'écrire du code en langage Java, en particulier pour les débutants.

Il permet, notament, d'effectuer de manière très simple, rapide et intéractive des tests d'exécution d'instructions Java (*snippets*).

Les bases de fonctionnement de JShell

JShell peut être utilisé de manière interactive sur la ligne de commande. Il possède une interface utile que vous pouvez utiliser pour taper des instructions, puis les exécuter.

Pour y accéder, utilisez la commande qui suit :

```
jshell
```

Suite à cela on obtient le *prompt* suivant qui attend la saisie de commandes à la syntaxe JShell ou Java.

La réponse de JShell à l'entrée d'une commande donnée est immédiate. Si la syntaxe est incorrecte ou si vous rencontrez une erreur d'exécution, un message d'erreur s'affichera et votre console reviendra à l'invite.

- Les commandes JShell sont précédées d'un '/'
- Toute commande introduite qui ne commence pas par ce caractère doit être conforme à la syntaxe Java

Les commandes JShell

Dans cette commande, nous utilisons le caractère barre oblique pour contrôler l'outil. Par exemple, '/help' fournira plus d'informations sur ce que font ces commandes

Affiche une liste exhaustive des commandes disponibles (relative à la version utilisée, bien entendu).Pour afficher l'aide d'une commande, vous devez d'abord connaître la syntaxe : /help [commande] . Par exemple, pour obtenir de l'aide pour 'exit', utilisez : /help exit.

Affiche l'aide sur la commande exit. Avec ces nouvelles informations, nous savons que nous devons utiliser la commande /exit pour quitter la console JShell.

Par exemple, la commande history peut vous dire quelles commandes vous avez exécutées précédemment dans une session.

```
/history
```

Cette commande affiche l'historique de toutes les commandes exécutées depuis le début de la session JShell en cours. Si tout se passe bien, le résultat devrait être :

```
/help
```

```
/help exit
```

```
/history
```

Exécuter une instruction Java

Dans la console JShell exécuter la commande:

```
exit
```

La réponse est un message d'erreur. Pourtant, on l'a vu, **exit** est bel et bien une commande correcte de JShell. L'explication est dans l'absence du caractère slash au début. Cette absence fait en sorte que JShell considère qu'il s'agit d'une instruction Java. **exit** n'est pas une instruction Java valide, d'où l'erreur.

Exécutons l’instruction : `"Java JShell".substring(5,11)`

La réponse renvoyée doit être: `$1 ==> "JShell"`

Cette réponse signifie qu’une variable appelée **$1** contient la chaine de caractère ‘**JShell**‘. Donc notre instruction précédente a bien été exécutée.

Explication: A l’exécution de l’instruction Java qui **retourne un résultat**, JShell crée une variable qui s’appelle **$x** ou x correspond a un nombre entier, et stoque le résultat de l’instruction dans celui ci.

Ensuite, il affiche le contenu de cette variable. Pour s’en rendre compte, exécuter maintenant:

```
System.out.println($1)
```

La méthode **substring(iDebut, iFin)** de la classe String, retourne une sous chaine de caractère qui commence à l'indice de cracatère **iDebut** et fini à l'indice **iFin**. D'où le résultat obtenu.

Exécuter maintenant l'instruction modifiée:

```
"Java JShell".substring(5,12)
```

L'indice de caractère (de fin) maximum dans cette chaine étant 11 (car elle contient 11 caractères), cette instruction lève l'exception **java.lang.StringIndexOutOfBoundsException** exactement comme à son exécution dans un programme Java.

Les variables dans JShell

Quand on exécute une instruction Java qui retourne une valeur, JShell crée automatiquement une variable pour y stocker cette valeur. Exécuter dans l'environnement JShell:

```
10 + 15
```

L'affichage de retour indique qu'une variable qui s'appelle **$1** contient le résultat de cette opération simple:

```
$1 ==> 25
```

On peut obtenir plus d’information à ce sujet grâce à la commande JShell:

```
/vars
```

Cette commande affiche la liste des variables en indiquant leurs types. On peut également améliorer le retour d’information immédiat. Exécuter:

```
/set feedback verbose
```

Si on exécute, après cela, l’opération d’addition modifiée:

```
10.2 + 15.3
```

Maintenant le retour immédiat est plus explicite. JShell nous informe tout de suite qu’il a créé une variable $2 de type double et affiche son contenu.

Remarquer que les variables créées implicitement portent un nom qui commence par le caractère $ suivi par un indice qui s’incrémente à chaque nouvelle création.

On remarque également que le type de la variable créée s'adapte au type du résultat de l'instruction exécutée.

Les variables peuvent être également indiquées explicitement avec un nom plus significatif. Dans ce cas le type doit être également indiqué:

```
double rayon = $2
```

Vérifier avec la commande /vars que le contenu de la variable rayon est bien égale à celui de la variable **$2**. Le résultat d'une instruction peut également être affecté

à une variable explicite:

```
double surface = Math.pow(rayon,2) * Math.PI
```

Ce qui calcule la surface d'un cercle en élevant son rayon au carré et en multipliant le résultat par **π** .

Vérifier le comportement de Java avec JShell

Une des grandes force de JShell est de pouvoir rapidement accéder aux résultats retournés par l'execution Java.

Exécuter les trois instructions suivantes:

```
Math.round(23.4999999)
```

```
Math.round(23.5000001)
```

```
Math.round(23.5000000)
```

Ce test montre que la methode Math.round(double) retourne bien la valeur arrondie à l'entier le plus proche du décimal passé en paramètre (les deux premières exécutions).Il montre surtout que si la valeur est à mi-chemin entre deux valeurs entières, c'est l'entier supérieur qui est retourné.

Maintenant, nous allons tester un comportement des instances de classes.

Mais d'abord partons d'un l'environnement propre.

Exécuter: `/vars`

`/list`

La session JShell continue de stocker la liste des variables et des instructions que nous avons précédemment exécuté.

Pour remettre l'environnement à zéro. Nous pouvons exécuter:

`/reset`

ensuite veuillez exécute les instructions dans l'ordre:

```
Date d1 = new Date()
Date d2 = d1
Date d3 = (Date)d1.clone()
/vars
```

L'explication du résultat, qui est assez courante, est que la variable qui contient une instance de classe ne pointe en fait que sur un emplacement mémoire où ces données de classe sont stockées.

Les variables contenant l'adresse mémoire d'une instance ne conteniennent pas l'instance elle-même.

Les trois déclarations font ce qui suit :

1. Créez une instance de Date et affectez la référence à cette instance (adresse mémoire à laquelle elle est allouée) à la variable d1
2. Attribue la valeur de d1 à d2. C'est la référence à l'instance qui est affectée. Donc d1 et d2 pointent vers la même instance.
3. La méthode clone() crée une nouvelle instance avec des données identiques à celles de la classe clonée. d3 contient donc une référence à une autre instance de la classe Date.

Tester le code d’une méthode

Par exemple, une de nos demandes est de créer une méthode qui calcule la clé d'un numéro de sécurité sociale en France.

Bien qu'il s'agisse d'un processus apparemment trivial, cette séquence d'instructions utilise plusieurs lignes de code distinctes. Pour entrer toutes les lignes, tapez simplement n'importe quel IDE et appuyez sur la touche [ENTER].

Commence par la première ligne:

```
long cle(String numero) {
```

JShell répond par :

```
...>
```

Et attendez le reste des lignes. Continuez à taper, ligne par ligne :

```
...> String n = numero.replaceAll("2A", "18");
```

```
...> n = n.replaceAll("2B", "19");
```

```
...> long ln = Long.parseLong(n);
```

```
...> long resultat = 97 - (ln % 97);
```

```
...> return resultat;
```

```
...> }
```

Lorsque nous entrons dans la dernière ligne qui constitue, syntaxiquement, la fin de la méthode, JShell se terminera et affiche un message nous informant (à condition qu'il n'y ait pas d'erreurs de saisie) que la méthode a été créée. Nous pouvons le vérifier en exécutant la commande :

```
/methods
```

On exécute également:

```
cle("2690299341732")
```

Si vous êtes née en corse vous obtiendrez une petite erreur que nous pourrons donc corriger ensemble. Pour le faire, on doit éditer la méthode. Nous pouvons l'éditer avec la commande edit, elle s’utilise comme ceci:

```
/set edit vi //Vi étant mon ide
```

Puis

```
/edit cle
```

L'éditeur paramétré s'ouvre avec le texte du code de la méthode et vous pouvez ensuite le modifier. Remplacez le nombre 19 de la 2e ligne par 18 et changez le nombre 18 de la 3e ligne par 19. Enregistrez ensuite les modifications et quittez.

Exercice

Créer, en Jshell, un script qui prend en paramètre un int, cet int représentera une note comprise entre 0 et 20, le script devra afficher une erreur si la note n'est pas comprise entre 0 et 20, elle affichera "très bien" pour l'intervalle 16 à 20, "bien" pour l'intervalle 14 à 16, "assez bien" pour 12 à 14, "moyen" pour 10 à 12, et insuffisant si la note est en dessous de 10.

Ajouts à l'API facultative de Java 9

- Les méthodes
  - or()
  - ifPresent()
  - ifPresentOrElse()
  - stream()

La méthode or()

Parfois, lorsque notre option est vide, nous voulons exécuter une autre action qui renvoie également une option.

Avant Java 9, la classe Optional ne disposait que des méthodes orElse() et orElseGet(), mais toutes deux devaient renvoyer des valeurs non enveloppées.

Java 9 introduit la méthode or() qui renvoie une autre option si notre option est vide.

Si notre premier Optional a une valeur définie, la lambda passée à la méthode or() ne sera pas invoquée, et la valeur ne sera pas calculée et retournée :

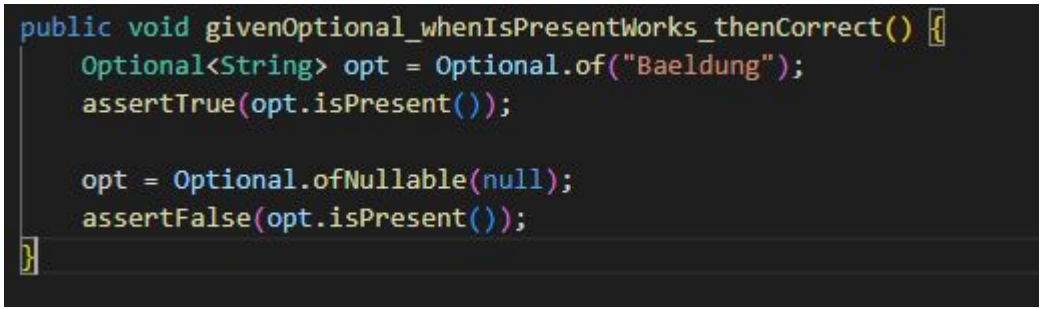
```java
public void givenOptional_whenPresent_thenShouldTakeAValueFromIt() {
    //given
    String expected = "properValue";
    Optional<String> value = Optional.of(expected);
    Optional<String> defaultValue = Optional.of("default");

    //when
    Optional<String> result = value.or(() -> defaultValue);

    //then
    assertThat(result.get()).isEqualTo(expected);
}
```

La méthode ifPresent()

Lorsque nous avons un objet Optional renvoyé par une méthode ou créé par nous, nous pouvons vérifier s'il contient une valeur ou non avec la méthode isPresent() :



```
public void givenOptional_whenIsPresentWorks_thenCorrect() {
    Optional<String> opt = Optional.of("Baeldung");
    assertTrue(opt.isPresent());

    opt = Optional.ofNullable(null);
    assertFalse(opt.isPresent());
}
```

Cette méthode renvoie un message vrai si la valeur wrapper n'est pas nulle.

La méthode ifPresentOrElse()

Lorsque nous disposons d'une instance Optional, nous voulons souvent exécuter une action spécifique sur la valeur sous-jacente de celle-ci.

D'autre part, si l'instance Optional est vide, nous voulons l'enregistrer ou suivre ce fait en incrémentant une certaine métrique.

La méthode ifPresentOrElse() a été créée exactement pour de tels scénarios. Nous pouvons passer un Consumer qui sera invoqué si l'option est définie, et un Runnable qui sera exécuté si l'option est vide.

Disons que nous avons définit un Optional et que l'on souhaite incrémenter un compteur si la valeur est présente:

```java
public void givenOptional_whenPresent_thenShouldExecuteProperCallback() {
    // given
    Optional<String> value = Optional.of("properValue");
    AtomicInteger successCounter = new AtomicInteger(0);
    AtomicInteger onEmptyOptionalCounter = new AtomicInteger(0);

    // when
    value.ifPresentOrElse(
      v -> successCounter.incrementAndGet(),
      onEmptyOptionalCounter::incrementAndGet);

    // then
    assertThat(successCounter.get()).isEqualTo(1);
    assertThat(onEmptyOptionalCounter.get()).isEqualTo(0);
}
```

Notez, que le callback passé comme deuxième argument n'a pas été exécuté.

Dans le cas d'un Optional vide, le deuxième callback est exécuté :

```
public void givenOptional_whenNotPresent_thenShouldExecuteProperCallback() {
    // given
    Optional<String> value = Optional.empty();
    AtomicInteger successCounter = new AtomicInteger(0);
    AtomicInteger onEmptyOptionalCounter = new AtomicInteger(0);

    // when
    value.ifPresentOrElse(
      v -> successCounter.incrementAndGet(),
      onEmptyOptionalCounter::incrementAndGet);

    // then
    assertThat(successCounter.get()).isEqualTo(0);
    assertThat(onEmptyOptionalCounter.get()).isEqualTo(1);
}
```

La méthode stream()

La dernière méthode, qui est ajoutée à la classe Optional dans le Java 9, est la méthode stream().

Java possède une API Stream très fluide et élégante qui peut opérer sur les collections et utilise de nombreux concepts de programmation fonctionnelle. La dernière version de Java introduit la méthode stream() sur la classe Optional qui nous permet de traiter l'instance Optional comme un Stream.

Disons que nous avons défini un Optional et que nous appelons la méthode stream() sur celui-ci.

Cela créera un flux d'un élément sur lequel nous pourrons utiliser toutes les méthodes disponibles dans l'API Stream :

```java
public void givenOptionalOfSome_whenToStream_thenShouldTreatItAsOneElementStream() {
    // given
    Optional<String> value = Optional.of("a");

    // when
    List<String> collect = value.stream().map(String::toUpperCase).collect(Collectors.toList());

    // then
    assertThat(collect).hasSameElementsAs(List.of("A"));
}
```

En revanche, si Optional n'est pas présent, l'appel de la méthode stream() sur celui-ci créera un Stream vide :

```java
public void givenOptionalOfNone_whenToStream_thenShouldTreatItAsZeroElementStream() {
    // given
    Optional<String> value = Optional.empty();

    // when
    List<String> collect = value.stream()
      .map(String::toUpperCase)
      .collect(Collectors.toList());

    // then
    assertThat(collect).isEmpty();
}
```

Nous pouvons maintenant filtrer rapidement les flux optionnels.

Opérer sur un flux vide n'aura aucun effet, mais grâce à la méthode stream(), nous pouvons désormais enchaîner l'API des optionnels avec l'API des flux. Cela nous permet de créer un code plus élégant et plus fluide.

exercice 1:

Créer une classe Book, qui possède une string nommée “Data”, cette classe aura une méthode set qui changera la valeur de Data.

Cette classe Book aura une seconde méthode: Search, qui affiche la string data is elle existe, elle affichera Data is null en cas contraire.

exercice 2 :

Réutiliser la classe Book, faite en sorte que l’absance de Data lors de l’utilisation de Search lance une methode qui met “Book” dans Data.

L'API Process

L'API de processus en Java était assez primitif avant Java 5. Avant cela, vous ne pouviez créer de nouveaux processus qu'en utilisant l'API Runtime.getRuntime().exec()

La première API ProcessBuilder a été introduite dans Java 5, ce qui la rend plus propre pour établir de nouveaux processus.

Java 9 ajoute un nouveau moyen d'obtenir des informations sur les processus actuels et ceux qui ont été créés.

Dans cet article, nous allons examiner ces deux améliorations.

Informations sur le processus Java actuel

Nous pouvons maintenant obtenir de nombreuses informations sur le processus via l'API java.lang.ProcessHandle.Info :

- la commande utilisée pour démarrer le processus
- les arguments de la commande
- l'instant où le processus a été lancé
- le temps total passé par celui-ci et l'utilisateur qui l'a créé

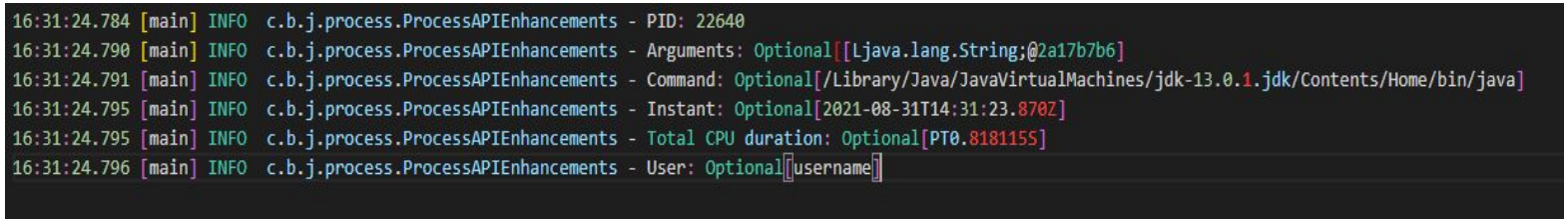
Voici comment nous pouvons le faire :

```
private static void infoOfCurrentProcess()
{
    ProcessHandle processHandle = ProcessHandle.current();
    ProcessHandle.Info processInfo = processHandle.info();

    log.info("PID: " + processHandle.pid());
    log.info("Arguments: " + processInfo.arguments());
    log.info("Command: " + processInfo.command());
    log.info("Instant: " + processInfo.startInstant());
    log.info("Total CPU duration: " + processInfo.totalCpuDuration());
    log.info("User: " + processInfo.user());
}
```

Il est important de noter que java.lang.ProcessHandle.Info est une interface publique définie dans une autre interface java.lang.ProcessHandle. Le fournisseur de JDK (Oracle JDK, Open JDK, Zulu ou autres) doit fournir des implémentations de ces interfaces de manière à ce que ces implémentations renvoient les informations pertinentes pour les processus.

Le résultat dépend du système d'exploitation et de la version de Java. Voici un exemple de ce à quoi la sortie peut ressembler :



Création du processus

Notre application Java peut faire appel à toute application en cours d'exécution dans notre système informatique, sous réserve des restrictions du système d'exploitation.

Nous pouvons donc exécuter des applications. Voyons quels sont les différents cas d'utilisation que nous pouvons exécuter en utilisant l'API de processus.

La classe ProcessBuilder nous permet de créer des sous-processus dans notre application.

Voyons une démonstration de l'ouverture de l'application Notepad sous Windows :

```
ProcessBuilder builder = new ProcessBuilder("notepad.exe");
Process process = builder.start();
```

destructeur de processus

Supposons que nous utilisions le système d'exploitation Windows et que nous voulons détruire l’application lancée précédemment.

Nous pouvons appeler la méthode destroy() sur notre objet Process pour l'arrêter.

Nous pouvons également tuer les processus qui sont en cours d'exécution dans notre système d'exploitation et qui pourraient ne pas être créés par notre application.

Il convient d'être prudent lors de cette opération, car nous pouvons détruire sans le savoir un processus critique qui pourrait rendre le système d'exploitation instable.

Nous devons d'abord trouver l'ID du processus en cours d'exécution en vérifiant le gestionnaire des tâches et trouver le pid.

Lors de l'exécution de la méthode destroy(), le sous-processus sera tué comme nous l'avons vu plus haut.

Dans le cas où destroy() ne fonctionne pas, nous avons l'option de destroyForcibly().

Nous devrions toujours commencer par la méthode destroy(). Après cela, nous pouvons effectuer une vérification rapide sur le sous-processus en exécutant isAlive().

S'il retourne vrai, alors on exécutera destroyForcibly();

Informations sur les processus créés

Il est également possible d'obtenir les informations d'un processus nouvellement créé. Dans ce cas, après avoir créé le processus et obtenu une instance de java.lang.Process, nous invoquons la méthode toHandle() pour obtenir une instance de java.lang.ProcessHandle.

Le reste des détails reste le même que dans la section ci-dessus :

```
String javaCmd = ProcessUtils.getJavaCmd().getAbsolutePath();
ProcessBuilder processBuilder = new ProcessBuilder(javaCmd, "-version");
Process process = processBuilder.inheritIO().start();
ProcessHandle processHandle = process.toHandle();
```

Enumération des processus vivants dans le système

Nous pouvons lister tous les processus actuellement présents dans le système, qui sont visibles par le processus actuel. La liste retournée est un instantané au moment où l'API a été invoquée, il est donc possible que certains processus se soient terminés après la prise de l'instantané ou que de nouveaux processus aient été ajoutés.

Pour ce faire, nous pouvons utiliser la méthode statique allProcesses() disponible dans l'interface java.lang.ProcessHandle qui nous renvoie un Stream de ProcessHandle :

```
private static void infoOfLiveProcesses() {
    Stream<ProcessHandle> liveProcesses = ProcessHandle.allProcesses();
    liveProcesses.filter(ProcessHandle::isAlive)
        .forEach(ph -> {
            log.info("PID: " + ph.pid());
            log.info("Instance: " + ph.info().startInstant());
            log.info("User: " + ph.info().user());
        });
}
```

Enumération des processus enfants

Il y a deux variantes pour faire cela :

- obtenir les enfants directs du processus actuel
- obtenir tous les descendants du processus courant

La première est réalisée en utilisant la méthode children() et la seconde est réalisée en utilisant la méthode descendants() :

```java
private static void infoOfChildProcess() throws IOException
{
    int childProcessCount = 5;
    for (int i = 0; i < childProcessCount; i++)
    {
        String javaCmd = ProcessUtils.getJavaCmd().getAbsolutePath();
        ProcessBuilder processBuilder = new ProcessBuilder(javaCmd, "-version");
        processBuilder.inheritIO().start();
    }

    Stream<ProcessHandle> children = ProcessHandle.current().children();
    children.filter(ProcessHandle::isAlive).forEach(ph -> log.info("PID: {}, Cmd: {}", ph.pid(), ph.info().command()));
    Stream<ProcessHandle> descendants = ProcessHandle.current().descendants();
    descendants.filter(ProcessHandle::isAlive).forEach(ph -> log.info("PID: {}, Cmd: {}", ph.pid(), ph.info().command()));
}
```

Déclenchement d'actions dépendantes à la fin d'un processus

On peut vouloir exécuter quelque chose à la fin du processus. Ceci peut être réalisé en utilisant la méthode onExit() de l'interface java.lang.ProcessHandle. Cette méthode nous renvoie un CompletableFuture qui permet de déclencher des opérations dépendantes lorsque le CompletableFuture est terminé.

Ici, le CompletableFuture indique que le processus est terminé, mais il importe peu que le processus se soit terminé avec succès ou non. Nous invoquons la méthode get() sur le CompletableFuture, pour attendre son achèvement :

```java
private static void infoOfExitCallback() throws IOException, InterruptedException, ExecutionException {
    String javaCmd = ProcessUtils.getJavaCmd().getAbsolutePath();
    ProcessBuilder processBuilder = new ProcessBuilder(javaCmd, "-version");
    Process process = processBuilder.inheritIO().start();
    ProcessHandle processHandle = process.toHandle();

    log.info("PID: {} has started", processHandle.pid());
    CompletableFuture onProcessExit = processHandle.onExit();
    onProcessExit.get();
    log.info("Alive: " + processHandle.isAlive());
    onProcessExit.thenAccept(ph -> {log.info("PID: {} has stopped", ph.pid());});
}
```

La méthode onExit() est également disponible dans l'interface java.lang.Process.

exercice 1:

Créer un second programme qui exécute un processus, et renvoie les informations sur son processus, et les affiches.

Ce dernier laisse le programme fonctionner, nous devons le kill nous même.

exercice 2:

reprenons ensuite le même programme, ou nous terminerons nous même le processus après 5 secondes.

Conclusion

Dans ce cours, nous avons abordé les ajouts intéressants apportés à l'API de processus dans Java 9, qui nous permettent de mieux contrôler les processus en cours d'exécution et les processus créés.