FORMATION

Java - Nouveautés des versions 8 à 17

Votre formateur

François Caremoli

Développeur pendant 7 ans sur des applications Java/JavaEE.
Profil full stack (du SQL ↔ HTML ou HTTP)

Lead Developper pendant 8 ans

Architecte SI pendant 2 ans.

Formateur en parallèle.

Et vous ?

- Présentation
  - Les langages et technologies que vous utilisez habituellement
  - Ce que vous connaissez de la programmation en général
  - Vos attentes vis-à-vis de la formation

- Prérequis
  - Droits administrateurs sur votre poste
  - Capacités à développer en Java

Ouverture de Slack

- Slack est un outil de chat. Il est utilisé par de nombreux développeurs. Il sera utilisé pour servir d'espace de discussion, de partage de documents, et d'historique à la formation. Il peut être utilisé en tant qu'application téléchargée, ou sur le navigateur.
- Vous avez dû recevoir par mail un lien d'invitation à Slack
- Rejoignez Slack et réagissez au message de bienvenu.

Supports de cours

- Tout est sur Slack.
- Le PPT du cours est épinglé en haut de la discussion.

Horaires, temps de pause et dossier pédagogique

- Les heures de cours sont notées sur la convocation.
- Il y a une pause de 15mn le matin et une pause de 15mn l'après midi.
- Et une pause déjeuner de 1h de 12h30 à 13h30.

- La formation sera une suite de notions à assimiler, d'exercices mettant en pratique ces notions et d'un ou plusieurs TP laissant plus de créativité au stagiaire pour utiliser ces connaissances dans un cadre plus pratique.

Plan de cours

- Les moments principaux du cours seront :
  - Le rappel des apports sur la programmation concurrente (Java 7)
  - Les lambdas et les streams (Java 8)
  - DateTime (Java 8)
  - Les modules (Java 9)
  - JShell (Java 9)
  - Optional (Java 9)
  - Process (Java 9)
  - Les apports des versions 10 à 17

Téléchargement et installation des composants

- Télécharger et décompresser :
- OpenJDK 17.0.2 : https://jdk.java.net/archive/
- Un client Git à jour. Vérifier que vous pouvez cloner le repo de la formation.

- Un IDE . La formation se fera avec Eclipse en version Entreprise, mais n’importe quel IDE fera l’affaire.

Programmation concurrente

Les concepts de la programmation multi-thread

- Le but de la programmation multi-thread est de simuler, ou de réaliser des opérations en parallèle sur une machine dotée de un à plusieurs microprocesseurs (ou coeurs).
- Ceci permet de distribuer la puissance de calcul du (des) microprocesseur(s) entre plusieurs applications, ou pour une même application, entre plusieurs tâches.
- Pour ce faire, le sytème d'exploitation crée :
  - Des process, environnements d'exécution indépendants, ayant leur propre espace mémoire (entre autres). Plus lourds à créer que les threads, ils nécessitent des pipes ou des sockets pour communiquer entre eux.
  - Des threads, environnements d'exécution partageant l'espace mémoire, les fichiers ouverts, les ressources réseau … Un process contient au moins un thread.

- De base, la JVM crée un process et un thread, bien qu'il soit possible de créer plusieurs process, le but de ce chapitre est de gérer des applications multi-tâches avec de nombreux threads.

Les Threads en Java : héritage de Thread

La première manière de créer des Threads en Java consiste en :

- **Créer une classe qui hérite de java.lang.Thread et surcharger sa méthode run**

```
private static final class LongTaskThread extends Thread {
    @Override
    public void run() {
        System.out.println("Démarrage LongTaskThread");
        System.out.println("Fin LongTaskThread");
    }
    }
```

- **Puis l'instancier et lancer start()**

```
LongTaskThread longTaskThread = new LongTaskThread();
longTaskThread.start();
```

Les Threads en Java : implémentation de Runnable

La seconde manière de créer des Threads en Java consiste en :

- Créer une classe qui implémente java.lang.Runnable et implémenter sa méthode run

```
private static final class LongTaskRunnable implements Runnable {

	@Override
	public void run() {
		System.out.println("Démarrage LongTaskRunnable");
		System.out.println("Fin LongTaskRunnable");
	}
}
```

- Puis créer un Thread avec ce Runnable et lancer start() sur le Thread créé :

```
new Thread(new LongTaskRunnable()).start();
```

- La seconde version est préférée, puisque l'implémentation ne force pas à hériter d'une classe (ce qui peut bloquer si l'on veut créer une hiérarchie de classes pour les Threads).

Le package java.util.concurrent

- Depuis Java5, créer ses Threads soi-même et les synchroniser avec “synchronized” n’est plus la norme.
- Il vaut mieux utiliser les classes et interfaces fournies par java.util.concurrent.
- L’objet de cette partie du cours sera de décrire quelques classes (ou interfaces) utiles dans ce package.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/package-summary.html

Callable

- Callable<> est un équivalent à Runnable, à l'exception qu'il renvoie un objet.
- Il peut être encapsulé dans une instance de FutureTask<>, ce qui permet :
  - De le lancer dans un Thread
  - De récupérer le résultat avec FutureTask.get()
- L'appel à FutureTask.get() bloque le Thread appelant, tant que call() n'est pas terminé.

```
private static final class LongTaskCallable
      implements Callable<Long> {
    @Override
    public Long call() {
         System.out.println("Démarrage LongTaskCallable");
         try {
               Thread.sleep(2000);
         } catch (InterruptedException e) {
               return 0l;
         }
         System.out.println("Fin LongTaskCallable");
         return 42l;
    }
    }
```

Callable

```
System.out.println("Démarrage Thread principal");

Callable<Long> callable = new LongTaskCallable();
FutureTask<Long> futureTask = new
FutureTask<>(callable);
new Thread(futureTask).start();
try {
     Long result = futureTask.get();
     System.out.println(String.format("Le résultat
     vaut : %s", result));
} catch (InterruptedException | ExecutionException e) {
throw new RuntimeException(e);
}

System.out.println("Fin Thread principal");
```

Code appelant

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html

Exécuteurs

Les interfaces de java.util.concurrent qui héritent de Executor aident à gérer les Threads à un plus haut niveau que lorsque l'on les crée, les joint, les synchronise … Un Executor peut lancer des instances de Runnable avec le code suivant :

```
Runnable runnable = new LongTaskRunnable();
Executor executor =
Executors.newSingleThreadExecutor();
executor.execute(runnable);
```

On préfèrera toutefois utiliser l'interface ExecutorService, plus complète (qui permet d'arrêter le Thread par exemple :), et qui permet d'appeler des Callables. Le code ci-dessous permet de lancer cinq fois LongTaskRunnable via un pool de 3 threads (ce qui rallongera la durée de l'opération mais permet de garder la main sur le nombre de Threads créés).

```
Runnable runnable = new LongTaskRunnable();
ExecutorService executorService =
Executors.newFixedThreadPool(3);
executorService.execute(runnable);
executorService.execute(runnable);
executorService.execute(runnable);
executorService.execute(runnable);
executorService.shutdown();
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html

Exercice : utiliser ExecutorService

Créer un Callable renvoyant un nombre aléatoire , et qui attend entre 5 et 10 secondes avant de le renvoyer. Grâce à un ExecutorService, lancer 10 fois le callable et afficher la première valeur retournée. Utiliser un FixedThreadPool pour créer l'ExecutorService.

Bonus : utiliser un FixedThreadPool et le dimensionner correctement pour l'exercice.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html

Les barrières cycliques

- **Une barrière cyclique (CyclicBarrier) est une barrière réutilisable (d'où le terme cyclique). Elle fonctionne comme suit :**
  - **Elle est initialisée avec un nombre de threads à attendre, et (optionnellement), une action à effectuer quand ce nombre est atteint.**
  - **Chaque Thread peut appeler : CyclicBarrier.await(), ce qui incrémente le nombre d'attentes dans la barrière.**
  - **Si le nombres d'attentes dans la barrière est inférieur au nombre de threads à attendre, le Thread est BLOCKED.**
  - **Sinon, la barrière s'ouvre, l'action optionnelle est effectuée, puis tous les threads bloqués repassent en RUNNABLE.**

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

Les barrières cycliques : exemple

```
public static void main(String[] args)
      throws InterruptedException {

     System.out.println("Démarrage Thread principal");
     CyclicBarrier cyclicBarrier = new CyclicBarrier(5,
           new Announcement());
     new Thread(new Runner(cyclicBarrier)).start();
     new Thread(new Runner(cyclicBarrier)).start();
     new Thread(new Runner(cyclicBarrier)).start();
     new Thread(new Runner(cyclicBarrier)).start();
     Thread.sleep(3000);
     new Thread(new Runner(cyclicBarrier)).start();
     System.out.println("Fin Thread principal");

}
```

```
private static final class Runner implements Runnable {

private CyclicBarrier barrier;

public Runner(CyclicBarrier barrier) {
      this.barrier = barrier;
}

@Override
public void run() {
     System.out.println("Démarrage Runner");
     try {
     System.out.println("Le coureur se place sur la piste");
     this.barrier.await();
     System.out.println("Il court !");
     Thread.sleep((long) (Math.random() * 10000));
     System.out.println("Il arrive !");
     } catch (InterruptedException | BrokenBarrierException
     e) {
     return;
     }
     System.out.println("Fin Runner");
     }
}
```

Exercice : synchroniser des Threads avec une barrière cyclique

- Créer une implémentation de Runnable, dont run() affiche une ligne sur la console. run() prend un nombre aléatoire de secondes pour s'exécuter (entre 0 et 10).
- Utiliser une CyclicBarrier, à injecter dans l'implémentation de Runnable, pour s'assurer que tous les Runnable se lancent en même temps.
- Dans le main(), lancer assez de Runnable pour déclencher l'ouverture de la barrière.
- Bonus : lancer un autre Runnable quand la barrière s'ouvre (il affiche une ligne)
- Bonus++ : avec deux barrières (une de démarrage, une de fin), tenter de mettre ces Threads dans une boucle, pour qu'ils se lancent continûment, mais de façon synchronisée.

L’interface Lock

Le package java.util.concurrent contient une interface : Lock, qui permet de créer des verrous en instanciant des implémentations de Lock (comme ReentrantLock). Cette version “objet” des verrous permet de :

- Créer des verrous hors d’un block synchronized avec lock()
- Tenter de créer un verrou sans bloquer le Thread courant (ou en bloquant uniquement un certain temps), avec tryLock()
- Créer un lock jusqu’à ce que le Thread en question soit interrompu

- Ceci peut permettre d’éviter des deadlocks, en tentant par exemple de verrouiller deux objets. Si le verrouillage est permis, la méthode ‘synchronisée’ peut être appelée, sinon, le Thread ne fait rien (et ne bloque personne).

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

Les sémaphores

- **Semaphore est une classe de java.util.concurrent. Un sémaphore est initialisé avec un nombre de permis, et permet à un objet appelant d'acquérir un permis (donc de décrémenter le nombre de permis) du sémaphore, et de rendre un permis (donc d'incrémenter le nombre de permis du sémaphore). Ce mode permet aux sémaphores de restreindre l'accès à une ressource à un nombre limité de Threads :**
- **Acquérir un permis se fait avec acquire(). Si le nombre de permis est supérieur à 0, ce nombre est décrémenté, sinon, le Thread appelant est bloqué.**
- **Rendre un permis se fait avec release(), ce qui augmente le nombre de permis.**
- **Release() peut donc débloquer les Threads bloqués sur acquire().**
- **Selon la justesse (fairness) du sémaphore, celui-ci peut débloquer les Threads dans l'ordre dans lequel ils ont été bloqués, ou non.**

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

Fork/Join

- **Fork/join est une implémentation d'ExecutorService qui permet d'utiliser efficacement les multiples coeurs d'une machine en divisant une grosse tâche en tâches plus petites. Le gain ne se fera que si les petites tâches sont indépendantes. Ainsi, les calculs pourront être parallélisés.**
- **On peut utiliser pour ce faire ForkJoinPool (un ExecutorService), qui va exécuter des ForkJoinTasks.**
- **On peut aussi (plus généralement) utiliser des méthodes qui implémentent déjà cette implémentation,avec Arrays.parallelSort(), ou les streams des lambdas (vus plus tard)**

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

TP : créer un pool de connexions

- La classe com.bigcorp.concurrent.HeavyResource existe dans le squelette du projet. Elle représente une ressource sur laquelle on fait des opérations longues.
- Pour chaque ressource, on peut appeler beginTransaction() et endTransaction(). Appeler endTransaction() sur une ressource avant d'appeler beginTransaction() (ou l'inverse) lance une exception. endTransaction() et beginTransaction() mettent 100ms à s'exécuter chacune.
- La classe ConnectionPoolTP contient une méthode main qui des appelle en série heavyResourceA, B, ou C (aléatoirement). Mais avec ce fonctionnement, on perd trop de temps à attendre que les HeavyResources répondent.
- Faire en sorte que le main passe par une méthode qui gère en parallèle, et sans erreur l'accès aux trois ressources.
- Bonus : comparer les temps des deux méthodes (séquentielle et parallèle).
- Bonus++ : Rajouter le fait qu'on ne peut appeler plus de deux ressources en même temps.

Ce qu’il faut retenir

- La programmation concurrente en Java, consiste en la manipulation de Threads, à bas niveau.
- Néanmoins, java.util.concurrent propose un grand nombre de classes pour gérer ces threads à plus haut niveau.
  - Les exécuteurs permettent de lancer des tâches en parallèle sur un certain nombre de threads
  - Les barrières cycliques et sémaphores permettent de synchroniser les threads avec des objets de haut niveau ...

Les conseils du formateur

Avant d’utiliser synchronize ou de lancer des Threads directement, allez voir dans java.util.concurrent si une classe ne correspond pas à vos besoins.

Lambdas & streams

Transmission de comportement en Java

- Il est parfois intéressant de fournir un comportement à une méthode.
- Par exemple, pour une interface homme-machine, ajouter un bouton dans une fenêtre, dont le rôle serait de fermer la fenêtre se ferait ainsi :

```
window.addButton(new CloseWindowButton());

avec :

public class CloseWindowButton implements Button

public interface Button{
    String click(Winow window);
}

public class Window{
    public void addButton(Button button){
....
```

Problème rencontré

- Le code précédent utilise le polymorphisme et est tout à fait valable en Java.
- Il sert uniquement à ajouter un comportement à la fenêtre : celui d'un bouton qui ferme la fenêtre.
- Pour créer un autre comportement, il faut créer une classe qui implémente Button. Ceci peut rendre le code verbeux si chaque bouton a un comportement différent. On se retrouve avec une classe par bouton, qui ne servent pas à grand chose.

Implémentation anonyme

- Pour rendre le code moins verbeux, et dispenser le développeur de créer autant de classes qu'il ne veut créer de boutons, Java propose depuis longtemps d'implémenter dans une classe anonyme des interfaces.

```
window.addButton(
     new Button(){

          public String click(Window window){
              //implémentation ici...
          }

});
```

Implémentation anonyme : problèmes

- Le code précédent ne force pas le développeur à créer de nombreuses classes, mais est peu lisible, et plutôt verbeux.
- Il utilise les principes de programmation objet, ce qui est bien, en théorie. Mais ici, ces principes sont détournés pour passer un comportement à une méthode (ou à une classe).
- Le mieux serait de passer ce comportement directement. Par exemple, en manipulant les fonctions comme n'importe quelle autre variable.
- La manipulation de fonctions comme toute autre variable est la programmation fonctionnelle. Comment l'implémenter en Java ?

Programmation fonctionnellle

- La manipulation de fonctions comme toute autre variable est la programmation fonctionnelle.
- Comment l’implémenter en Java ?

Lambdas

- Java 8 intègre les lambdas, des blocs de code courts représentant des méthodes.
- Ces méthodes prennent 0 ou plusieurs paramètres et renvoient une valeur ou void.
- Ci-dessous un exemple de lambda avec un bloc

```
(type1 parameter1, type2 parameter2) -> { return parameter1 };
```

- On peut stocker ces méthode dans une variable (de type Consumer, BiConsumer ....), ou les utiliser telles quelles.

```
Consumer<String> method = ( n) -> { System.out.println(n); };
```

- L'intérêt des lambdas est qu'elles peuvent être utilisées en paramètre de méthodes, pour paramétriser un traitement. Elles peuvent servir à de la programmation fonctionnelle.

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Interface fonctionnelle

- Les lambdas peuvent être stockées dans des variables. Or, une variable en Java a un type, et ce type n'est pas 'méthode'.
- Une lambda est considérée par Java comme LA méthode d'une interface ne proposant qu'une seule méthode public abstract. Ce genre d'interface s'appelle "interface fonctionnelle".
- Une annotation @FunctionalInterface a été introduite avec les lambdas pour 'marquer' ces interfaces. Elle n'est pas obligatoire, mais permet de détecter des erreurs dès la compilation.
- Si, dans un code Java, le compilateur détecte qu'une lambda est utilisée à la place d'une interface fonctionnelle, il peut déduire la méthode que la lambda remplace (il n'y en a qu'une dans l'interface fonctionnelle), et de là, les types de retour et des arguments (puisque la signature de la méthode est complètement définie dans l'interface).

Lambdas : arguments

- Les arguments de la lambda sont placés à gauche du symbole -> (tiret puis symbole supérieur).
- Comme dans toute signature de méthode, ils sont contenus dans des parenthèses et séparés par des virgules.
- Le type des arguments est optionnel : en effet, la signature de la méthode de l'interface fonctionnelle contient déjà l'information de ces types : le compilateur peut donc la deviner.
- Si un seul argument est présent, les parenthèses aussi sont optionnelles.

Lambdas : corps d’une expression

- Le corps d'une expression lambda est défini à droite de l'opérateur ->. Il peut être :
  - une expression unique
  - un bloc de code composé d'une ou plusieurs instructions entourées par des accolades

- Le corps de l'expression doit respecter certaines règles :
  - il peut n'avoir aucune, une seule ou plusieurs instructions
  - lorsqu'il ne contient qu'une seule instruction, les accolades et le mot clé return ne sont pas obligatoires et la valeur de retour est celle de l'instruction si elle en possède une.
  - lorsqu'il y a plusieurs instructions, elles doivent obligatoirement être entourées d'accolades.

Lambdas : les variables

Dans le corps d'une expression lambda, il est possible d'utiliser :

- Les variables passées en paramètre de l'expression
- Les variables définies dans le corps de l'expression
- Les variables **final** définies dans le contexte englobant
- Les variables effectivement final définies dans le contexte englobant : ces variables ne sont pas déclarées final, mais une valeur leur est assignée et celle-ci n'est jamais modifiée. Il serait donc possible de les déclarer final sans que cela n'engendre de problème de compilation. Le concept de variables effectivement final a été introduit dans Java 8. Ce sont par exemple les arguments de la méthode qui contient la lambda.

Exercice : Lambdas

Le package lambda contient les classes suivantes :

- Machine : une machine qui traite de la matière avec la méthode travaille
- Traitement : une interface fonctionnelle qui définit comment traiter une matière
- Matière : un simple POJO qui contient un prix, un nom et une masse
- Usine : la classe qui contient le main, et dans laquelle on va instancier des machines et les faire traiter des matériaux. Dans cette usine, les machines travaillent avec de nombreux traitements. On ne veut pas instancier une classe par traitement.

Modifier la méthode main de la classe usine pour appeler la méthode travaille() avec une lambda.

Références de méthodes : concept

- Si une méthode attend une interface fonctionnelle en argument, il est aussi possible de lui passer directement une méthode d'une autre classe à la place de cet argument.
- La syntaxe pour ce faire est Class::method ou object::method
- Ceci permet de raccourcir la syntaxe des lambdas :

```
List withoutBlanks = sampleList.stream().filter(s -> s.isBlank())

// equivalent à :

List withoutBlanks = sampleList.stream().filter(String::isBlank)
```

Exercice : Références de méthodes

- Dans Usine, utiliser System.out.println() pour qu'une machine affiche le matériau qu'elle contient, en utilisant une lambda qui appelle System.out.println()
- Faire en sorte d'appeler System.out.println() sur la machine, en utilisant une référence de méthode

- Bonus : appeler la méthode afficheNom de la matière grâce à une référence de méthode.

Utilisation des lambdas : les streams

L'API Stream permet de réaliser des opérations fonctionnelles sur un ensemble d'éléments. De nombreuses opérations de Stream attendent en paramètre une interface fonctionnelle.
Ceci conduit naturellement à utiliser les expressions lambdas et les références de méthodes dans la définition des Streams. Un Stream permet donc d'exécuter des opérations standards dont les traitements sont exprimés grâce à des expressions lambdas ou des références de méthodes.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Streams : généralités

- Les Streams permettent de traiter des objets. Pour traiter les types primitifs int, long et double, il existe IntStream, LongStream et DoubleStream.
- Un stream ne stocke pas d'éléments. Il traite et transporte des éléments d'une source (collection, entrée sortie, générateur ...) vers une sortie.
- Les opérations des streams ne modifient pas leurs sources.
- Comportement paresseux : de nombreuses opérations comme le filtrage, mapping, ou la suppression des doublons peuvent être implémentées de façon paresseuse (lazy). Ceci permet des optimisations sur les traitements.
- Les opérations de streams sont divisées entre opérations intermédiaires et terminales. Les opérations intermédiaires sont toujours paresseuses.
- Les streams peuvent travailler sur des données infinies. Les méthodes limit(n) et findFirst() permettent de traiter ces données.

Streams : obtention

Il est possible d'obtenir des streams via :

- une Collection grâce à stream() et parallelStream()
- un tableau grâce à Arrays.stream(Object[])
- les classes de Stream : Stream.of(Object[]), IntStream.range(int, int), Stream.iterate(Object, UnaryOperator)
- les lignes d'un fichier : BufferedReader.lines()
- les streams de chemins de fichiers sont obtenus grâce aux méthode de Files
- une liste aléatoire : Random.ints()
- d'autres méthodes du JDK : BitSet.stream() Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream()
- d'autres méthodes d'autres bibliothèques ...

Streams: première méthode

- En passant une lambda à forEach, on peut exécuter une méthode pour chaque élément d'un stream:

```
List<Integer> entiers = new ArrayList<>();
entiers.add(1);
entiers.add(2);
entiers.stream().forEach(i ->
System.out.println(i));
```

- En chaînant les streams, et en ajoutant deux lambdas, on peut afficher un message sur les entiers dont la valeur vaut 1:

```
entiers.stream()
.filter(v -> v == 1)
.forEach(i ->
System.out.println(i));
```

Streams: d'autres méthodes

- On peut mapper (avec les méthodes map...) pour transformer les éléments du stream, puis utiliser une méthode terminale pour agréger des résultats :

```
long somme = entiers.stream()
                    .filter(v -> v
<10)
                    .mapToInt(i -> i)
                    .sum();
```

- On peut aussi reconstruire une collection avec l'aide de la classe utilitaire Collectors :

```
List<Integer> newList =
entiers.stream()
.filter(v -> v < 10)
.collect(Collectors.toList());
```

Finalement, on peut simplement lancer le calcul en parallèle :

```
List<Integer> newList =
entiers.parallelStream()
.filter(v -> v < 10)
.collect(Collectors.toList());
```

Streams: opérations de filtre

- filter(Predicate) : renvoie un Stream qui contient les éléments pour lesquels l'évaluation du Predicate passé en paramètre vaut true
- distinct() : renvoie un Stream qui ne contient que les éléments uniques (elle retire les doublons). La comparaison se fait grâce à l'implémentation de la méthode equals()
- limit(n) : renvoie un Stream que ne contient comme éléments que le nombre fourni en paramètre
- skip(n) : renvoie un Stream dont les n premiers éléments sont ignorés

Streams: opérations de transformation

- map(Function) : applique la Function fournie en paramètre pour transformer l'élément en créant un nouveau
- flatMap(Function) : applique la Function fournie en paramètre pour transformer l'élément en créant zéro, un ou plusieurs éléments

Streams: opérations de recherche

- anyMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur au moins un élément vaut true
- allMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut true
- noneMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut false
- findAny() : renvoie un objet de type Optional qui encapsule un élément du Stream s'il existe
- findFirst() : renvoie un objet de type Optional qui encapsule le premier élément du Stream s'il existe

Streams: opérations de réduction

- reduce() : applique une Function pour combiner les éléments afin de produire le résultat
- collect() : permet de transformer un Stream qui contiendra le résultat des traitements de réduction dans un conteneur mutable

TP : Lambdas et Streams

- Créer une classe Child.java, contenant un nom et un age.
- Instancier dans une collection une dizaine d'instances de Child.
- Afficher le nom de chaque enfant avec un stream.
- Utiliser les streams pour afficher le nom de l'enfant ayant le plus grand âge.
- Pareil pour le plus petit.
- Afficher la moyenne d'âge des enfants.
- Afficher si au moins un enfant a dépassé un âge.

- Certaines méthodes (comme min) renvoient un Optional. Utilisez get() pour récupérer le 'vrai' objet contenu dans l'Optional. La classe Optional sera vue dans un chapitre ultérieur.

Ce qu’il faut retenir

- Les lambdas permettent de faire du développement fonctionnel en Java.
- Actuellement, le plus grand utilisateur de lambdas reste l’API stream().
- Celle-ci permet d’enchaîner des opérations sur les collections de manière simple (à la manière d’un SQL).

Les conseils du formateur

Savoir développer avec des streams est une bonne chose, mais ne doit pas être systématique. Si le code d’un stream vous paraît au final plus compliqué que celui d’une boucle, posez-vous la question de l’intérêt de l’utilisation de Stream.

DateTime

Java.util.Date

- Historiquement, les dates étaient modélisées en Java via les classes Java.util.Date, Calendar et Timezone.
- Ces classes n’étaient pas très pratiques.
- Elles étaient mutables, donc pas directement compatible avec du code multithread.
- Le fait que de nombreux développeurs et frameworks utilisaient des bibliothèques comme joda.time montre que ce qu’offrait Java ne répondant pas aux besoins des utilisateurs.
- ... on peut encore s’en servir, mais il y a mieux ....

Le package java.time

- Il contient des classes comme : LocalDate, LocalDateTime, LocalTime, YearMonth, MonthDay, Year, Instant.
- Il y a deux manières de représenter le temps :
  - la manière lisible par un humain,avec les années, mois, jours, heures, minutes, secondes ....
  - celle lisible par une machine, qui mesure le temps à partir d'une date : l''epoch', en nanosecondes. L'epoch est le 01/01/1970 à minuit
- java.time contient des classes permettant de gérer le temps de ces deux manières.
- Dans la suite du cours, par défaut, les jours et mois et calendriers seront ceux du calendrier grégorien.

https://docs.oracle.com/javase/tutorial/datetime/iso/index.html

Quelle classe utiliser ?

- Avez-vous besoin d'une date que des humains vont manipuler? Ou uniquement pour des machines. Avez-vous besoin d'une date avec zone temporelle ? D'une date, d'un temps ? Uniquement d'un mois ? Selon ces réponses, vous saurez quelle classe utiliser.
- Le tableau présent sur cette page peut vous aider : https://docs.oracle.com/javase/tutorial/datetime/iso/overview.html

https://docs.oracle.com/javase/tutorial/datetime/iso/overview.html

Mois et jours de la semaine (enums)

- DayOfWeek est une enum contenant les septs jours de la semaine, avec un nom en anglais et une valeur numérique (lundi valant 1). Elle contient aussi des méthodes utilitaires. DayOfWeek.TUESDAY.plus(4) renvoie Thursday.
- Month est une enum avec les douze mois de l'année. Elle contient aussi des méthodes utilitaires. Month.FEBRUARY.length(boolean) renvoie le nombre de jours du mois, selon un paramètre indiquant si l'année est bissextile.

https://docs.oracle.com/javase/tutorial/datetime/iso/enum.html

DateTime

LocalDate

- Les classes les plus utilisées du package seront sans doute LocalDate et LocalDateTime
- D'autres classes existent comme :
  - YearMonth : le mois d'une année
  - MonthDay : le jour d'un mois
  - et Year : une année
- Une LocalDate, ainsi que les autres classes du package java.time s'instancie comme suit (on crée des instances immutables):

```
//Date locale actuelle
LocalDate.now();
//Date locale du 12 janvier 1975
LocalDate.of(1975, 1, 12);
```

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html

LocalDate

- Toutes les classes représentant un moment de java.time implémentent Temporal et TemporalAdjuster, ce qui permet de convertir ces classes facilement.

```
LocalDate parameterDate = LocalDate.of(1975, 1, 12);
//Compare uniquement les années de parameterDate et now()
Year.from(parameterDate).isAfter(Year.now());
```

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html

Formattage et parsing

- On peut formatter un LocalDateTime avec un DateTimeFormatter :

```
// Création d'un formatteur de dateTime
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy à
HH:mm");
System.out.println("Le départ aura lieu le " + LocalDateTime.of(2022, 7, 3, 10,
23).format(dateTimeFormatter));
//affiche : Le départ aura lieu le 03/07/2022 à 10:23
```

- On peut aussi transformer une chaîne de caractères en LocalDate avec la méthode parse()

```
// Parsing d'une chaîne de caractère représentant une date en ISO-8601
LocalDate date = LocalDate.parse("2029-03-28", DateTimeFormatter.ISO_DATE);
```

- D’autres formatteurs sont disponibles et cette syntaxe s’applique aux autres Temporal.

https://docs.oracle.com/javase/tutorial/datetime/iso/index.html

Exercices LocalDate

- Créer un programme qui instancie une LocalDate (avec .of()) et affiche si cette dernière est dans une année bissextile.
- Créer un programme en Java qui prend un paramètre au lancement. Ce paramètre représente une date au format ISO-8601 (AAAA-MM-JJ) . S'il parvient à transformer ce paramètre en LocalDate, il affiche si oui ou non, la date est une année bissextile. Si il ne parvient pas à transformer la LocalDate, il affiche une erreur.
- Faire en sorte qu'il affiche le mois de l'année et le jour de la semaine.
- Bonus : afficher de manière lisible pour un humain la date avec le jour de la semaine.

DateTime

Temporal Adjuster

- TemporalAdjuster est une interface fonctionnelle permettant de convertir un Temporal en un autre. Or :
  - De nombreux TemporalAdjusters sont disponibles via la classe TemporalAdjuster**s**
  - Les objets Temporal ont la méthode with() permettant de passer en paramètre un TemporalAdjuster (et donc de convertir des dates).
  - Attention, on crée à chaque fois de nouvelles instances immutables.
  - Il est facile de faire des opérations sur les dates ainsi.

```
Trouver le dernier jour d'une année
LocalDate lastDayOfYear = LocalDate.of(2000, 16, 1).with(TemporalAdjusters.lastDayOfYear());
```

https://docs.oracle.com/javase/tutorial/datetime/iso/adjusters.html

TemporalQuery

- TemporalQuery est une interface fonctionnelle permettant de vérifier si un Temporal correspond à des critères, ou de récupérer des informations sur le Temporal. Elle peut renvoyer n'importe quel type.
  - De nombreuses TemporalQueries sont disponibles via la classe TemporalQueries
  - Les objets Temporal ont la méthode query() permettant de passer en paramètre une TemporalQuery.

```
Boolean isYearAfter2000 = LocalDate.now().query(t-> t.get(ChronoField.YEAR) > 2000);
```

Exercices TemporalAdjusters

- Trouver et afficher le dernier mardi précédant aujourd'hui.
- Bonus : trouver le premier jour (de la semaine) de la quatrième année après celle actuelle.

DateTime

Period & Duration

- Period mesure un intervalle de temps en jours et Duration mesure un intervalle de temps avec une échelle plus petite (heure, minute, seconde).
- Ces deux classes s'utilisent de la même manière, avec la méthode between prenant deux paramètres :
  - un paramètre de début de période (inclusif).
  - un paramètre de fin de période (exclusif).
- Ces deux classes proposent ensuite des méthodes utilitaires pour la durée.

```java
// Trouver le nombre de jours entre le dernier mercredi et aujourd'hui
LocalDate dernierMercredi = LocalDate.now().with(TemporalAdjusters.previous(DayOfWeek.WEDNESDAY));
Period.between(dernierMercredi, LocalDate.now()).getDays();

// Trouver le nombre de secondes entre deux dateTimes
System.out.println(Duration.between(LocalDateTime.of(2000, 1, 1, 12, 30), LocalDateTime.now()).toSeconds());
```

https://docs.oracle.com/javase/tutorial/datetime/iso/period.html

Conversions : non ISO et fuseaux horaires

- On peut utiliser d'autres calendriers et convertir les dates avec la méthode from :

```
// D'une date du calendrier grégorien à une date japonaise
LocalDateTime occidentalDate = LocalDateTime.of(2013, 01, 20, 19, 30);
JapaneseDate japaneseDate = JapaneseDate.from(occidentalDate);
LocalDate.
```

- Pour les fuseaux horaires, on peut utiliser ZoneId (un identifiant de TimeZone) et ZoneOffset(uniquement le décalage horaire par rapport à UTC/Greenwich), pour instancier une ZonedDateTime.

```
LocalDateTime heureDepart = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
ZoneId zoneParis = ZoneId.of("Europe/Paris");
ZoneId zonePorto = ZoneId.of("Europe/Lisbon");
ZonedDateTime heureDepartZoneParis = ZonedDateTime.of(heureDepart, zoneParis);
ZonedDateTime heureDepartZonePorto = heureDepartZoneParis.withZoneSameInstant(zonePorto);
```

https://docs.oracle.com/javase/tutorial/datetime/iso/timezones.html

TP : affichage de jours

- Afficher tous les mois d’une année et leur durée.
- Afficher tous les lundis d’un mois donné et d’une année donnée.

- Bonus : tenter de faire le calcul suivant :
  - Un boulanger va travailler en 2023 les lundis, mardi et vendredi.
  - Etant superstitieux, il ne travaillera pas les vendredi 13.
  - Calculer le nombre de jours qu’il va travailler.

Ce qu’il faut retenir

- Les Calendar et Date sont à oublier.
- Le package java.time contient de nombreuses classes répondant à de nombreux besoins fonctionnels, autant s’en servir.
- De plus, les LocalDateTime et LocalDate sont maintenant bien intégrés dans les frameworks, et sont facilement convertibles en ISO-8601 (donc en JSON).

Les conseils du formateur

- Lire la documentation du package java.time : https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html peut sembler long, mais cela peut sauver du temps dès lors que l'on fait des opérations sur les dates, les temps ou les durées.
- Il y a des interfaces fonctionnelles dans ce package : utiliser des lambdas peut aider (ou pas) à la clarté du code.

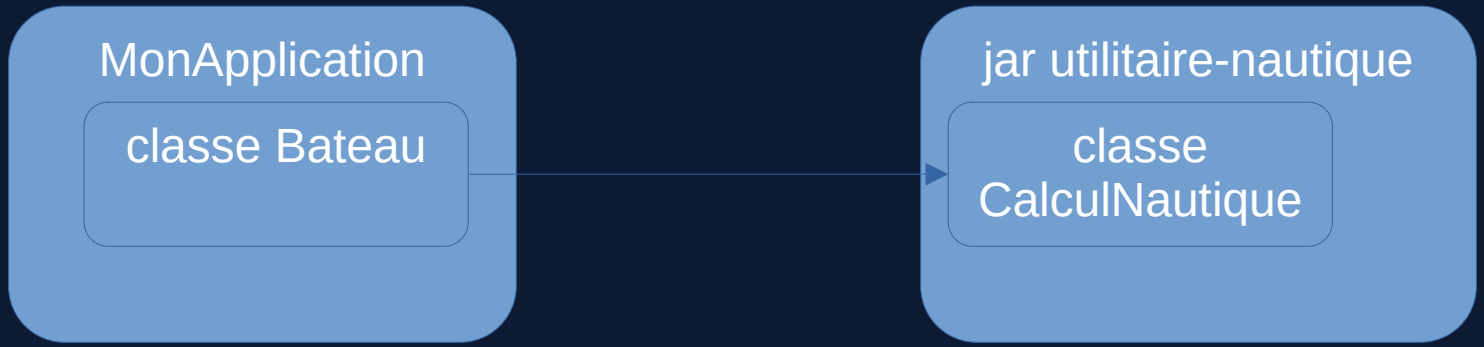
Classloading

Le classpath

- Quelles classes peut-on utiliser dans un projet ?
  - Celles fournies par Java de base : dans le package java.lang .(java.lang.String par exemple)
  - Celles fournies par les extensions de java. Par exemple dans le package javax.xml.
  - Celles ajoutées dans le classpath
- Le classpath permet de préciser au compilateur et à la JVM où se trouvent les classes nécessaires à la compilation et l'exécution d'une application.
- Le classpath est constitué de chemins vers des répertoires et/ou des archives sous la forme de fichiers .jar ou .zip. Chaque élément du classpath peut donc être :
  - Pour des fichiers .class : le répertoire qui contient l'arborescence des sous-répertoires des packages ou les fichiers .class (si ceux-ci sont dans le package par défaut)
  - Pour des fichiers .jar ou .zip : le chemin vers chacun des fichiers
  - Le répertoire à partir duquel on lance javac ou java est dans le classpath

Usage du classpath

- Si notre projet a besoin de classes contenues dans un JAR (un zip contenant des classes). Il faut ajouter à la compilation et à l'exécution ce JAR dans le classpath.




```
javac -cp utilitaire-nautique.jar simulateur.bateau.Bateau.java
java -cp utilitaire-nautique.jar simulateur.bateau.Bateau.java
```

JAR

- Un JAR (Java ARchive) est un fichier zip contenant :
  - des classes compilées (des fichiers .class), dans une arborescence compatible avec leur nom de package.
  - un fichier META-INF/MANIFEST.MF décrivant le jar (licence, informations pour exécuter le jar ...)

Usage du classpath

- On peut ajouter les différentes ressources du classpath en les séparant par un ":" sous Unix ou “;” sous Windows.

```
javac -cp utilitaire-nautique.jar;autre.jar simulateur.bateau.Bateau.java
javac -cp utilitaire-nautique.jar:autre.jar simulateur.bateau.Bateau.java
```

- Si la même classe avec le même nom de package est présente dans plusieurs jars utilisés, celle du premier jar sera utilisée.
- Un JAR peut aussi contenir dans son fichier META-INF/MANIFEST.MF les classpath dont il a besoin (souvent sous la forme d’autres JARs), ainsi que la classe contenant la méthode main. Ainsi on peut déployer ce JAR et le lancer avec un simple : java -jar nom-du-jar

Rôle des ClassLoaders

- Les ClassLoader Java font partie de l’environnement d’exécution de la JVM. Le système d'exécution Java n'a pas besoin de connaître les fichiers et les systèmes de fichiers grâce aux classloaders.
- Les classes Java ne sont pas chargées en mémoire en une seule fois au démarrage de la JVM, mais lorsqu'une application en a besoin directement ou non. À ce moment, le ClassLoader Java est appelé par la JVM et ces ClassLoaders chargent dynamiquement les classes en mémoire.
- On peut charger des classes à la volée, et même générer une classe dans une JVM pour la charger ensuite via un ClassLoader.
- Il y a aussi un risque qu’un programme fasse référence à une classe que le ClassLoader ne peut trouver (et une ClassNotFoundException...).

Avec un IDE et Maven

- L’utilisation du Classpath est incoutournable pour développer et déployer une application utilisant des composants externes ...
- ... mais ceci est tellement fastidieux que presque plus aucun développeur ne le fait à la main.
- Les IDEs et des outils de build comme Maven permettent de définir à haut niveau des dépendances. L’outil de build télécharge ensuite ces dépendances, les ajoute dans le classpath à l’exécution, et même à la livraison.
- Néanmoins, en cas de souci, il est absolument nécessaire de comprendre comment le classpath fonctionne.

Ce qu’il faut retenir

- Un programme Java a besoin de nombreuses classes pour fonctionner.
- Il est recommandé de réutiliser les classes déjà existantes, développées par des collègues ou d’autres entreprises, via des JARs.
- Il faut définir au lancement de la JVM les JARs à utiliser, c’est ce à quoi sert le classpath : le classpath est une liste de JARs (ou de répertoires) qui contiennent les classes nécessaires à la compilation, ou l’exécution du projet.
- Aujourd’hui, les développeurs modifient rarement ce classpath à la main, mais il faut tout de même avoir conscience de ce mécanisme, en cas de problème.

Les conseils du formateur

- Si une NoClassDefFoundError (ou une ClassNotFoundException) survient, c'est généralement qu'une classe dont vous avez besoin n'est pas dans le classpath (le jar manque, ou ce n'est pas le bon, ou ce n'est pas la bonne version...)
- Les applications Web Java gèrent le classpath différemment des jars, il vaut mieux s'y intéresser quand on développe une application Java Web. Consulter la documentation du serveur Web (ou du conteneur de Servlets).
- Utilisez Maven (ou un autre outil de build).

https://docs.oracle.com/javase/tutorial/essential/environment/paths.html

https://maven.apache.org/

Modules

Problèmes de modularisation en Java

La modularisation en Java se fait principalement via des JARs (Java ARchives). Ces fichiers contiennent des classes et un descripteur (le fichier META-INF/MANIFEST.MF)

Limites du JAR

Les JARs sont utiles (et utilisés). Néanmoins, ils ont été conçus pour répondre à des problèmes techniques de bas niveau, et ont montré des limites au fur et à mesure que les applications Java grandissaient :

- le JAR ne gère pas les versions : d'où l'utilisation de Maven (ou autres ...) pour ce faire.
- le JAR ne permet pas de déclarer ses dépendances : Maven pallie ceci.
- Un fichier JAR est un simple conteneur : toutes les classes public d'un JAR sont exposées au 'monde entier' (en réalité, à tout programme incluant le JAR dans son classpath).

Limites du Classloader

Le mécanisme de classloading charge les classes à la demande, en scannant tous les JARs. Par conséquence :

- il est impossible pour la JVM de déterminer si tous les JAR requis sont présents dans le classpath. Ainsi si une classe n'est pas trouvée, une exception de type NoClassDefFoundError est levée à sa première utilisation
- si la classe demandée est dans plusieurs JARs, ce qui est souvent dû à deux versions d'un même JAR présent dans le classpath, le classloader prendra le premier JAR qui contient la bonne classe. Or, ceci :
  - peut dépendre de l'OS, et amener un comportement différent selon les systèmes. Ce genre de problèmes est extrêmement difficile à débugger.
  - peut amener des comportements indésirables, puisque certaines classes seront 'prises' dans un JAR et d'autres dans un autre.

Limites du Classloader (2)

Le classpath est paramétrable de manière indépendante lors de :

- la compilation,
- l'exécution.

Si les paramétrages diffèrent, une compilation (voire un packaging) de l'application peut se faire correctement, mais pas son exécution.

Finalement, le parcours séquentiel des jars du classpath pour trouver une classe, voire le parcours des classes pour trouver celles qui sont annotées, peut prendre du temps pour une application ayant besoin d'un grand nombre de classes.

Rôles des modules

- Les modules ont été ajoutés dans Java9.
- Ils permettent :
  - d’encapsuler les JARs en décrivant les packages auxquelles les autres modules ont accès : les packages exportés. De fait, ceci empêche tout autre module d’accéder à un package non exporté.
  - de déclarer des services.
  - de déclarer les modules dont a besoin un autre module.
  - d’effectuer des vérifications à la compilation, et/ou à l’exécution, pour s’assurer que les modules requis sont bien présents.

Déclaration

Un module doit avoir :

- ses sources dans son répertoire
- un fichier module-info.java à la racine de l’arborescence des sources
- Les données d’un module se trouvent dans un fichier, à la racine d’un JAR.

Exemple de déclaration dans un projet Maven de type JAR

- Les données d’un module se trouvent dans un fichier, à la racine d’un JAR.
- Étant donné que module-info.java se situe à la racine du JAR, il n’appartient au package par défaut (void).
- Il est déconseillé de coder des classes avec le package par défaut : dans un projet correct, module-info.java devrait être le seul fichier dans ce package. Dans un projet Maven, ce fichier se situera dans src/main/java.

Exemple de déclaration dans un projet multi-modules

- Le code source de Java est organisé ainsi :

```
src
|--com.corp.modulea
|  |--module-info.java
|  |--com
|     |--corp
|        |--modulea
|           |--ClasseA.java
|--com.corp.moduleb
|  |--module-info.java
|  |--com
|     |--corp
|        |--moduleb
|           |--ClasseB.java
```

Contenu du fichier module-info.java

Tout comme META-INF/MANIFEST.MF, le fichier module-info.java contient des méta-données. Les méta-données (données profondes), servent à définir d'autres données. Les méta-données de module-info.java définissent :

- le nom du module, équivalent à son identifiant.
- la liste des dépendances requises par ce module.
- les packages exposés par ce module, pour être utilisés 'normalement' par les classes d'autres modules.
- les services exposés et/ou consommés.
- les packages exposés pour être utilisés par réflexion.
- des lignes d'import, si nécessaire.

module

Le fichier module-info.java commence par la directive ‘module’, suvi du nom du module, et d’une paire d’accolades :

```
module nom-module{
}
```

Il n’y a qu’une seule directive module par fichier.

Nom du module

Le nom du module n’est pas une simple chaîne de caractères :

- Il est constitué de chaînes de caractères séparées par des points.
- Il ressemble à un nom de package et obéit aux mêmes contraintes de nommage.
- Il devrait être unique au sein d’une équipe, d’une entreprise, voire mondialement unique.
- Pour obéir à la dernière règle, nommer ces modules comme les projets Maven semble être une bonne idée. La notation d’URL inversée est alors utilisée : com.maboite.monprojet.monmodule

Requires

- ‘requires’ est une directive se trouvant dans module. La syntaxe de requires est :

```
module modulea{
    requires moduleb;
}
```

- Le fichier ci-dessus décrit que modulea a besoin de moduleb. modulea peut alors accéder à tous les types (classes,interfaces ...) publics des packages **exportés** par moduleb.
- un module peut contenir de 0 à n directive requires.
- Toute dépendance doit être déclarée explicitement sauf :
  - la dépendance au module de base java.base . Il est implicitement requis car il contient des types requis par tout code Java. Ceci est équivalent à l’import automatique des classes du package java.lang.
  - les dépendances déclarées transitives dans la description d'une autre dépendance.

Exports

- ‘exports’ est une directive se trouvant dans module. La syntaxe de exports est :

```
module moduleb{
    exports nom.package;
}
```

- Le fichier ci-dessus décrit que moduleb exporte le package nom.package.
- Un module peut contenir 0 à n directives exports.

Pour qu’un autre module que moduleb accède à ClasseB, une classe de moduleb, les conditions suivantes doivent être respectées :

- ClasseB doit être public
- Le package de ClasseB doit être exporté en utilisant le mot clé exports
- Le module qui a besoin de ClasseB doit déclarer moduleb comme une dépendance en utilisant le mot clé requires

- Tout manquement à un de ces règles amène une erreur (à la compilation ou à l’exécution).

Exports

- Les sous-packages d'un package exportés ne sont pas accessibles : il faut exporter explicitement tous les sous-packages concernés un par un.

Rétrocompatibilité

- Pour des raisons de rétro-compatibilité, des règles spéciales s'appliquent au cas où toutes les dépendances ne sont pas 'modularisées' :
  - Si un jar A dépend d'un jar B et que le jar A ne contient pas de module-info.java, les règles de module ne s'appliquent pas (il n'y a aucune encapsulation supplémentaire), même si le jar B contient un module-info.java. Ceci protège les développeurs qui ne peuvent ou ne veulent utiliser les modules de bugs, même s'ils utilisent des jar qui sont 'modularisés'.
  - En revanche, si un jar A dépend d'un jar B, et que le jar A contient un module-info.java, et non le jar B, les règles de module s'appliquent. Cela signifie que le jar A doit définir qu'il dépend du module B, alors que ce module n'existe pas ...

Module automatique

Pour résoudre le cas précédent, la notion de module automatique a été ajoutée. Un module peut utiliser des informations d'un JAR non modularisé pour référencer celui-ci :

- Soit le fichier META-INF/MANIFEST.MF du JAR contient une entrée : Automatic-Module-Name, le JAR propose un module automatique avec ce nom.
- Sinon, le système de module déduit un nom de module du JAR. Pour ce faire, le numéro de version (ex -0.4.2) du JAR est ignoré, et tout caractère qui n'est ni un chiffre ni une lettre est transformé en points.

  En conséquence, le JAR : my-super-jar-2.0.12.jar a le nom de module automatique : my.super.jar .

  Le jar byte-buddy-1.0.2.jar a le nom de module automatique : byte.buddy. byte étant un mot réservé Java, ce module est interdit. Ce Jar n'est pas importable avec uniquement le nom de fichier.

Exercice

- Le projet Maven contient les sous-projets :
  - com.bigcorp.project.graphics
  - com.bigcorp.project.data-repository
- La classe com.bigcorp.project.graphics.windows.StartWindow utilise com.bigcorp.project.data.repository.UserRepository
- Lancer le main de StartWindow et s'assurer que l'application s'exécute correctement.
- Créer le fichier module-info.java pour le projet data-repository et n'y mettre que la directive module. L'application fonctionne-t-elle ?
- Supprimer le précédent fichier. Créer le fichier module-info.java pour le projet graphics et n'y mettre que la directive module. L'application fonctionne-t-elle ?
- Créer les deux fichiers module-info.java pour que l'application fonctionne correctement.
- Bonus : tenter de faire fonctionner l'application avec uniquement un fichier module-info.java dans le projet graphics

Provides with

- ‘provides ... with ’ est une directive se trouvant dans module. La syntaxe est :

```
module moduleb{
    provides MonService with MonServiceImpl;
}
```

- Le module ci-dessus décrit que moduleb est un fournisseur de service.
- Un module peut contenir 0 à n directives provides.
- provides ... with spécifie que le module implémente un ou plusieurs services. Le module est un fournisseur de services.
- Après ‘provides’ est référencée une interface ou classe abstraite.
- Après ‘with’ est référencé le nom de la classe concrète qui implémente le service ou hérite de la classe abstraite.
- Il est alors possible d’accéder au service via les mécanismes de service de Java. (par exemple avec un ServiceLoader). Il faut alors utiliser la directive uses dans le module consommant le service.

Uses

- uses permet de spécifier une interface utilisée par le module.
- Un autre module (ou plusieurs) est censé fournir une (ou plusieurs) implémentations de cette interface, grâce à la directive ‘provides with’.

```
module autre.module {

    uses MonService;

}
```

- Les règles des modules s’appliquent (il faut que autre.module requiert le module qui contient le service , et celui qui contient l’implémentation).
- Il est possible de récupérer les implémentations dans le code, par exemple comme suit :

```
MonService monService = ServiceLoader
                  .load(monService.class)
                  .findFirst()
                  .orElseThrow();
```

Exercice : services (bonus)

- Le projet Maven contient le sous-projet :
  - com.bigcorp.project.data-contract
- Celui-ci contient une interface AddressService, implémentée dans data-repository par AddressServiceImpl.
- Rendre data-contract modularisé et utiliser provides with dans le fichier module-info.java de data-repository, et uses dans le module-info.java de graphics.
- Décommenter les deux dernières instructions de StartWindow pour tester.

Open(s)

- Avant Java 9, il était possible d'utiliser la réflexion pour connaître tous les types d'un package, et lire ou écrire dans tous les attributs de ce type, et exécuter toutes ses méthodes.
- Il n'est pas exagéré de dire qu'aucune classe n'était complètement encapsulée.
- Avec le système de module, les restrictions d'export s'appliquent aussi à la réflexion.
- La permission d'utiliser la réflexion sur un package se fait avec les directives open et opens.

Open(s)

- La directive opens :

```
module moduleb{
    opens package;
}
```

- indique que les types publics d'un package (et leurs types internes) sont accessibles aux autres modules à l'exécution, et que les types du package et leurs membres sont accessibles par la réflexion.
- un module peut avoir de 0 à n directives opens.

Open(s)

- La directive opens to :

```
module moduleb{
    opens package to modulea;
}
```

- indique que les types publics d'un package (et leurs types internes) sont accessibles uniquement au modulea à l'exécution, et que les types du package et leurs membres sont accessibles par la réflexion.
- Il est possible de séparer les modules par des virgules :

```
module moduleb{
    opens package to modulea, modulec;
}
```

Open(s)

- Il est possible d’ouvrir un module :

```
open module moduleb{

}
```

- pour indiquer que tous les packages d’un module doivent être accessibles à l’exécution, et via la réflexion.

Mots réservés et syntaxe

- Module, provides, with, requires, exports, opens, ... n’ont pas été ajoutés à la liste des mots clés réservés du langage Java. Ce sont des « mots clés contextuels ». Ils sont réservés dans le fichier module-info.java, mais pas dans le code Java.
- Aucun ordre n’est obligatoire pour les directives, mais il est recommandé de les grouper et de garder le même ordre dans tous les fichiers d’un service, d’une entreprise ...
- Les commentaires et JavaDoc sont permis dans les descripteurs de module.

Graphe de dépendances : problème

- Un module A peut dépendre d'un module B pour utiliser ses classes, et ces classes peuvent elles-mêmes utilisées dans les méthodes que le module A exporte.

```
// Cette classe est exportée
public class ClasseADuJarA{

    public ClasseBDuJarB execute(){
    ....
    }

}
```

- Ceci fonctionne sans problème. Par contre, les modules qui dépendent du module A vont devoir indiquer qu'ils dépendent du module A (pour exécuter execute()) et du module B pour utiliser ClasseBDuJarB. Si le module A dépend de 30 autres modules, tous les modules dépendant de A vont devoir indiquer qu'ils dépendent aussi de ces 30 autres modules (!) .

Graphe de dépendances : solution

- Il est possible d'utiliser le mot-clé transitive dans un requires :

```
module a{
    requires com.bigcorp.a;
    requires com.bigcorp.b;

    requires transitive com.bigcorp.c;
}
```

- Ceci permet de notifier que tout module qui dépend du module a dépend aussi automatiquement de com.bigcorp.c .

Exercice requires transitive

- Faire en sorte que data-repository dépende transitivement de data-contract.
- Modifier le module-info.java de graphics pour qu'il ne dépende plus que de data-repository
- S'assurer que le code compile

Modulariser ses projets

- Les dépendances entre modules amènent à trier ces modules de haut en bas.
- En bas se trouvent les modules qui ne dépendent d’aucun autre module.
- En haut se trouvent les modules dont personne ne dépend.
- Le graphe de module est ensuite construit.

Lancement via les modules

- Pour information, pour ajouter des modules au classpath (ou plutôt au module-path), il faut ajouter l'option --module-path aux commandes javac, jar, et java .
- L'option --module-path est très proche de --cp

Packaging et JAR

Ci-dessous un exemple d'utilisation de --module-path en ligne de commande pour compiler, créer un jar et exécuter un module Java :

```
$ javac
    --module-path deps
    -d target/classes
    src/main/java/module-info.java
    src/main/java/com/example/Main.java
$ jar --create
    --file target/hello-modules.jar
    target/classes/module-info.class
    target/classes/com/example/Main.class
$ java
    --module-path target/hello-modules.jar:deps
    --module com.example/com.example.Main
```

TP

Étoffer les modules data-repository, et graphics :

- Raccorder le module business, prenant place ‘entre’ data-repository et graphics.
  - business dépend de data-repository et graphics dépend de business.
  - Créer dans business des interfaces avec des méthodes (autant que vous voulez). Implémenter les méthodes correspondantes dans data-repository. Utiliser les interfaces de business dans graphics.
- Créer un projet dépendant du module business, mais qui n’est pas modularisé.

Ce qu’il faut retenir

Les modules sont une nouveauté de Java9. Ce sont des JARs (généralement), qui contiennent un fichier module-info.java, il est possible de :

- Renforcer l’encapsulation, en ne mettant à disposition d’autres modules qu’un petit nombre de packages.
- Mieux fiabiliser la configuration, en permettant de contrôler à la compilation et à l’exécution la présence des bonnes classes.
- Découper un seul projet en modules (c’est le cas de la JVM).
- De par son existence, le fichier module-info.java est le ‘document d’architecture’ présentant ce à quoi sert un module et ce dont il a besoin. Il peut de plus être commenté pour ajouter de la documentation.

Freins à la modularisation

La modularisation est souvent présentée comme une ‘ceinture de sécurité’ et non comme une ‘voiture de course’. Elle n’apporte que des sécurités sur un ou plusieurs programmes. Néanmoins, comme toute évolution, elle peut aussi apporter des problèmes.

De plus, la modularisation de projets d’une entreprise se fait plus naturellement du bas vers le haut. Or, les modules de plus bas niveau sont généralement des JARs provenant des référentiels Maven, qui ne sont pas encore tous modularisés.

Les projets sont déjà tous ‘modularisés’ en entreprise, parce qu’ils utilisent un outil de build comme Maven.

https://www.youtube.com/watch?v=UqnwQp1uHuY

Les conseils du formateur

- Faut-il modulariser ? L'avis qui suit est très subjectif :
  - Un JAR utilitaire partagé dans l'entreprise déjà existant: pourquoi pas. A condition de se préparer à tester. Le risque de régressions plus bas si les JARs dont lui même dépend sont modularisés. Les gains que la meilleure encapsulation apporte pourraient être intéressants par rapport au coût du refactoring.
  - Le même JAR, mais nouveau : oui.
  - Une application dont personne ne va dépendre, et qui dépend de JARs non modularisés : pas vraiment (à moins que vous n'ayiez que ça à faire).
  - La même application, mais qui dépend de JARs modularisés, pourquoi pas si elle est déjà existante. Ce sera plus intéressant si elle est nouvelle.

JShell

Introduction

JShell (Java Shell Tool) est un outil interactif en ligne de commande, introduit dans la JDK 9. Il permet d’exécuter rapidement du code Java.

JShell est un outil REPL (Read-Evaluate-Print Loop). Il évalue du code et montre ses résultats immédiatement.

https://docs.oracle.com/en/java/javase/20/docs/specs/man/jshell.html

Démarrage et arrêt

Si un répertoire bin de Java est dans le PATH du système, lancer jshell (ou jshell.exe sur un système d'exploitation Windows) :

```
jshell
```

La ligne de commande affiche alors jshell> , ce qui indique que jshell interprète le code Java qui va suivre.

Pour quitter jshell, utiliser la commande : /exit

```
jshell >/exit
```

Possibilités

Il est possible avec Jshell de :

```
jshell> 1+1 //Exécuter des commandes
$1 ==> 2

jshell> 1+1; //Exécuter des commandes
$2 ==> 2

jshell> $4+$4; //Réutiliser des variables créées par jshell
$2 ==> 4

jshell> int i = 4 * 2; //Définir des variables
i ==> 8
```

Méthodes

Il est aussi possible avec Jshell de créer/modifier des méthodes :

```
jshell> void coucou(int a, int b){
   ...>     System.out.println("a vaut : "  + a + " , et b vaut : " + b );
   ...> }//Définir des méthodes
|  created method coucou(int,int)

jshell> long coucou(int a, int b){
   ...>     System.out.println("a vaut : "  + a + " , et b vaut : " + b );
   ...>     return a + b;
   ...> }//Redéfinir des méthodes
|  created method coucou(int,int)
```

Redéfinition de types

```
jshell> int x = 2; // Définition de x
x ==> 2

jshell> x = "4"; // Changement de type de x sans indiquer le nouveau type
|  Error:
|  incompatible types: java.lang.String cannot be converted to int
|  x = "4";
|      ^-^

jshell> String x = "4"; // Changement du type de x
x ==> "4"
```

Commandes

jshell propose des commandes :

- /exit quitte jshell
- /list affiche la liste des commandes exécutées de cette session
- /vars affiche la liste des variables
- /methods affiche la liste des méthodes

Auto-complétion

L’appui sur <Tab> permet d’auto-compléter la saisie.

Cela fonctionne aussi pour les commandes

Exercice

Coder la méthode factorielle(n) de manière récursive, et la lancer avec différents arguments, via JShell.

Ce qu’il faut retenir

Jshell permet d’écrire du code Java et de l’exécuter sur un terminal et sans IDE.

Des raccourcis et facilités de code ont été ajoutés pour rendre l’expérience moins pénible.

Les conseils du formateur

- Jshell peut être utile ...
- ... peut être pour des opérations rapides qu'on ne peut ou ne veut pas faire en shell (traitement sur des fichiers, des répertoires, des flux réseaux ...)

Optional

Introduction

- Optional est une classe de type “Conteneur”. Il peut contenir une valeur, ou null . S’il contient une valeur, isPresent() == true et get() renvoie cette valeur.
- Optional a été ajoutée depuis Java 8.

https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

isPresent et get

- Si isPresent()==true, optional contient une valeur non nulle.
- get() renvoie la valeur de l'optional si elle existe, ou lance une NoSuchElementException dans le cas contraire.

```
//monObjet n'est pas contenu dans un Optional
if(monObjet != null){
    monObjet.appelleMethode();
}

//monObjet est contenu dans un Optional
if(monOptional.isPresent()){
    monOptional.get().appelleMethode();
}
```

Optional

orElse

- orElse(other) renvoie l'objet contenu s'il est non null, ou other si ce dernier est null.

```
//monObjet n'est pas contenu dans un Optional
if(monObjet != null){
    monObjet.appelleMethode();
}else{
    monObjetParDefaut.appelleMethode();
}

//monObjet est contenu dans un Optional
monOptional.orElse(monObjetParDefaut).appelleMethode();
```

Optional

ifPresent

- ifPresent(Consumer c) exécute Consumer (une interface fonctionnelle) si l'objet contenu est non null. Sinon, il ne fait rien.

```
//monObjet n'est pas contenu dans un Optional
if(monObjet != null){
    maMethode(monObjet);
}

//monObjet est contenu dans un Optional
monOptional.ifPresent(maLambda);
```

orElseGet

- orElseGet(Supplier s) retourne l'objet contenu s'il est non null. Sinon, la méthode appelle s pour renvoyer un objet.

```
//monObjet n'est pas contenu dans un Optional
if(monObjet == null){
    monObjet = creeObjet();
}

//monObjet est contenu dans un Optional
monOptional.orElseGet(maLambda);
```

Optional

of

- of(Object) est une méthode statique qui permet de créer un Optional à partir d’un objet non null. ofNullable(Object) est l’équivalent qui permet de créer un Optional à partir d’un objet null.

Streams

Les streams utilisent beaucoup Optional, pour renvoyer une valeur non nulle, et permettre à d'autres méthodes de traiter cette valeur fluidement :

```
//La méthode max renvoie un Optional
Child older = children.stream().min(Comparator.comparing(c ->
c.getAge())).orElse(defaultChild);
```

Exercice

- Créer une méthode qui prend deux paramètres, et renvoie un Optional<Object>
- Selon les paramètres, l'optional contiendra ou non un objet.
- Dans le client, avec ce que renvoie la méthode, utiliser isPresent, get, orElse, ifPresent, orElseGet . Dans ces dernier cas, utiliser une lambda.

Ce qu’il faut retenir

- Optional est une classe qui permet de traiter les objets null avec des méthodes, plutôt qu’avec des comparaisons != null et == null.
- Il est utilisé par Streams et d’autres bibliothèques, il est nécessaire de connaître cette classe, au moins en tant qu’utilisateur d’instances de Optional.

API Process

Introduction

L’API Process a été améliorée en Java9 . Elle permet de ;

- Lancer une commande (et donc un process).
- Lister et manipuler les processus en cours.
- Nettoyer des processus.
- Rediriger leur sortie.
- Tester des exécutions de commandes.
- Superviser des commandes, leur temps de vie, les redémarrer....

Au centre de cette API se trouvent les classes : Process, ProcessHandle et ProcessBuilder

https://docs.oracle.com/javase/9/core/process-api1.htm

ProcessHandle

ProcessHandle est une interface qui identifie et donne le contrôle des processus natifs. Elle dispose de méthodes utilisateurs pour récupérer des processus :

```
//Itère sur tous les processus et affiche leur pid
for (ProcessHandle p : ProcessHandle.allProcesses().toList()) {
     System.out.println(p.pid());
}
```

ProcessHandle.info() renvoie une instance de ProcessHandle.Info, qui permet de récupérer les arguments, la commande, la ligne de commande, l'utilisateur, la date de démarrage du processus. L'utilisation totale du CPU est aussi affichée. Toutes ces informations peuvent être disponibles ou non, selon l'OS.

https://docs.oracle.com/javase/9/docs/api/java/lang/ProcessHandle.html
https://docs.oracle.com/javase/9/core/process-api1.htm

ProcessHandle

ProcessHandle.current() permet aussi de récupérer des informations sur le processus en cours.

Il est aussi possible de demander la destruction d'un processus (ceci peut réussir ou échouer selon l'OS et les droits des processus).

ProcessBuilder

ProcessBuilder crée des processus pour l'OS.

Chaque ProcessBuilder est créé avec un ensemble d'informations (commandes, options, flux vers lesquels les sorties. La méthode start() crée alors un processus. Des appels répétés à start() créent de nouveaux processus.

```java
ProcessBuilder processBuilder = new ProcessBuilder("java", "--version");
processBuilder.redirectOutput(Redirect.appendTo(new File("java-version.txt")));
//lance le process
Process p = processBuilder.start();
//L'arrête
p.destroy();
```

https://docs.oracle.com/javase/9/docs/api/java/lang/ProcessBuilder.html

Exercice

- Créer une nouvelle classe.
- Sa méthode main permet de :
  - trouver le premier processus s’appelant notepad (ou tout autre éditeur de texte que vous utilisez) (sauvegardez votre texte d’abord !!) .
  - afficher toutes les informations du processus.
  - puis le terminer.

Ce qu’il faut retenir

L’API Process permet, grâce aux classes :

- Process
- ProcessBuilder et
- ProcessHandle

De manipuler :

- lister
- démarrer
- arrêter

des processus assez facilement.

Les conseils du formateur

- L’API Process peut être utile pour gérer les processus, mais attention aux spécificités du système d’exploitation quand on va manipuler SES processus : Java se veut transparent là-dessus, mais au final, les systèmes peuvent différer.

JDK 9 : autres nouveautés

Améliorations des streams Java 8

Les Streams ont des nouvelles méthodes :

- takeWhile
- dropWhile
- iterate
- ofNullable : crée un Stream à partir d’un objet, null ou non.

Stream.takeWhile()

takeWhile(Predicate) renvoie un nouveau Stream composé des éléments qui passent le prédicat, jusqu'à ce que le premier échoue au test du prédicat.

contrairement à filter(Predicate), des éléments du stream qui passaient le test du prédicat, mais qui se trouvent après le premier élément à avoir échoué, ne seront pas présents dans le nouveau stream.

```
Stream.of("a", "b", "c", "", "e")
          .takeWhile(s -> !s.isEmpty())
          .forEach(System.out::print);
```

Stream.dropWhile()

dropWhile(Predicate) renvoie un nouveau Stream composé des éléments, qui suivent le premier élément à ne pas passer le test de prédicat.

```
Stream.of("a", "b", "c", "", "e")
          .dropWhile(s -> !s.isEmpty())
          .forEach(System.out::print);
```

Stream.iterate()

- Stream.iterate() propose de quoi construire un Stream avec une graine(seed) et une opération qui calcule la prochaine valeur du stream à partir de la précédente.

  Stream.iterate(T seed, UnaryOperator<T> next)

- Il est possible d'utiliser une surcharge de la méthode iterate, qui prend en deuxième argument une condition d'arrêt. La méthode Stream.iterate() permet de créer un stream d'objets.

  La méthode a comme signature :

  Stream.iterate(T seed, Predicate<T> hasNext, UnaryOperator<T> next)

```
// Crée un stream avec les nombres pairs de 0 à 20
Stream.iterate(0, i -> i <= 20, i -> i + 2).forEach(System.out::println);
```

Exercice : utilisation de takeWhile et iterate

- Utiliser takeWhile et iterate (version avec deux arguments) de Stream pour afficher tous les multiples de 3 de 0 à 40.
- Bonus : Combinez dropWhile et takeWhile pour n'avoir que les nombres dans une certaine fourchette (de 20 à 40 par exemple).

HTTP2

Avant la JDK 9, HttpURLConnection était utilisée pour communiquer avec HTTP. Néanmoins, cette classe avait quelques problèmes :

- HttpURLConnection supporte le protocole HTTP/1.1 mais pas HTTP/2.0 . Avec HTTP/1.1 il n'est possible d'envoyer qu'une requête par connexion TCP. HTTP/2.0 permet d'envoyer plusieurs requêtes par connexion, ce qui peut améliorer les performances d'une application.
- HttpURLConnection ne fonctionne qu'en mode synchrone. Le processus de fonctionnement est le suivant : une requête HTTP est envoyée par la classe. Quand la réponse est retournée, la réponse peut être traitée. Ceci peut aussi être amélioré au niveau des performances.

HTTP2

java.net.http.HttpClient, HttpRequest et HttpResponse sont les nouvelles classes nécessaires pour traiter les requêtes HTTP2. Attention, ces classes nécessitent le module java.net.http :

```
// Creation du HTTP Client
HttpClient httpClient = HttpClient.newHttpClient();

// Création de l'HttpRequest GET www.google.fr
HttpRequest httpRequest = HttpRequest
            .newBuilder()
            .uri(new URI("https://www.google.fr"))
            .GET()
            .build();

// Travail en mode asynchrone avec un CompletableFuture.
// La requête est émise, mais le thread peut continuer à travailler
CompletableFuture<String> cf = httpClient.sendAsync(
                           httpRequest,

HttpResponse.BodyHandlers.ofString()).thenApply(HttpResponse::body);

// On revient en mode synchrone (ici le thread peut être bloqué)
System.out.println(cf.get());
```

Ajouts aux collections

Il est possible d'appeler Set.of(....) et Map.of(...) pour créer des Set et Map immutables :

```
Set.of("arthur","fredegonde");
[arthur, fredegonde]

Map.of(1l,"Robert",4l,"Renault");
{1=Robert, 4=Renault}
```

ReactiveStreams

Un Stream reactif a besoin de traiter des événements :

- Ces événements sont envoyés au Publisher.
- Des Subscriber peuvent souscrire au Publisher, via Publisher.subscribe(Subscriber).
- Ces Subscriber peuvent alors traiter les messages. Pour ce faire, le développeur a correctement implémenté leurs méthodes : onSubscribe(), onNext(), onError() et onComplete().
- Un Processor peut aussi être implémenté, pour joueur le rôle d'un Publisher et Subscriber.
- Tout ceci permet de créer un flux de données, ou les traitements peuvent être finement définis et parallélisés.

JDK 9 : autres nouveautés

Méthodes privées d'interfaces

- Il est dorénavant possible de créer des méthodes privées dans une interface.

```
interface Volant{



    // la méthode privée partagée
    private void vole() {
        System.out.println("Bonjour de la méthode privée");
    }
}
```

CompletableFuture

- CompletableFuture représente une classe qui :
  - implémente Future, et donc peut traiter des tâches de manière asynchrone
  - implémente CompletionStage : un ‘étage’ de traitement.
- Ces étages peuvent être composés différemment pour faire des traitements :
  - en parallèle
  - chaînés
  - qui se combinent
- CompletableFuture fonctionne bien avec des Executors.

CompletableFuture

- Voir les exemples dans la classe : CompletableFutureExample

JDK 10

Fonctionnalités en preview

- Une fonctionnalité en preview est une fonctionnalité dont la conception, la spécification, et l'implémentation sont complètes. Mais la fonctionnalité n'est pas permanente : elle peut évoluer ou même ne pas exister dans des versions futures.
- Pour utiliser ces fonctionnalités, il faut activer le paramètre enable-preview à la compilation et à l'exécution.

https://docs.oracle.com/en/java/javase/20/language/preview-language-and-vm-features.html

JDK 10

Inférence de type statique (var)

- Java 10 propose var, qui permet d'inférer (déduire) le type d'une variable. La déduction se fait à la compilation. Les lambdas et l'opérateur<> permettaient déjà au compilateur d'inférer certains types.
- Dorénavant, le mot clé var permet aussi d'inférer le type des variables locales.
- Ce qu'il est possible de faire avec var :

```
// L'inférence de type fonctionne avec les types primitifs, les constructeurs, les appels
// de méthode
var compteur = 5;
var localInference = new LocalInference();
var monSet = Set.of(1,2);
System.out.println("Le résultat de l'addition vaut " + 5 + compteur);

// L'inférence de type fonctionne avec classe fille -> classe mère
var monMetal = new Metal();
monMetal = new Or();

var monArgent = new Argent();
```

JDK 10

var

- Il est des choses impossibles à faire avec var :
  - Il n'est pas possible d'initialiser une variable, en typage automatique, après sa déclaration : les deux étapes (déclaration et initialisation) doivent être faites en même temps.
  - Il n'est pas possible d'utiliser le mot clé var pour déduire automatiquement le type de retour d'une fonction.

JDK 10

var : intérêt

- var est là pour améliorer la visibilité, pas pour rendre les lignes de code moins compréhensibles.
- L’utiliser, par exemple, dans des boucles for, où le type ne sert à rien peut être une bonne idée

var : intérêt

- A contrario, passer tous les types en var peut plus gêner le développeur (qui passe plus de temps à lire du code qu'à en taper), que l'aider.

var : Exercices

- Créer une classe avec une méthode main.
- Cette méthode stocke des variables de type var à partir de :
  - types primitifs
  - objets construits
  - appels d’autres méthodes
- Créer aussi une collection et itérer dessus en utilisant var

Autres améliorations de la JDK 10

- Tout est là : https://www.azul.com/blog/109-new-features-in-jdk-10/

JDK 11

JDK 11

Inférences de type pour les lambdas

- Il est possible d’utiliser var dans les lambdas, mais avec la limite suivante :
  - Soit tous les arguments sont typés avec var, soit aucun ne doit l’être

Lancement d’application sans compilation

- Il est possible de lancer un fichier source Java sans le compiler. Le fichier source est compilé en mémoire, puis exécuté par la JVM, sans créer de fichier .class .
- Attention ! Seule une application tenant sur un seul fichier source peut être exécuté ainsi.
- Il est bien sûr possible de contourner cette limitation en écrivant de nombreuses classes dans le fichier source, mais ceci contredit des bonnes pratiques de développement orienté objet.
- La première classe du fichier source est celle pour laquelle la méthode main() sera exécutée : l’ordre des classes est important.

Lancement d'application sans compilation : exemple

- Le programme suivant :

```
package com.bigcorp.journal.main.javamain;

public class SimpleMain {

	public static void main(String[] args) {
		System.out.println("Lancement");
		for (String arg : args) {
			System.out.println("Avec l'argument " + arg);
		}
	}

}
```

- peut être lancé ainsi (en étant positionné dans le répertoire où se trouve le fichier source) :

```
java SimpleMain.java
```

Lancement d’application sans compilation : arguments

- Les arguments sont autorisés, et suivent le nom du fichier source :

```
java SimpleMain.java arg1 arg2
```

- Ces arguments seront transmis à la méthode main, comme à l’accoutumée.

JDK 11

Ajouts aux chaînes de caractère

- String.lines() crée un Stream de String, chaque élément du Stream représentant une ligne :

```
String multilineString = "Voilà mon conseil : \n \n découper ses phrases \n en lignes.";

// Crée un stream pour chaque ligne de multilineString
multilineString.lines().forEach(System.out::println);
```

- La méthode String.strip() permet d'enlever tout caractère 'blanc' du début et de la fin d'une String. Ceci diffère de String.trim() qui enlevait tout 'espace'

Ajouts aux chaînes de caractère

- A côté de strip() se trouvent :
  - stripLeading() : qui supprime les caractères blancs de début de chaîne
  - stripTrailing() : qui fait de même pour les caractères blancs de fin de chaîne
  - isBlank() permet de savoir si une chaîne est vide ou ne contient que des caractères blancs :

https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#isWhitespace(int)

Ajouts aux chaînes de caractère

- String.repeat(n) permet de renvoyer la chaîne de caractères d'origine répétée n fois
- StringBuilder et StringBuffer implémentent Comparable et peuvent être utilisés dans des collections triées.

Exercice : chaîne de caractères

- Créer une chaîne de caractères assez longue, contenant des caractères blancs, des retours chariot ...
- La dupliquer 3 fois.
- Utiliser les nouvelles méthodes de String pour en extraire toutes les lignes, sans caractère blanc au début (ou à la fin).

Classes internes

- En Java, une classe (appelée Outer) peut en contenir une autre (appelée Inner).
- Dans ce cas Inner peut accéder à tous les champs et à toutes les méthodes privés de Outer car elles sont logiquement liés.
- Le compilateur Java crée un nouveau fichier de classe pour les classes internes, dans ce cas Outer$Inner.class. le processus de compilation crée également une méthode dite pont d'accessibilité entre les classes.
- Java 11 aide à générer ces méthodes de pont avec l'aide de “nestmates”. Les classes internes peuvent maintenant accéder aux champs et aux méthodes des classes externes sans travail supplémentaire du compilateur.
- Avant Java 11, il n'était pas possible d'accéder à un champ de la classe externe via une réflexion sans définir le contrôle d'accessibilité sur true. Avec le changement de contrôle d'accès imbriqué, ce contrôle n'est plus nécessaire.

Predicate et Optional

- Optional.isEmpty() apparaît pour savoir si l'optional est vide ou non.
- Predicate.not(Predicate) est une méthode statique qui renvoie un Predicate qui est l'inverse de celui passé en paramètre. Ceci peut rendre le code avec des références de méthode plus lisible.

Suppression de modules

Dans la version Java 11 :

- Java FX : une interface graphique en Java est supprimé.
- CORBA : un protocole de communication faisant transiter des objets est supprimé.
- Java JEE : certains modules, dont JAXB et JAXWS sont supprimés.

Ces modules ont été dépréciés depuis Java9 et sont (normalement) récupérables ailleurs.

Ce qu’il faut retenir

Java 11 apporte :

- L’inférence de type pour les lambdas
- Le lancement d’une JVM avec un fichier .java
- Des méthodes à String (pour gérer les caractères blancs notamment) et StringBuilder (et StringBuffer)
- Le Predicate not

Les modules CORBA, JavaFX et certains JEE ont été supprimés.

JDK 12-13

JDK 12-13

Instruction switch

Auparavant, l'opérateur switch était utilisé comme suit :

```
Etat etat = Etat.ARRETE;
switch (etat) {
case ARRETE:
     double vitesse = 0;
     break;
case DEMARRE:
     vitesse = 1;
     break;
case AVANCE_RAPIDE:
     vitesse = 2;
     break;
}
```

Cette syntaxe héritée du C nécessitait d'utiliser des break; et n'était pas très lisible.

De plus, les variables déclarées dans le switch étaient utilisables dans n'importe quel case après leur déclaration.

JDK 12-13

switch : nouvelle syntaxe

Switch peut maintenant être utilisé ainsi (en preview Java 12, intégré dans Java 14) :

```
switch (etat) {
case ARRETE -> {
     System.out.println("Arret");
}
case AVANCE_RAPIDE -> {
     System.out.println("Avance rapide");
}
}
```

- Les case sont dans des blocs de code.
- Les caractères -> séparent le cas des blocs de code qui vont contenir les instructions qui traitent les cas.
- Les breaks ne sont plus nécessaires. La syntaxe ressemble plus à celle des blocs de boucle et conditions. La portée des variables est réduite aux différents blocs des cas.

switch : exercice

Coder l'algo suivant avec l'ancien switch et le nouveau, en vous aidant de yield :

- en prenant en compte une variable int statut , et un booléen.
- l'algorithme met à jour une variable de type enum
- l'enum a trois valeurs : OK, KO, UNDEFINED
- si statut == 1, enum = OK, si statut ==0, enum = KO
- si statut == 2, et que le booléen vaut true, enum = OK
- si statut == 2, et que le booléen vaut false, enum = KO
- enum = UNDEFINED sinon

JDK 12-13

Les blocs de texte

Les blocs de texte ont été ajoutés en tant que ‘preview’ dans Java 13 et définitivement adoptés dans Java 15.

Ils améliorent l’ergonomie du développement avec du texte prenant plusieurs lignes.

Un bloc de texte est une String, mais sa valeur est définie différemment :

```
String textBlock = """
                Hé bonjour, comment allez-vous ?""";
String string = "Hé bonjour, comment allez-vous ?";
//Renvoie true
System.out.println(textBlock.equals(string));
//Renvoie 32
System.out.println(textBlock.length());
```

JDK 12-13

Intérêt des blocs de texte

Les blocs de texte améliorent la création de texte sur plusieurs lignes.

```
String premierTextBlock = """
          Le Bret.
          Si tu laissais un peu ton âme mousquetaire
          La fortune et la gloire…
          Cyrano.
                                  Et que faudrait-il faire ?
          (...)
          """;
//remplace

String string = "Le Bret.\n" +
          "Si tu laissais un peu ton âme mousquetaire\n" +
          "La fortune et la gloire…\n" +
          "Cyrano.\n";
```

JDK 12-13

Indentation du contenu des blocs de texte

- L'instruction suivante est indentée, pour des raisons d'indentation de code :

```
String premierTextBlock = """
          Le Bret.
          Si tu laissais un peu ton âme mousquetaire
          """;
```

- Prendre en compte l'indentation du code dans le bloc de texte ferait en sorte que le bloc de texte changerait, en fonction de l'emplacement de l'instruction Java.
- Le bloc de texte serait différent, selon que le développeur l'écrirait dans une méthode, ou dans une triple boucle imbriquée ....

JDK 12-13

Exercice : blocs de texte

Ecrire un programme Java qui affiche correctement le texte suivant :

```
<!DOCTYPE html>
<html>
   <body>

      <h1>My First Heading</h1>
      <p>My first paragraph.</p>

   </body>
</html>
```

Autres apports de la JDK 12

- Le formattage de nombre compact permet de transformer 12000 en 12K et 23 678 898 en 23M

```
NumberFormat fmt = NumberFormat.getCompactNumberInstance();
String result = fmt.format(13_250_350);
System.out.println(result);
```

- Et d’autres :

https://www.oracle.com/java/technologies/javase/12-relnote-issues.html#NewFeature

JDK 14

Changements dans les switches

- Tous les changements dans les switches passent de preview à permanent.

Clarifications sur la NullPointerException

- Au lancement de la JVM, l'activation du paramètre : ShowCodeDetailsInExceptionMessages permet d'avoir des messages plus explicites lors de la survenue d'une NullPointerException.
- La JVM sera lancée avec :
  - -XX:-ShowCodeDetailsInExceptionMessages pour ne pas activer cette fonctionnalité ou
  - -XX:+ShowCodeDetailsInExceptionMessages pour l'activer.

```
Exception in thread "main" java.lang.NullPointerException
     at c.NullPointerExceptionLauncher.callMethod2(NullPointerExceptionLauncher.java:23)
     at c.NullPointerExceptionLauncher.callMethod1(NullPointerExceptionLauncher.java:19)
     at c.NullPointerExceptionLauncher.main(NullPointerExceptionLauncher.java:12)
```

Live Monitoring

- Certains outils de monitoring doivent utiliser un fichier (créé par la JVM) pour le lire et afficher des informations.
- C’est le cas de Java Mission Control qui se base sur Java Flight Recorder pour acquérir des données de monitoring.
- La JVM peut maintenant envoyer ces données en direct.
- Le but est que les outils de monitoring puissent afficher des informations de supervision aussi en direct.
- Ceci permettra à des outils comme JDK Mission Control d’offrir de meilleures performances

JDK 14

Nouvel instanceof

- Ce changement est en preview et a été livré dans JDK 16.
- instanceof se voit ajouter un comportement qui le rend moins verbeux.
- La syntaxe ‘classique’ d’instanceof est la suivante : instanceof suivi de la déclaration d’une nouvelle variable, dont la valeur est un cast de la précédente.

```
if (meuble instanceof Chaise) {
      Chaise c = (Chaise) meuble;
      System.out.println("La chaise a " + c.pieds + " pieds.");
}
```

- Toutes ces opérations peuvent être fusionnées en une seule ligne :

```
if (meuble instanceof Chaise c) {
      System.out.println("La chaise a " + c.pieds + " pieds.");
}
```

instanceof : exercice

- Créer une classe mère Vehicule, et deux classes filles Auto et Velo.
- Auto définit la méthode rouleSurAutoroute().
- Velo définit la méthode rouleSurChemin().
- Créer une méthode qui prend en argument un Vehicule. La méthode permet de caster des instances de Vehicule vers Auto et Velo et de les faire rouler sur l'autoroute et sur le chemin selon le cas, avec l'ancien instanceof.
- Faire de même avec le nouveau instanceof

Outils associés à la JDK 14

- jpackage est un outil qui sert à créer des applications Java auto contenues (JEP 343: Packaging Tool (Incubator)).
- L'API jdk.incubator.foreign permet d'accéder à de la mémoire en dehors de la heap.
- Le ramasse-miettes Z (Z Garbage Collector) est disponible pour Linux, Windows et MacOS.

JDK 15

JDK 15

Classes scellées

- Les classes scellées ont été proposées avec la version 15 de Java et livrées avec la version 17 de Java.
- Elles permettent de sceller, c'est à dire de définir de façon explicite les classes qui peuvent hériter d'une autre.
- Une classe scellée a des classes filles, mais la classe mère définit la liste exhaustive des classes qui peuvent en hériter.
- Les mots clés sealed et permits ont été ajoutés. Ci-dessous la classe mère Celeste permet aux classes Planete, Comete et Etoile d'en hériter :

```
public sealed class Celeste permits Planete, Comete, Etoile
```

Classes scellées : contraintes

Les classes filles définies avec permits ont les contraintes suivantes :

- Elles doivent être accessibles par la classe scellée à la compilation
- Elles doivent hériter directement de la classe scellée
- Elles doivent définir comment elles perpétuent le sceau de la classe mère, en ayant exactement un des attributs suivant :
  - final: nul ne peut en hériter
  - sealed: elles définissent avec permits la liste exhaustive de leurs classes filles
  - non-sealed: N'importe quelle autre classe peut en hériter
- Si la classe scellée est dans un module nommé, les classes filles doivent être dans le même module.
- Si la classe scellée n'est pas dans un module nommé, les classes filles doivent être dans le même package.

JDK 15

Classes scellées : vérifications à la compilation

- Les classes scellées apportent du contrôle sur le diagramme d'héritage des classes.
- Elles apportent aussi du contrôle à la compilation. En effet, le compilateur connaît le diagramme complet d'héritage d'une classe scellée. Il peut donc empêcher un développeur de créer un instanceof inutile.

https://docs.oracle.com/en/java/javase/20/language/sealed-classes-and-interfaces.html

Interfaces scellées

- Une interface peut être aussi scellée : l'interface définit alors de façon explicite les classes qui l'implémentent, et les interfaces qui en héritent.
- Les mêmes contraintes que vues précédemment s'appliquent.

Fonctionnalités dépréciées

Tout est là : https://www.oracle.com/java/technologies/javase/15-relnote-issues.html

JDK 15

jpackage

jpackage est un outil en ligne de commande, qui :

- Permet de créer des packages contenant une application et une JVM. Ces packages correspondent au formats classiques des plateformes cibles:
  - msi et exe pour Windows,
  - pkg et dmg pour macOS,
  - deb et rpm pour Linux.
- Permet de spécifier les paramètres de lancement au moment du packaging.

JDK 17

Apports de la LTS 17

- Switch et Pattern matching (preview) : switch pourrait évoluer pour gérer des patterns, tout comme le fait instanceof.

```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

- java.util.HexFormat permet de convertir les types primitifs et byte[] vers des valeurs hexadécimales.

https://docs.oracle.com/en/java/javase/17/

LTS, non LTS

- LTS signifie : Long Time Support. Ce sont des versions de Java qui sont censées être supportées longtemps.
- Pour le moment, les versions LTS sont les 7, 8, 11, 17 et 21.
- Ceci concerne en théorie les JDKs fournies par Oracle. En pratique, pour le moment, les fournisseurs de JDKs suivent le même mode de fonctionnement.
- Ceci amène plus naturellement les développeurs et administrateurs à favoriser les versions LTS pour la production. Les autres versions peuvent être utilisées pour la formation, les prototypes, ou pour utiliser une fonctionnalité de Java avant qu'une version LTS ne l'embarque.

https://www.oracle.com/java/technologies/java-se-support-roadmap.html

Fin

Conclusion

- Questions ? Réponses !

Fin

Conclusion

Merci pour votre attention !