# FORMATION

## Java - Nouveautés des versions 8 à 21

09/10/2024 – 11/10/2024

m2iformation.fr

# Bienvenue sur cette formation M2I

- Victor Dauphin
  Formateur indépendant et
  Développeur Web Fullstack Consultant

- Technologies de prédilection: Spring & Angular

- Me contacter: victor.dauph@gmail.com
- Si retard ou absence: 06 95 19 99 40

# Le réseau M2I Formation

# Le groupe M2I

- Le groupe M2i est leader de la formation IT, Digital et Management en France depuis plus de 35 ans.
- L’engagement pour la qualité en étant certifié Qualiopi et Datadock.
- Plus de 300 collaborateurs dédiés à la montée en compétences de votre capital humain.
- Le catalogue M2I : https://www.m2iformation.fr/catalogues/
- Engagement vers la féminisation du numérique : https://www.m2iformation.fr/numerique-au-feminin/
- La démarche qualité : https://www.m2iformation.fr/demarche-qualite/

# Horaire et convocations

- 9h - 17h (alarm)
- 15 mn de pause le matin (alarm)
- 1h de pause déjeuner (alarm)
- 15 mn de pause l’après midi (alarm)
- Dernier jour à 16h30 (si le plan de cours est terminé uniquement)

# Présentations et prérequis

Prérequis:

- Connaissances pratiques de Java
- JAV-SE  ’Java - Les fondamentaux de la programmation’
- JAV-DVO "Java - Pour les développeurs objet
- Avoir installé JDK22 et un IDE (Eclipse ou autre)
- Droit admin

Présentation stagiaires :

- Est-ce que tout le monde a les prérequis?
- Quelles sont vos attentes?
- Est-ce qu’il y’a des certifications?

# Déroulé et structure de la formation et formalités

- Première chose à faire : signer les feuilles d'émargement (ni croix ni initiales)

- Dernier jour de la formation : le centre de formation à l'obligation contractuelle de fournir vos évaluations à votre entreprise avant 15h, donc au retour de la pause déjeuner, 2 ou 3 h avant la fin de la formation, je vous ferai remplir les évaluations formateur.

- 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.

- Si il me reste du temps, je reviens sur toutes les questions hors plan de cours que les stagiaires m'ont posées durant la formation.

# Structure et versionning Github

- Un repo Github pour les structures et corrections d’exercies
- https://github.com/VictorDauph/Jav-New
- 1 notion = 1 chapitre = 1 comit Github

# Support de cours et outils pédagogiques

- Lien vers le Slack ou Discord de la formation.
- Qualiopi : fournir le support de cours et les énoncés des tp de validation des acquis.
- Support de cours sur Teams et Github ou Drive

# Validation des acquis au quotidien

- Google Forms de demi-journée pour la validation des acquis et l'adaptabilité.
- Ici je met le lien vers le google form préparé la veille de ma formation et je le met aussi sur slack ou discord.

# L’application et les TP de la formation

- Nous allons développer ensemble différentes fonctionnalités d ’un CRM en utilisant les outils des nouvelles versions de Java.

# Plan de cours

- Chapitre 1 :Classloading, modules et génération de JAR
- Chapitre 2 : Programmation Concurrente
- Chapitre 3 : Programmation Fonctionnelle et Lambdas
- Chapitre 4 : Switchs et Patterns
- Chapitre 5 : Nouveautés Complémentaires de Java

# Téléchargement et installation des composants

- Télécharger et décompresser :
- OpenJDK 22.0.1:https://www.oracle.com/java/technologies/javase/jdk22-archive-downloads.html
- Un client Git à jour. Vérifier que vous pouvez cloner le repo de la formation.

  https://github.com/VictorDauph/Jav-New
- Un IDE . La formation se fera avec Eclipse, mais n’importe quel IDE fera l’affaire.

# Bilan de début de formation

- Retour de début de formation à M2I
- contact@2aiconcept.com en copie
- Au plus tard à 10h30 le premier jour de la formation.

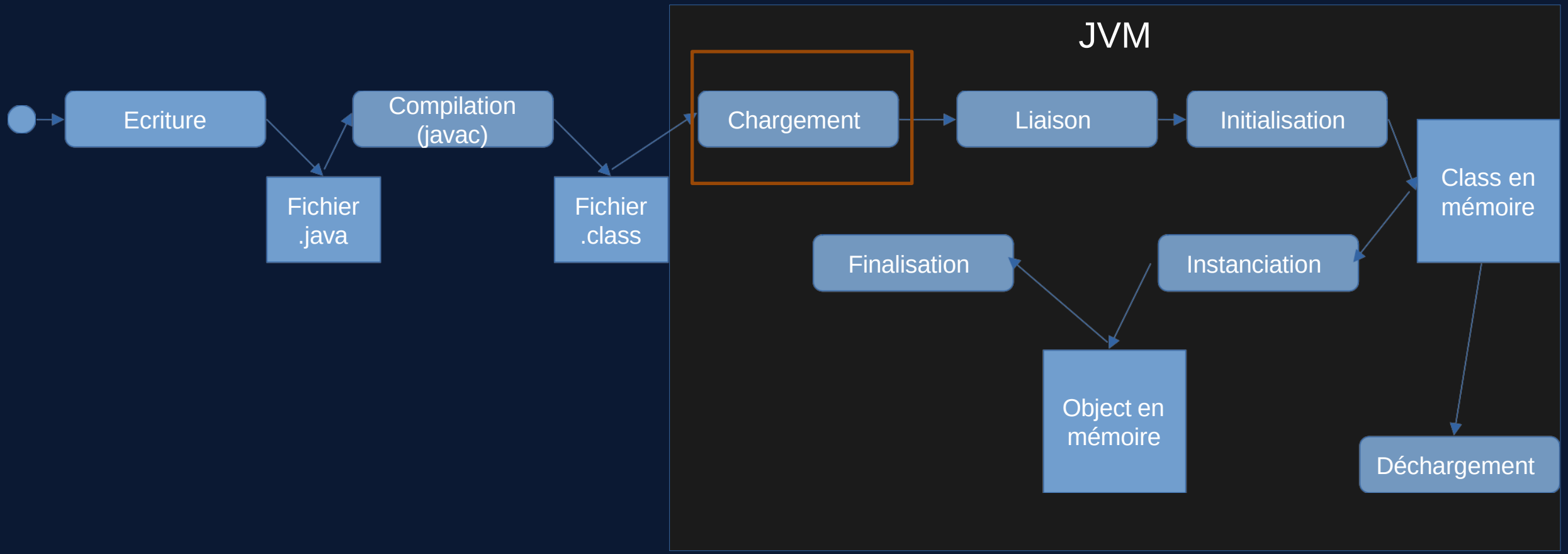
# LTS, non LTS

- 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

# Classloading

# Cycle de vie d'une classe / d'un objet

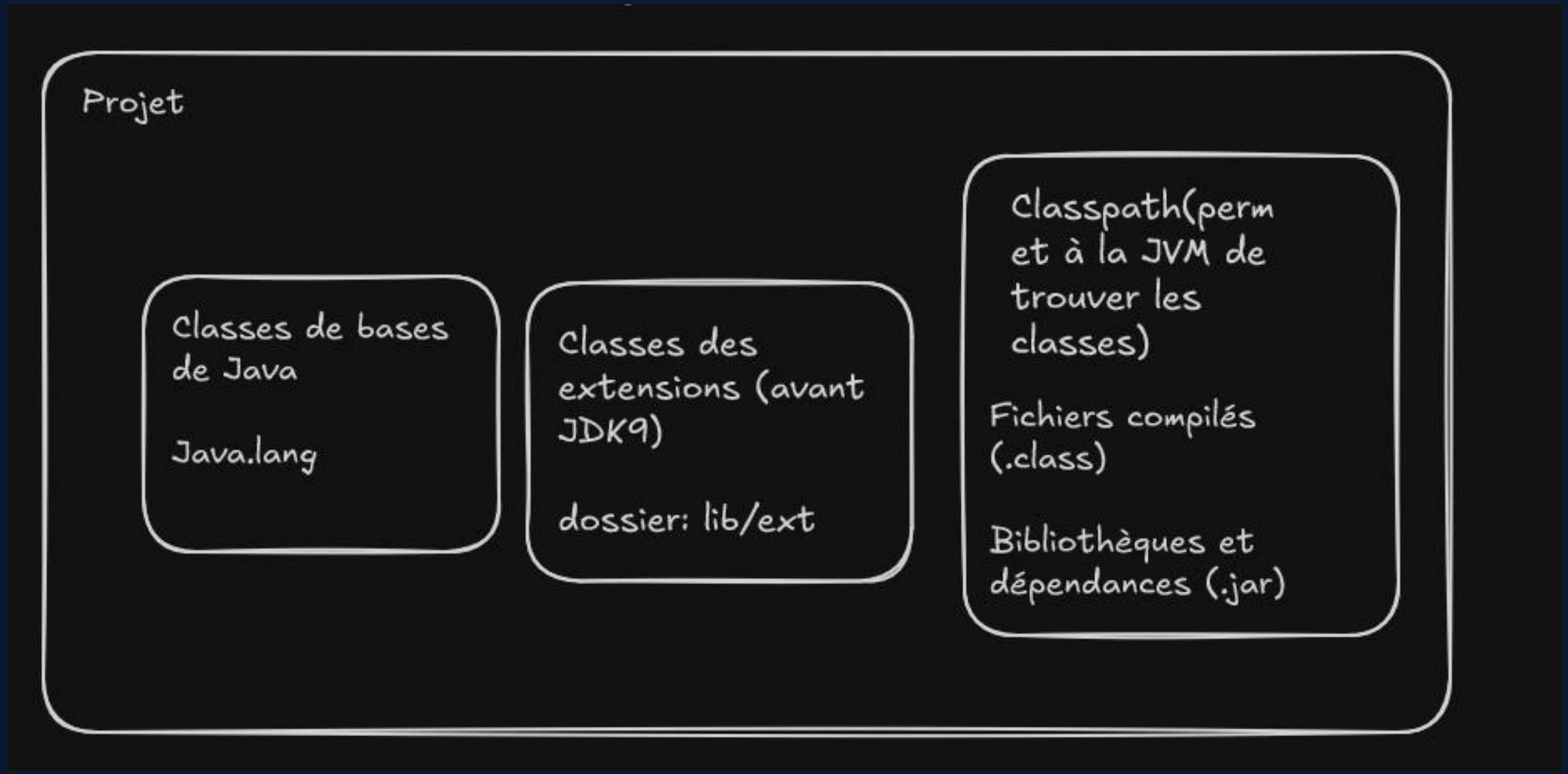


# Le classpath

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.


Projet
Classes de bases de Java
Java.lang
Classes des extensions (avant JDK9)
dossier: lib/ext
Classpath(perm et à la JVM de trouver les classes)
Fichiers compilés (.class)
Bibliothèques et dépendances (.jar)

# 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 ...)

# Rôle des ClassLoaders

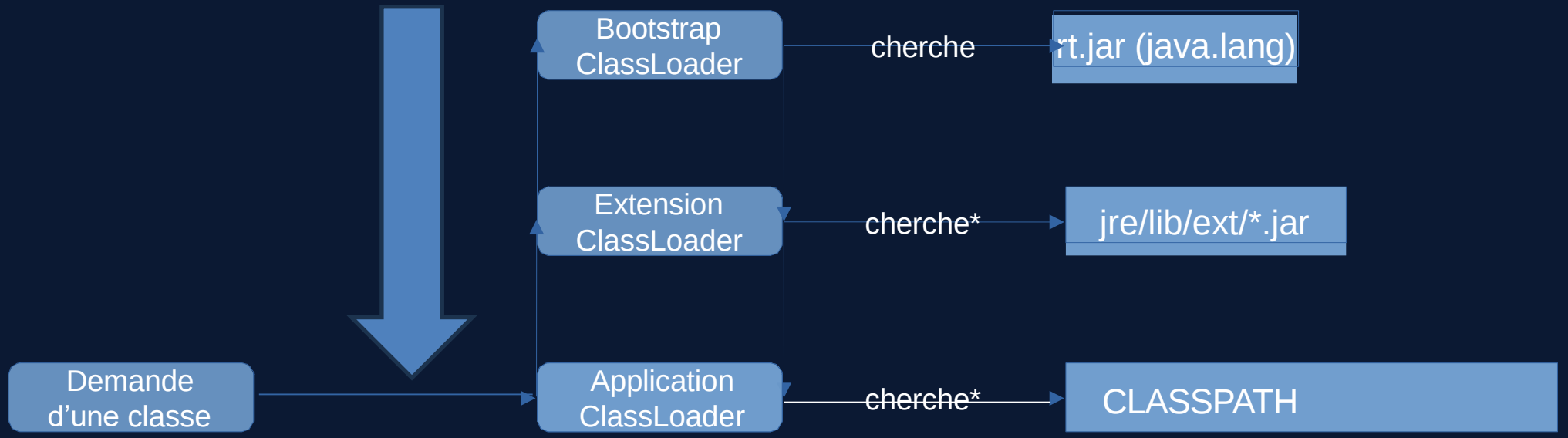
- Les ClassLoader Java font partie de l’environnement d’exécution de la JVM. Le système d'exécution Java n'a pas besoin de connaître les fichiers et les systèmes de fichiers grâce aux classloaders.
- Les classes Java ne sont pas chargées en mémoire en une seule fois au démarrage de la JVM, mais lorsqu'une 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...).

# Hiérarchie des ClassLoaders

- BootStrap ClassLoader n'a aucun ClassLoader parent. Il est également appelé Primordial ClassLoader .
- Extension ClassLoader : l'extension ClassLoader est un enfant de Bootstrap ClassLoader et charge des classes Java à partir des bibliothèques d'extensions JDK
- System ClassLoader : Il charge les classes trouvée dans le CLASSPATH.

- Les classloaders ont une organisation hiérarchique : un classloader demande toujours à son classloader père  d'essayer de charger la classe.
- Une classe est associée au classloader qui l'a chargée. Une fois une classe chargée, celle-ci est identifiée par son nom et son classloader. Ainsi, deux classes de même nom chargées par deux classloaders différents sont considérées comme différentes par la JVM.

# Hiérarchie des ClassLoaders




*si le ClassLoader parent n'a rien renvoyé.

# Avec un IDE et Maven

- L’utilisation du Classpath est incoutournable pour développer et déployer une application utilisant des composants externe
- 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.

https://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

# 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 qui contiennent des classes et les classes du projet.

# 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).

# 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
  - D’exporter (ou non) ses packages
  - de déclarer des services.

    de déclarer les modules dont a besoin un autre module.

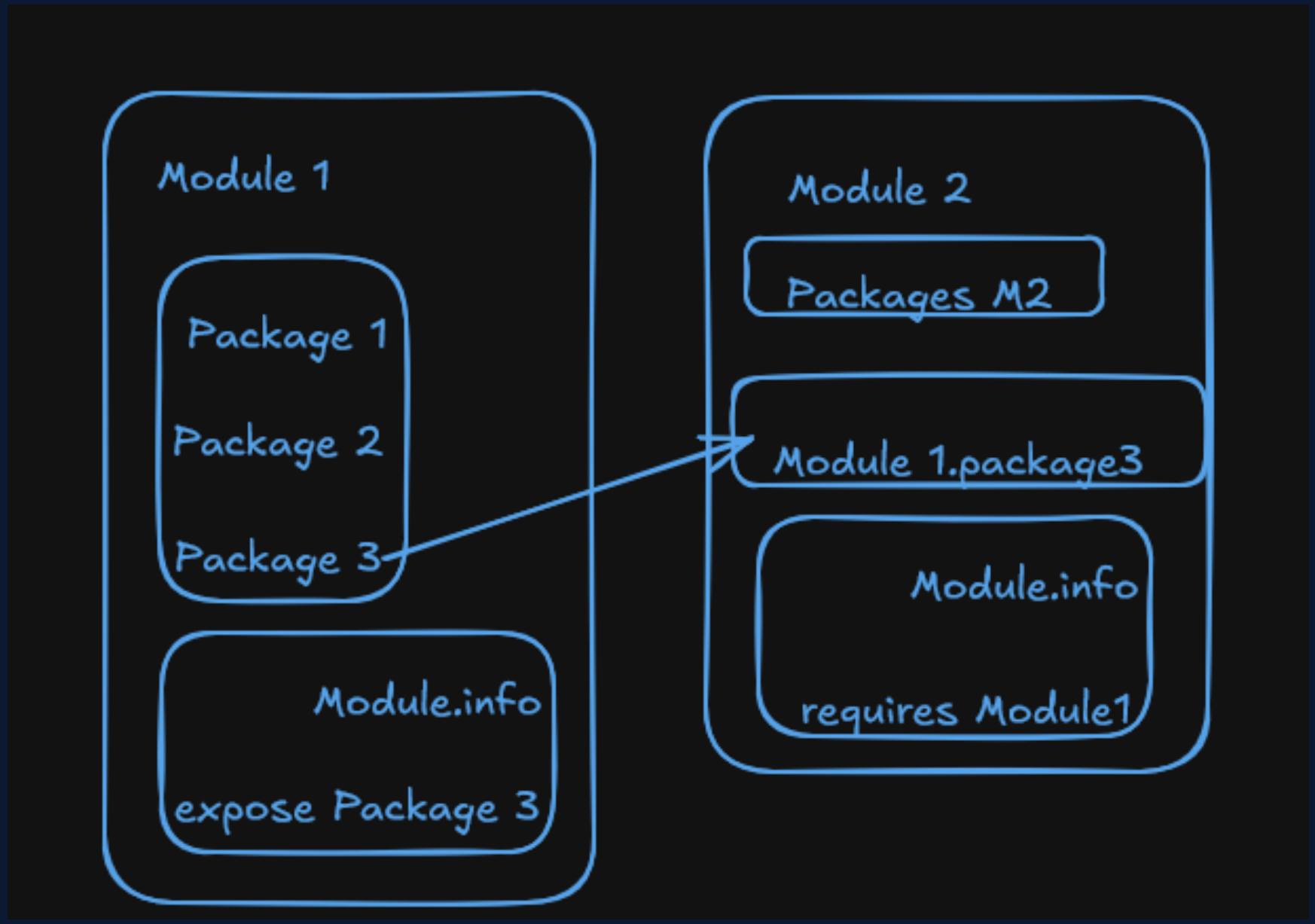
    d’effectuer des vérifications à la compilation, et/ou à l’exécution, pour s’assurer que les modules requis sont bien présents.

# Déclaration

Un module doit avoir :

- ses sources dans son répertoire
- un fichier module-info.java à la racine de l'arborescence des sources

Le fichier module-info.java est optionnel mais s'il est absent tous les packages du module sont exposés.




La doc: https://docs.oracle.com/javase/9/docs/api/java/lang/module/package-summary.html

# Contenu du fichier module-info.java

Tout comme META-INF/MANIFEST.MF, le fichier module-info.java contient des méta-données:

- le nom du module
- la liste des dépendances
- les packages exposés
- les services exposés et/ou consommés.
- Les imports.

# 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 module.

# 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.
- ex:  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.
  Donne accès à tous les types (classes,interfaces ...) publics des packages **exportés** par moduleb.

- 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 module qui a besoin de ClasseB doit déclarer moduleb comme une dépendance en utilisant le mot clé requires

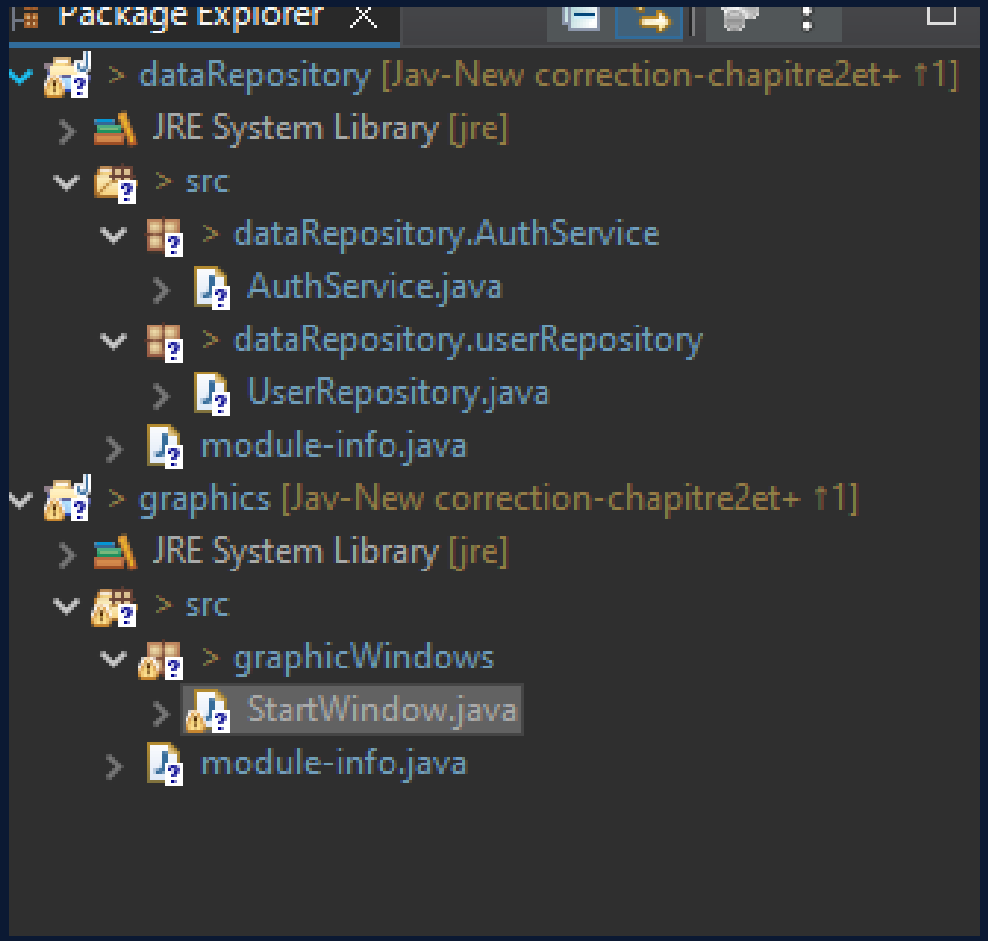
/!\ Les sous-packages d'un package exportés ne sont pas accessibles : il faut exporter explicitement tous les sous-packages concernés un par un.

# 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.

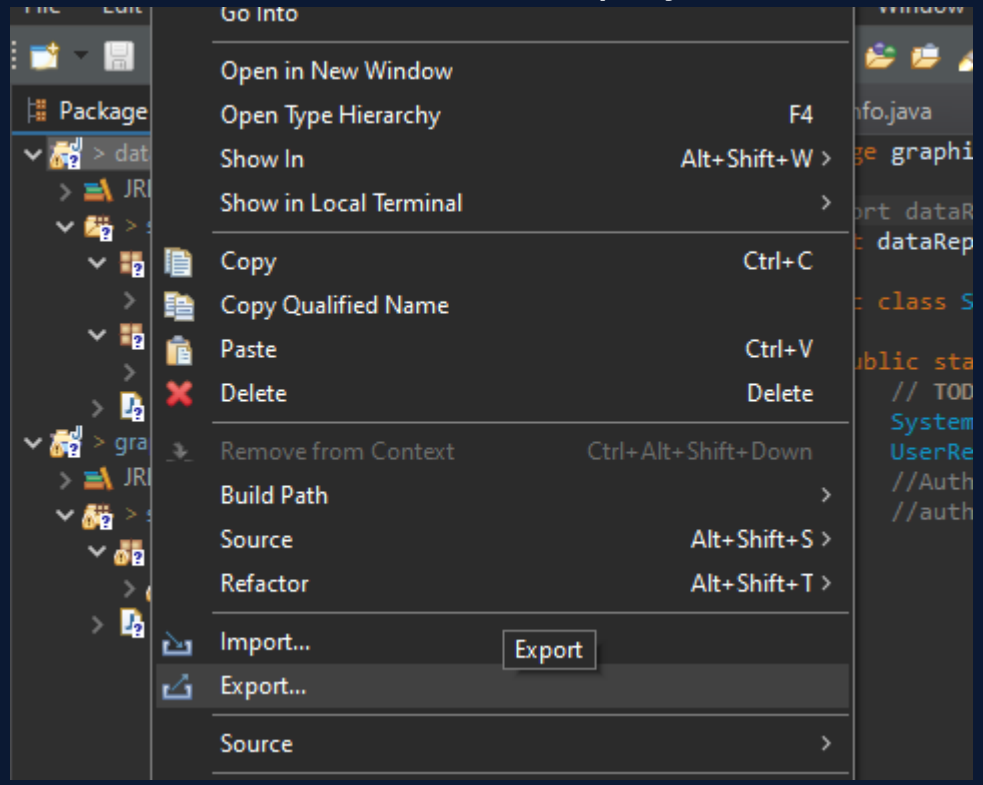
# Exemple de déclaration dans un projet multi-modules

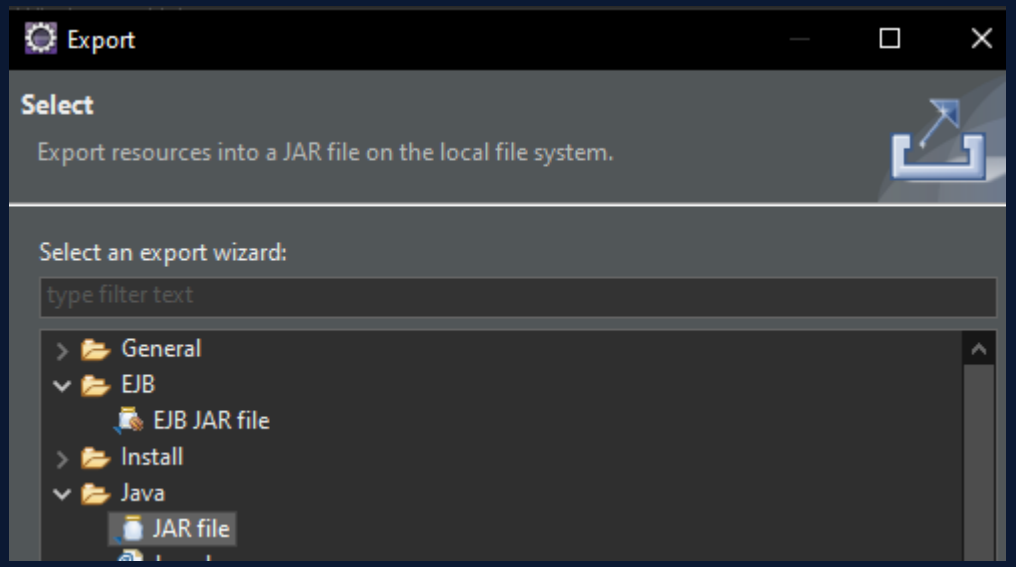
- Le code source de Java est organisé ainsi :

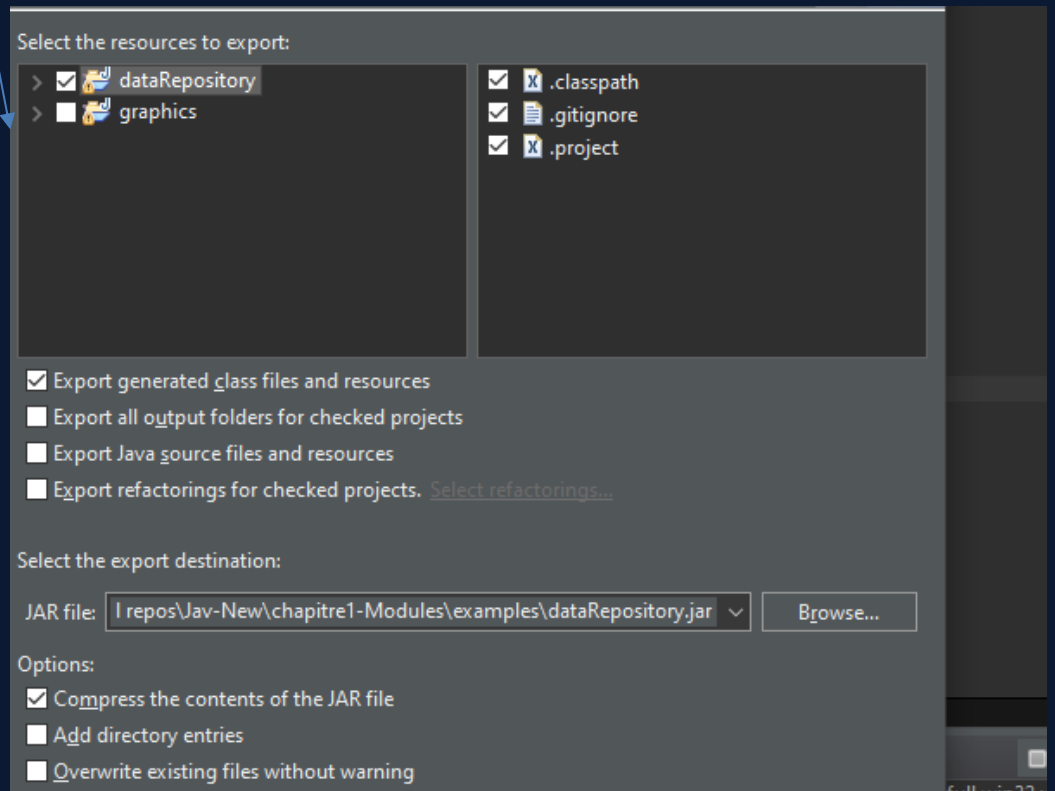


# Exporter un Jar

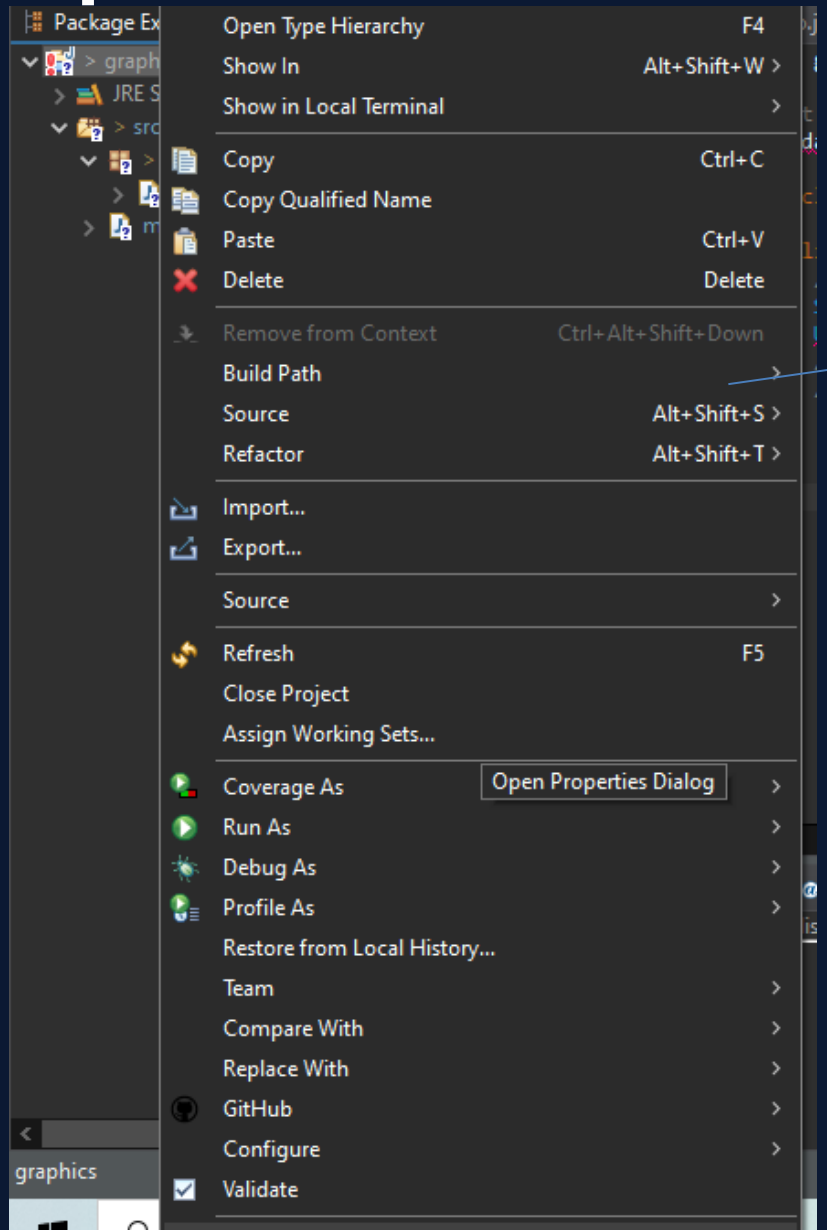
Clic droit sur le projet

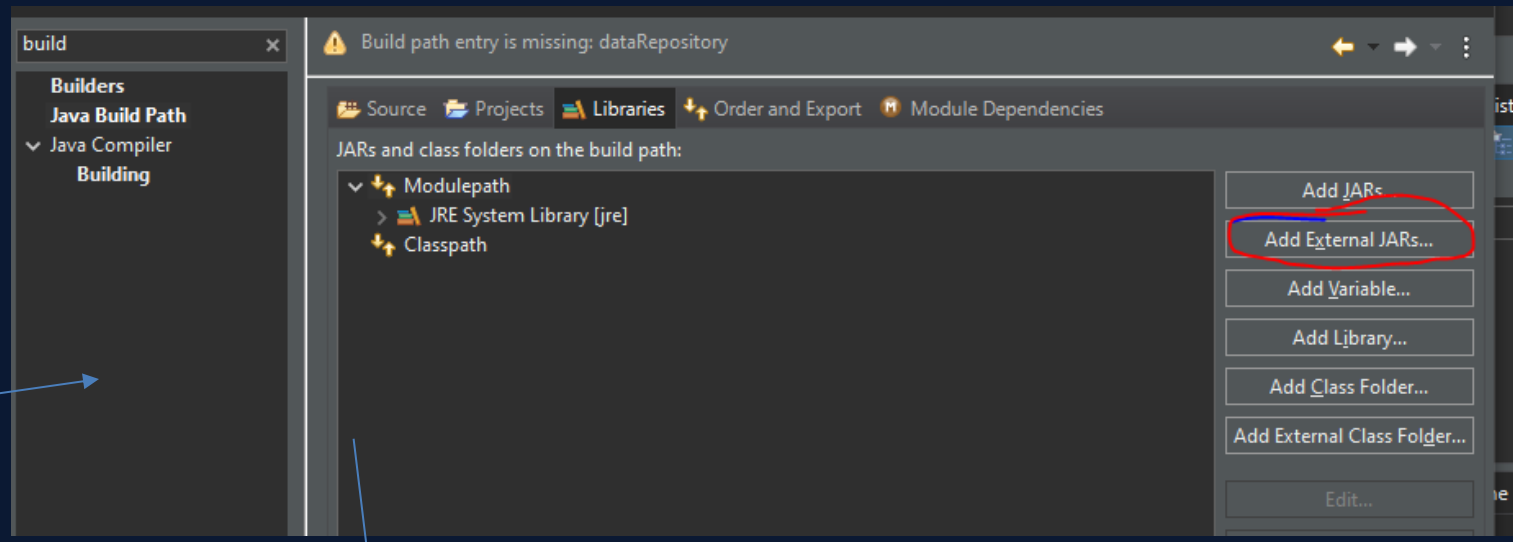


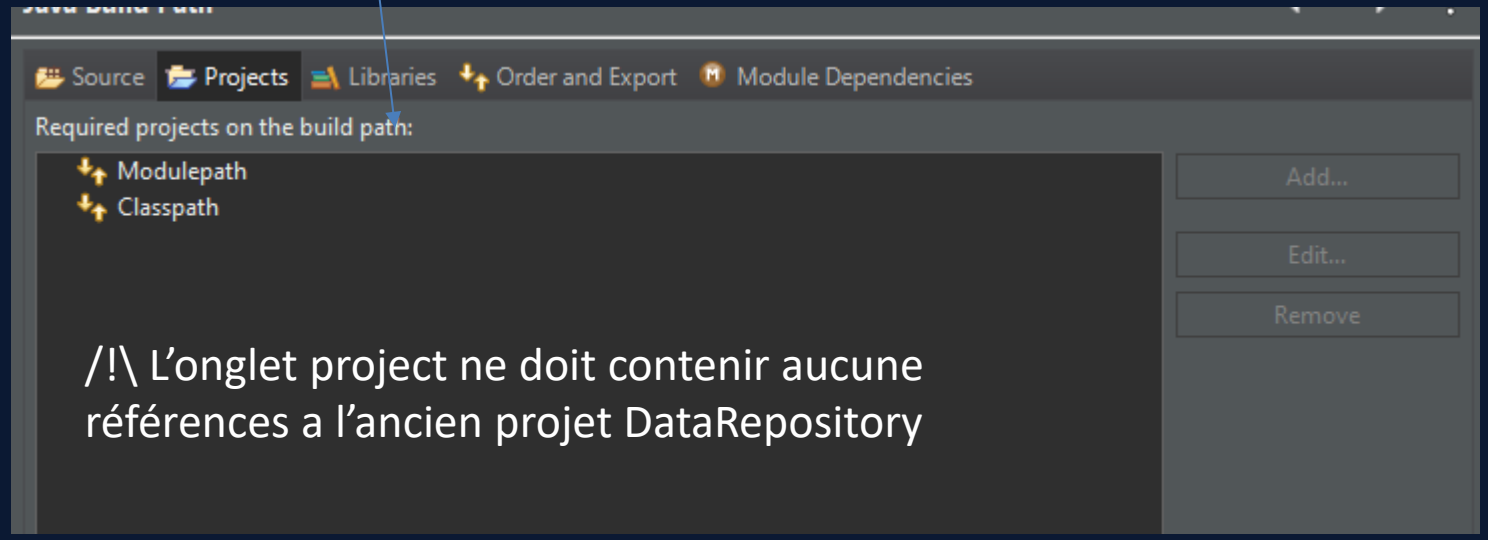



# Importer un Jar




/!\ L'onglet project ne doit contenir aucune références a l'ancien projet DataRepository

# COMMIT

# 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 contient les sous-projets :

- graphics
- dataRepository

- La classe graphic.graphicWindows.StartWindow utilise .dataRepository.UserRepository
- 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.

# COMMIT

# 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.
- Tester avec le main de StartWindow

# COMMIT

# 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 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

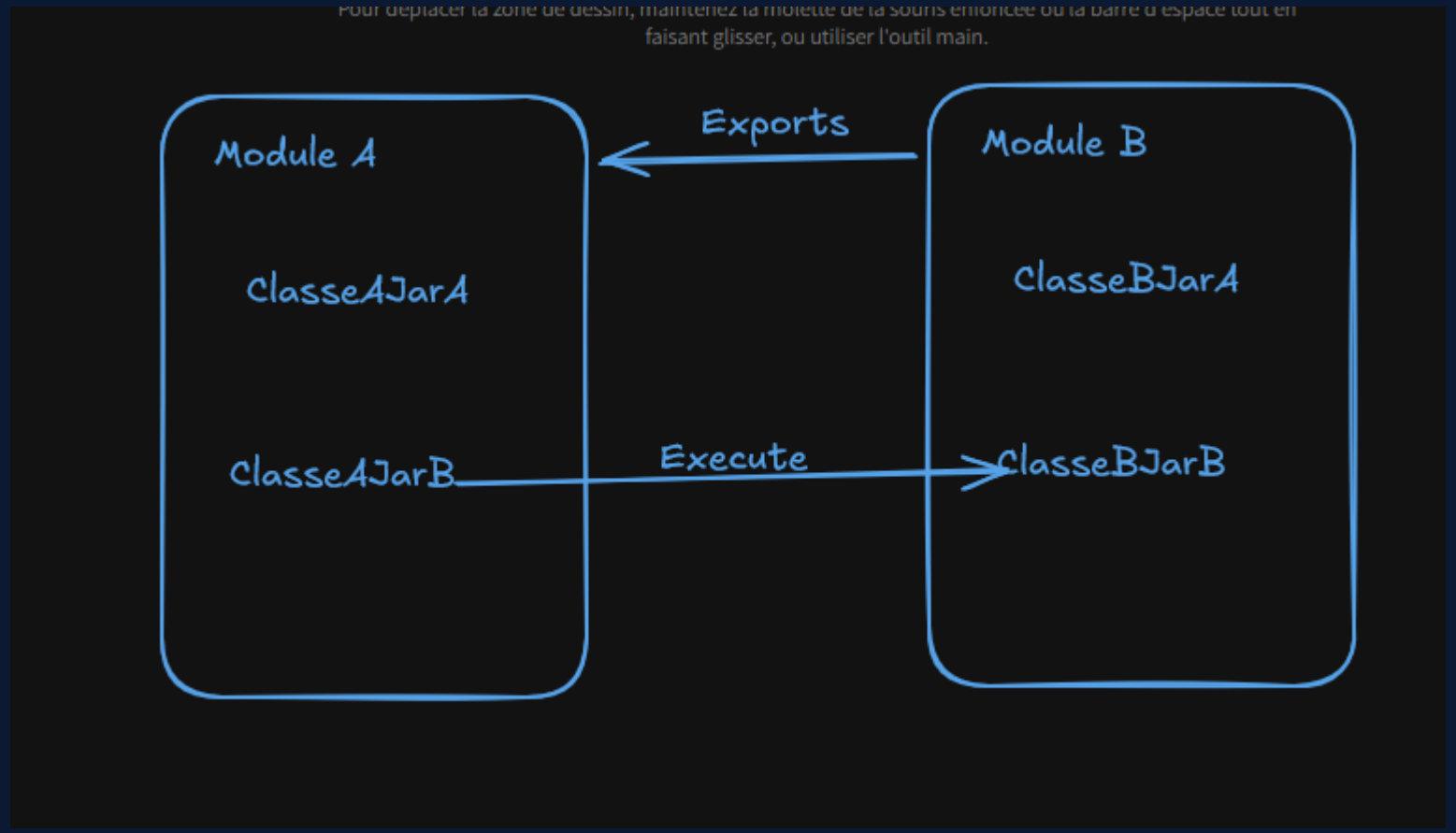
- Un module A peut dépendre d'un module B pour utiliser ses classes, et ces classes peuvent elles-mêmes utilisées dans les méthodes que le module A exporte.

```
// 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 (!) .


Pour déplacer la zone de dessin, maintenez la molette de la souris enfoncée ou la barre d'espace tout en faisant glisser, ou utiliser l'outil main.
Module A
ClasseAJarA
ClasseAJarB
Exports
Execute
Module B
ClasseBJarA
ClasseBJarB

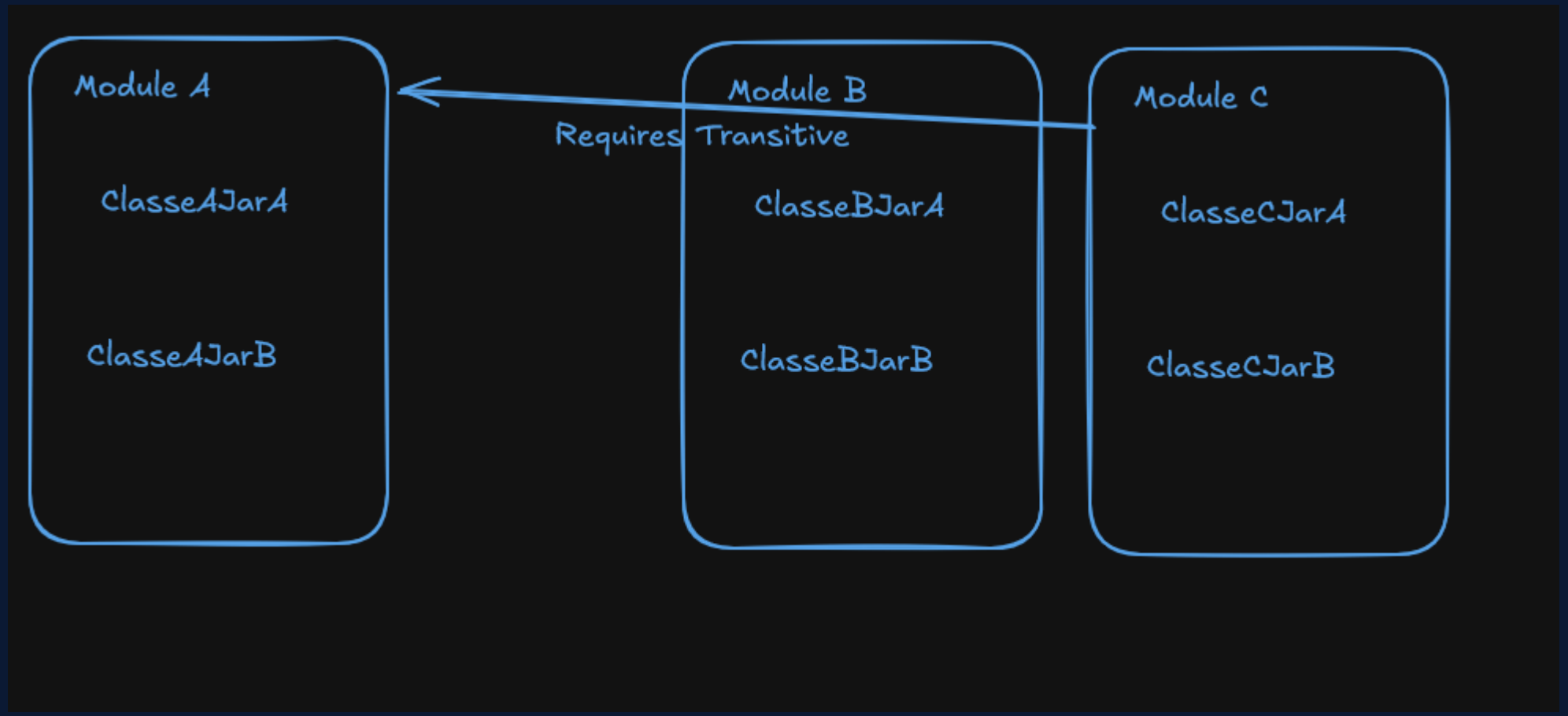
# Graphe de dépendances : solution

- 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 .


Module A
ClasseAJarA
ClasseAJarB
Module B
Requires Transitive
ClasseBJarA
ClasseBJarB
Module C
ClasseCJarA
ClasseCJarB

# 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

# Exemple Transitive

- Points d'alerte:

  Les packages doivent être en camelCase

  Le pom du parent doit contenir:

```
<properties>
<maven.compiler.source>9</maven.compiler.source>
<maven.compiler.target>9</maven.compiler.target>
</properties>
```

  A doit être dans le classpath de c

# COMMIT

# 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.

# TP

Étoffer les modules data-contract, et graphics :

- Raccorder le module business, prenant place ‘entre’ data-contract et graphics.
  - business utilise data-contract et graphics utilise business.
  - Créer dans data-contract des interfaces avec des méthodes (autant que vous  voulez). Implémenter les méthodes correspondantes dans business. Utiliser les interfaces de data-contract dans graphics.

# COMMIT

# 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 :
  - Oui si contraintes de sécurité, projet neuf, pas de framework.
  - Oui si on veut générer des .jar et que notre projet est déjà modulaire.

Else

- Non

On peut alternativement utiliser des dépendances Maven, ou un framework d'injection de dépendances (comme Spring, CDI, ou Dagger) pour gérer la création et la gestion du cycle de vie des objets.

# Programmation concurrente

# Les concepts de la programmation multi-thread

1. Création de threads

   Étendre la classe Thread ou implémenter Runnable.
   Utiliser ExecutorService pour gérer un pool de threads

2. Gestion des threads

   Cycle de vie : Nouveau → Exécutable → En cours → Bloqué/En attente → Terminé.
   Méthodes wait(), notify(), join() pour coordonner les threads.

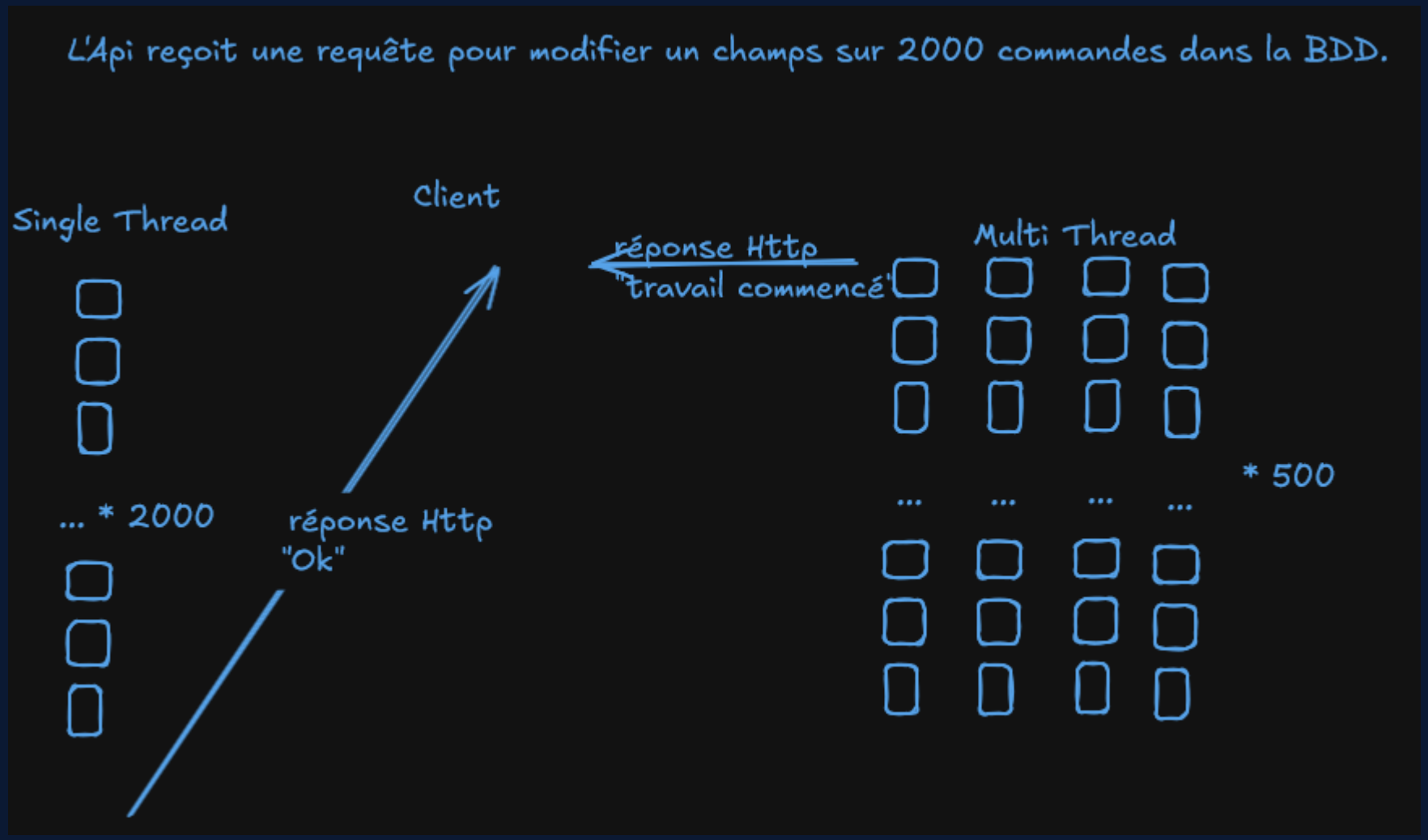
3. Outils de Concurrence

   CountDownLatch, CyclicBarrier pour la synchronisation de groupe.
   Semaphore pour limiter l'accès aux ressources partagées.

4. Programmation asynchrone

   CompletableFuture pour des tâches non-bloquantes et asynchrones.

# Les concepts de la programmation multi-thread




https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

# Example : 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();
```

# Example: 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");
	}
```

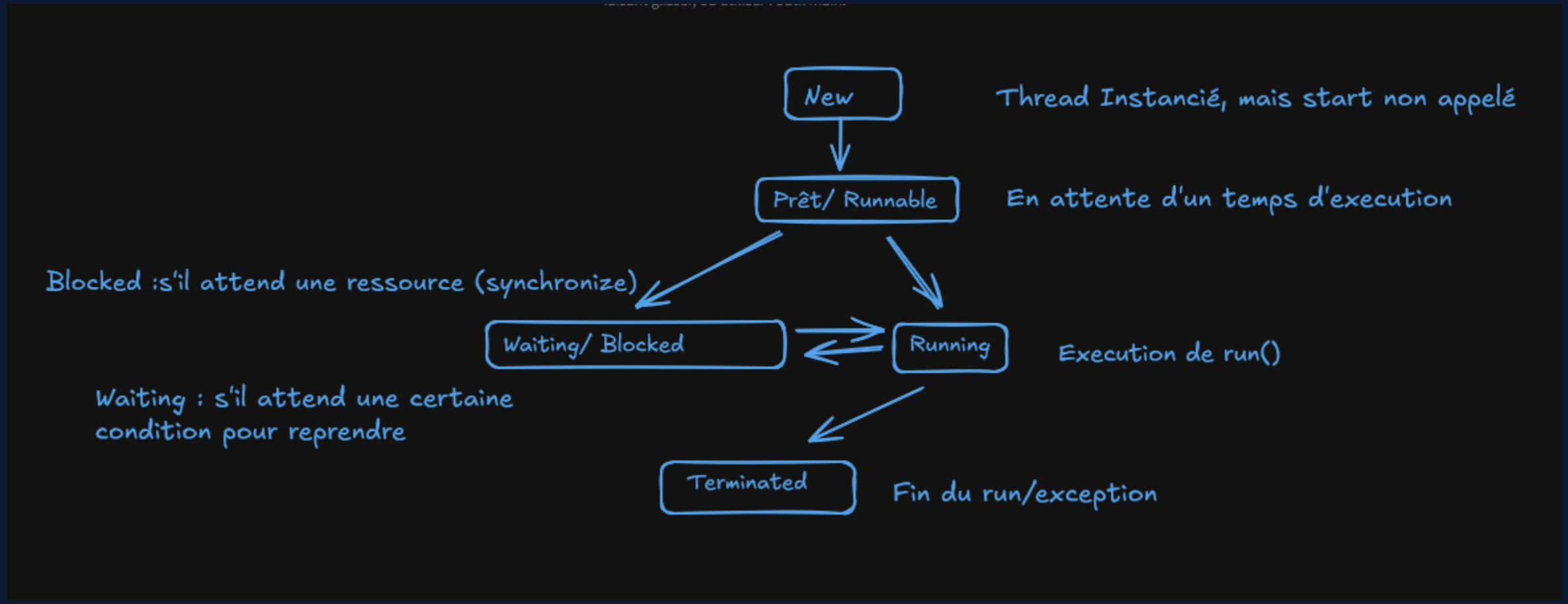
- Puis créer un Thread avec ce Runnable et lancer start() sur le Thread créé :

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

- La seconde version est préférée, car pour créer une hiérarchie de classe pour les threads, il vaut mieux étendre d'une interface que d'hériter d'une classe.

# COMMIT

# Cycle de vie d’un Thread



# 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

**Executor :** Gère les threads à un niveau plus abstrait. Ne lance que des runnable

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

**ExecutorService :** Interface plus complète permettant :

- Arrêt des threads.
- Lancer des Callables.

```
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*

# COMMIT

# Les barrières cycliques

Une barrière cyclique (CyclicBarrier) est une barrière réutilisable (d'où le terme cyclique). Elle fonctionne comme suit :

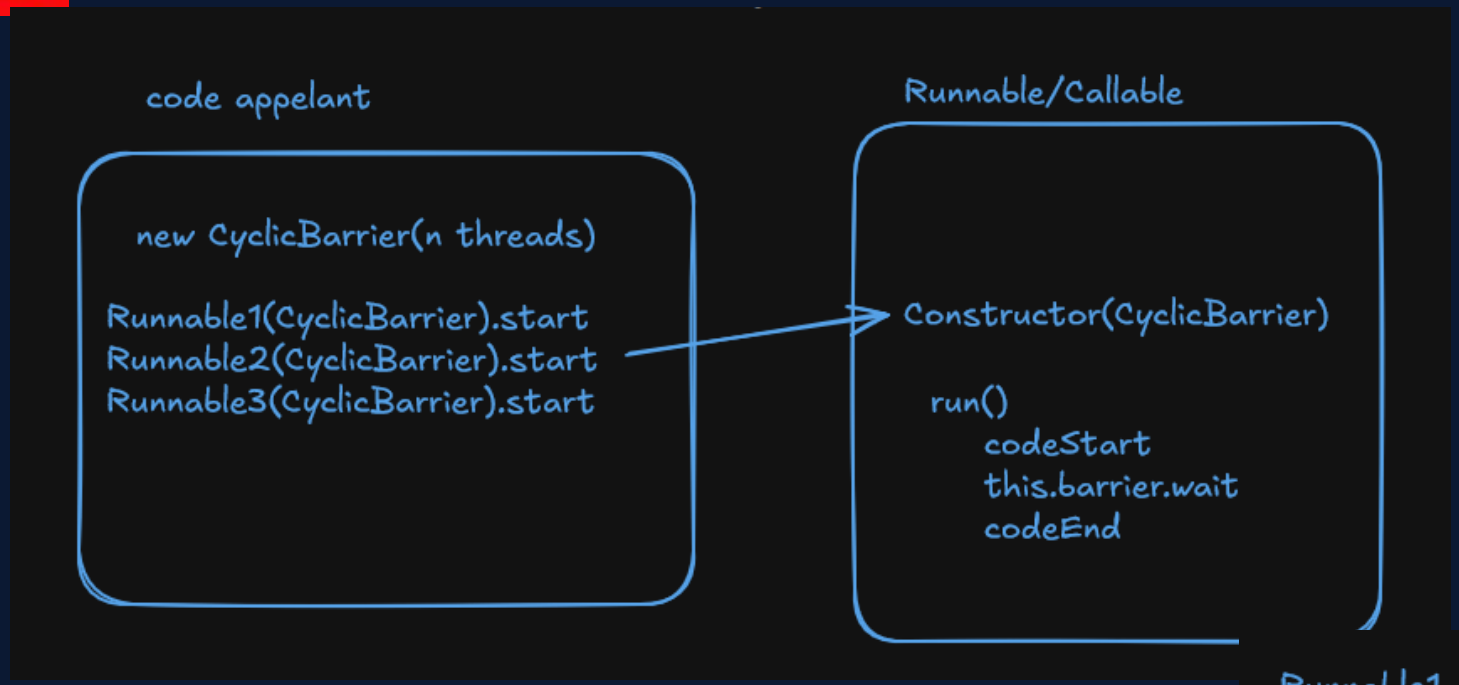
- Initialisation : Nombre de threads à attendre + Action optionnelle.
- Méthode : CyclicBarrier.await()

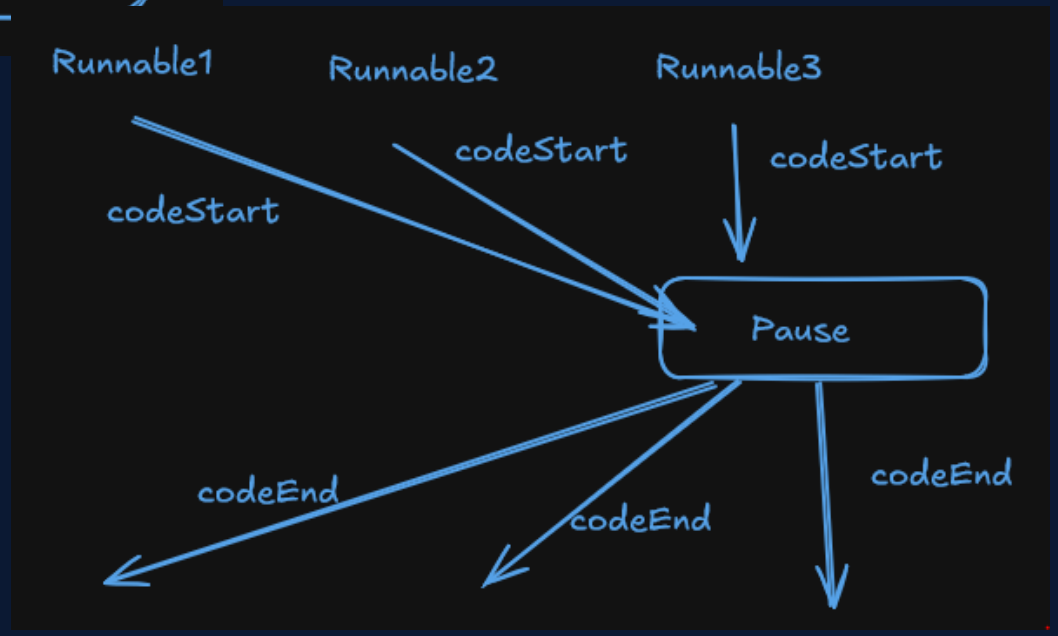
- Incrémente le nombre d'attentes.

- État des Threads :

- BLOCKED si nombre d'attentes < nombre de threads à attendre.
- RUNNABLE lorsque la barrière s'ouvre.

- Action : Effectuée une fois le seuil atteint.




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");

}
```

```
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.

# COMMIT

# L’interface Lock

Fonctionnalités :

- Créer des verrous : lock() (hors synchronized)
- tryLock() : verrouillage non-bloquant / délai
- Verrou jusqu'à interruption du thread

Avantages :

- Évite les deadlocks
- Contrôle du verrouillage (méthodes synchronisées)

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

# Les sémaphores

Utilité des sémaphores : Un sémaphore est particulièrement utile quand vous avez un nombre limité de ressources partagées (comme des connexions de base de données) et que vous souhaitez empêcher qu'un trop grand nombre de threads y accèdent simultanément.

Exemple simple : Imaginez un parking avec 3 places. Chaque voiture (thread) doit attendre une place libre avant de se garer (acquire), et quand elle quitte, elle libère une place (release). Si toutes les places sont prises, les voitures doivent attendre qu'une place se libère.

Comportement bloquant : Lorsqu’un thread appelle acquire() et qu’il n’y a plus de permis, il est mis en attente jusqu’à ce qu’un autre thread appelle release(). Ce mécanisme peut éviter la surcharge d'une ressource critique.

Justesse (Fairness) : Avec un sémaphore juste, les threads sont débloqués dans l'ordre dans lequel ils ont été mis en attente. Sinon, il peut y avoir une certaine forme de "désordre", ce qui peut être utile dans certains cas pour des optimisations.

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

# Fork/Join

Implémentation d'ExecutorService

- Optimisation pour les multiples cœurs.

Diviser pour régner

- Divise une grosse tâche en tâches plus petites.
- Efficace pour les tâches indépendantes.

Utilisation de ForkJoinPool

- Exécute des ForkJoinTasks.

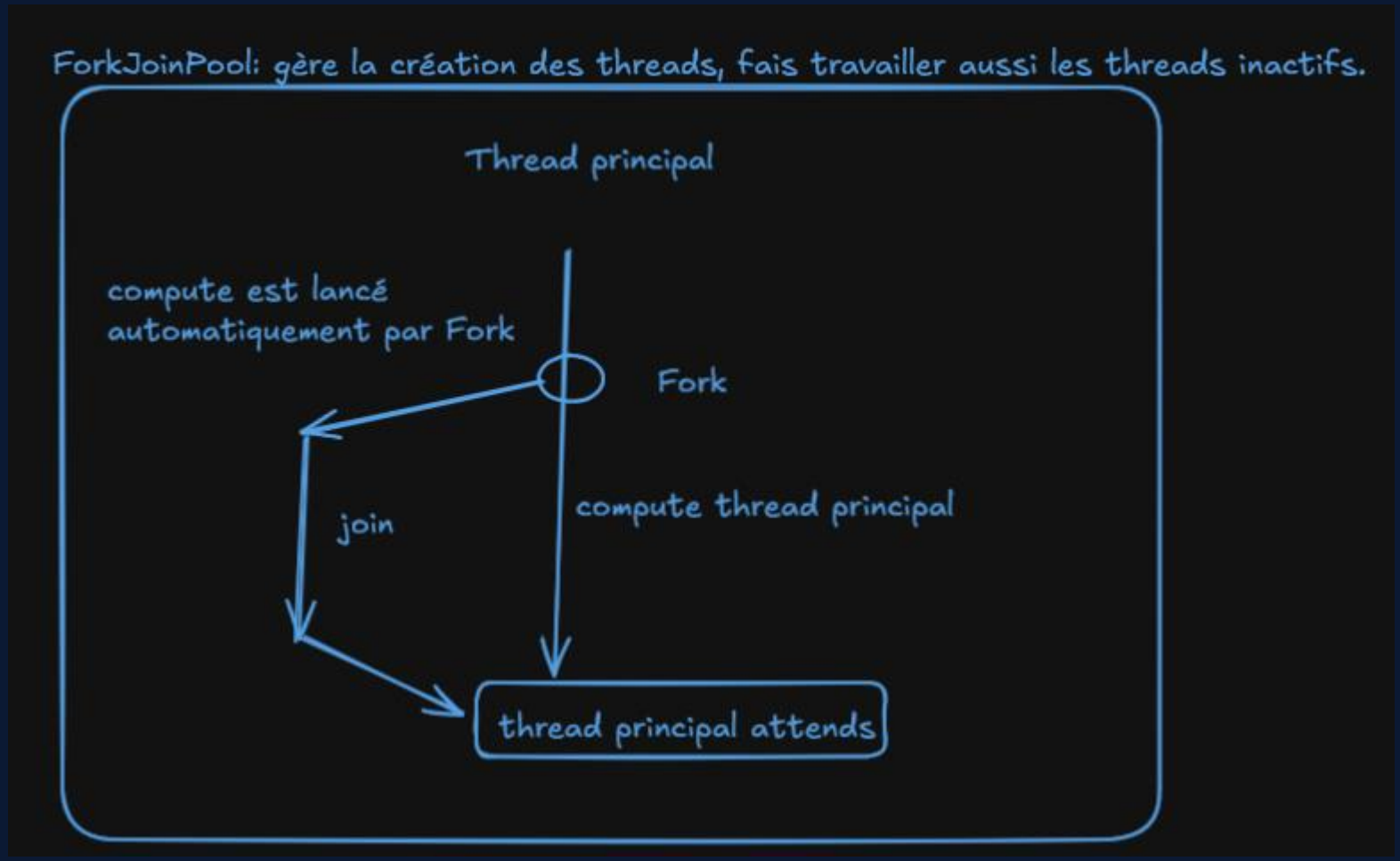
Méthodes intégrées

- Exemples : Arrays.parallelSort(), Streams (avec lambdas).

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

# Fork/Join

Voir exemple package forkJoin



# CompletableFuture

Qu'est-ce que c'est ?

Asynchrone : Permet d'exécuter des tâches de manière non bloquante.
Résultats : Représente un résultat d'opération qui peut être complété à l'avenir.

Caractéristiques Principales

Chaining : Supporte la composition de tâches via des méthodes comme thenApply(), thenAccept(), et thenCompose().
Exception Handling : Gère les exceptions avec exceptionally() et handle().
Combinaison : Permet de combiner plusieurs CompletableFuture avec allOf() et anyOf().

# Exemple CompletableFuture

-

# 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).

# COMMIT

# 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.

# Programmation Fonctionnelle: 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

- Moins de classes, mais code verbeux et peu lisible
- Détournement des principes objet pour passer du comportement
- Idéal : Passer des comportements directement
- Programmation fonctionnelle : Manipuler fonctions comme variables

# Lambdas

- Java 8 : Introduction des lambdas.
- Blocs de code : Représentent des méthodes courtes.
- Paramètres : 0 ou plusieurs.
- Retour : Valeur ou void.

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

- On peut stocker ces méthode dans une variable de type Interface Fonctionnelle.

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

- L'intérêt des lambdas est qu'elles peuvent être utilisées en paramètre de méthodes.
- Utiles en programmation fonctionnelle

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

# Interface fonctionnelle

- Lambdas en Java :
- Stockées dans des variables, mais pas de type ‘méthode’.
- Interface Fonctionnelle :
- Interface avec une seule méthode public abstract.
- Lambdas sont interprétées comme cette méthode.
- @FunctionalInterface :
- Optionnelle, mais aide à détecter des erreurs de compilation.
- Déduction des Types :
- Le compilateur peut déduire les types de retour et des arguments à partir de l'interface fonctionnelle.

# Interfaces fonctionnelles

- Ci-dessous deux interfaces fonctionnelles fournies par Java. Elles font partie des nombreuses interfaces de java.util.function.

```java
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);

// ....

@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
```

# Liste des interfaces fonctionnelles

Function<T, R> : Accepte un argument de type T et retourne un résultat de type R.

BiFunction<T, U, R> : Accepte deux arguments de types T et U, et retourne un résultat de type R.

Predicate<T> : Accepte un argument de type T et retourne un boolean (test une condition).

BiPredicate<T, U> : Accepte deux arguments de types T et U, et retourne un boolean.

Consumer<T> : Accepte un argument de type T et ne retourne rien (effectue une action).

BiConsumer<T, U> : Accepte deux arguments de types T et U et ne retourne rien.

Supplier<T> : Ne prend aucun argument et retourne un résultat de type T.

# Interfaces fonctionnelles

- Il n’est pas interdit de créer soi-même ses propres interfaces fonctionnelles, au besoin.
- Si ces interfaces fonctionnelles sont déjà disponibles dans java.util.function, il vaut mieux les réutiliser.

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

# Lambdas : arguments

- Arguments :
- Placés à gauche du symbole ->.
- Signature :
- Contenus dans des parenthèses, séparés par des virgules.
- Type des Arguments :
- Optionnel (dérivé de l'interface fonctionnelle).
- Un seul Argument :
- Parenthèses optionnelles.

# Lambdas : corps d’une expression

- Position :
- À droite de l'opérateur ->.
- Types de Corps :
- Expression unique
- Bloc de code (une ou plusieurs instructions avec accolades).
- Règles à Respecter :
- Instructions : 0, 1 ou plusieurs.
- Expression unique :
- Pas d'accolades ni return requis.
- Valeur de retour = valeur de l'instruction.
- Plusieurs instructions :
- Accolades obligatoires.

# Lambdas : les variables

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

- Paramètres de l'Expression

  Variables passées en paramètres.

- Variables Locales

  Variables définies dans le corps de l'expression.

- Variables Finales

  Variables déclarées final dans le contexte englobant.

- Variables Effectivement Finales

Non déclarées final, mais leur valeur n'est jamais modifiée.
Introduites dans Java 8 (ex : arguments de la méthode contenant la lambda).

# Lambdas : Exemple

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

```
BiConsumer<String, String> lambdalogger = (String parameter1, String parameter2) -> {
System.out.println(parameter2);
};
lambdalogger.accept("bon","jour");
Function<String,String> stringReturner = (param1)->{return param1;};
System.out.println(stringReturner.apply("Bon Jour"));
```

# 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.

# Exemple: 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 :

```
BiFunction<String, String, String> concatenate = (str1, str2) -> Utils.concat(str1, str2);

BiFunction<String, String, String> concatenate = Utils::concat;
```

# 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.

# COMMIT

# Utilisation des lambdas : les streams

- API Stream : opérations fonctionnelles sur ensembles d'éléments.
- Paramètres : nécessite une interface fonctionnelle.
- Expressions lambdas : utilisation naturelle avec Streams.
- Références de méthodes : intégration dans la définition des Streams.
- Opérations standards : traitements exprimés par lambdas/références.

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

# Streams : généralités

- Types d'opérations :

Intermédiaires : opérations qui renvoient un nouveau Stream et sont exécutées de manière paresseuse (ex. : filter, map).

Terminales : opérations qui terminent le traitement d'un Stream (ex. : collect, forEach, reduce).

- Données infinies : Les Streams peuvent traiter des séquences infinies d'éléments.

Méthode limit(n) : Limite le nombre d'éléments traités à n.

Méthode findFirst() : Retourne le premier élément d'un Stream, même si le Stream est infini.

# Streams : généralités

Streams : traitement d'objets uniquement.

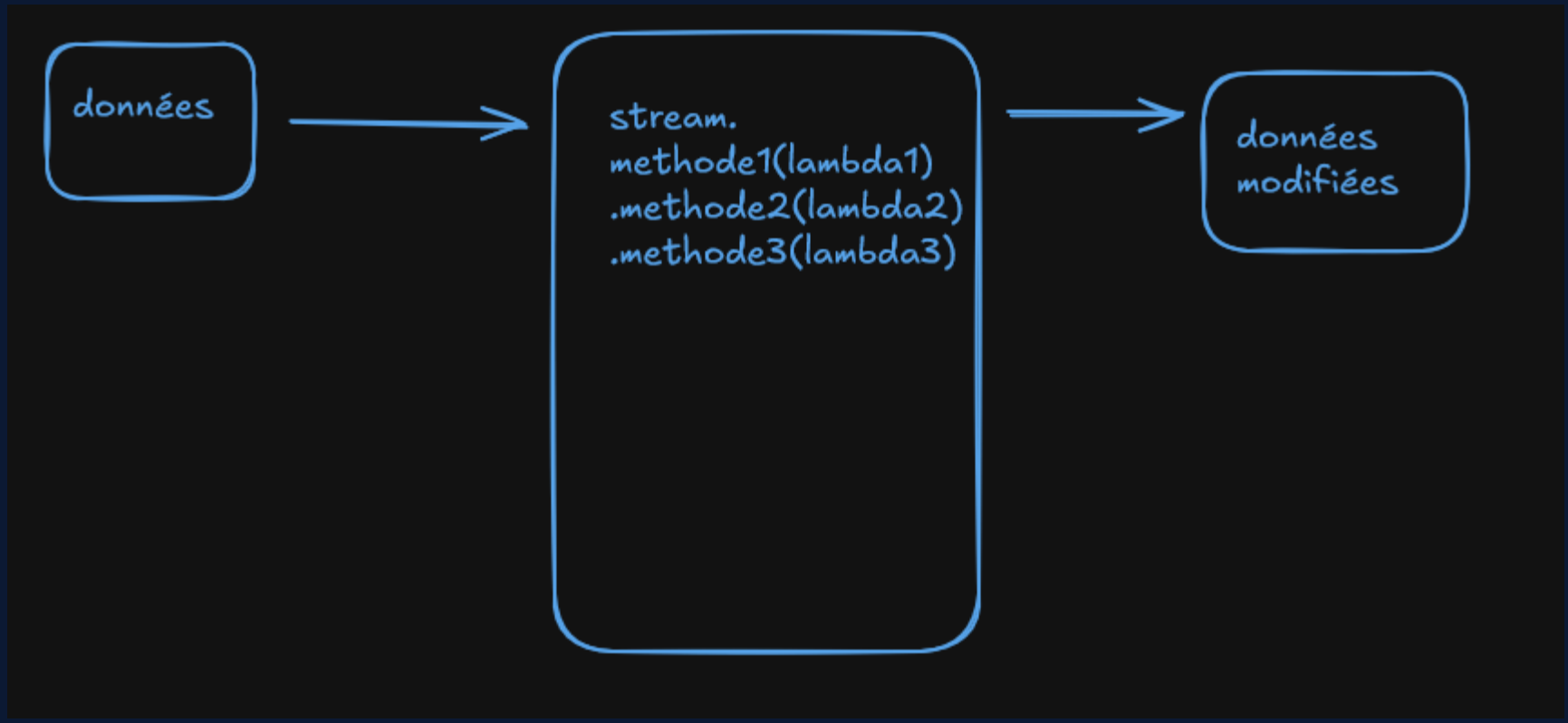
Types primitifs : IntStream, LongStream, DoubleStream.

Pas de stockage : traite et transporte des éléments d'une source à une sortie.

Opérations non-modificatrices : ne changent pas leurs sources.

Comportement paresseux : optimisations (filtrage, mapping, suppression des doublons).

# Streams : généralités



# 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 et collections

- Interface Collection : racine de nombreuses classes (ArrayList, HashSet).
- Méthode stream() : permet d’utiliser les streams avec les collections.
- Maps : non streamables directement.
- Méthodes de Map : utiliser keySet(), values(), entrySet() pour streamer une Map.

# 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 (list, map…)

# Exemples opérations

```
List<User> userList = generateUserList();
List<User> transformedList= userList.stream()
.distinct() //Pour que distinct fonctionne, il faut bien override equal et hashcode
pour les objets.
.map(user->{
user.setId(null);
return user;})
.collect(Collectors.toList());
// Afficher la liste transformée
transformedList.forEach(System.out::println);
// Concaténer les hashCodes des utilisateurs
String listStringified = transformedList.stream()
.map(user -> String.valueOf(user.toString())) // Convertir user en String
.reduce("", (partialString, hashCode) -> partialString + hashCode); // Concaténer les
strings


// Afficher le résultat
System.out.println("Concaténation des strings: " + listStringified);
```

# COMMIT

# Optional

- 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();
}
```

# 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();
```

# 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);
```

# 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().max(Comparator.comparing(c ->
c.getAge())).orElse(defaultChild);
```

# Exemple: Optionnal

```java
//La méthode max renvoie un Optional
List<User> users = Main.generateUserList();
User defaultUser= new User(null,null,null,0);
User older = users.stream().max(Comparator.comparing(c -> c.getAge())).get();

System.out.println(older);

List<User> emptyUserList = new ArrayList<>();
//orElse permet de renvoyer une valeur qui n'est jamais nulle
Optional<User> olderOptional = emptyUserList.stream().max(Comparator.comparing(c ->
c.getAge()));
older = olderOptional.orElse(defaultUser);
System.out.println(older);
```

# COMMIT

# Améliorations des streams dans JDK 9

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).

# COMMIT

# 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.

# 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.

# COMMIT

# 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.

# Les Switchs

# 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.

# switch : nouvelle syntaxe (yield)

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

# COMMIT

# Pattern Matching

- Switch et Pattern matching (preview) : switch évolue 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();
    };
}
```

# Pattern Matching

**Pattern Matching pour Switch**

- Introduction de conditions supplémentaires: when.

```
switch (obj) {
    case Integer i when i > 10 -> System.out.println("C'est un entier supérieur à 10: "
    case Integer i -> System.out.println("C'est un entier: " + i);
    case String s -> System.out.println("C'est une chaîne de caractères: " + s);
    case Double d -> System.out.println("C'est un nombre à virgule flottante: " + d);
    case null -> System.out.println("C'est null");
    default -> System.out.println("Type inconnu");
}
```

https://docs.oracle.com/en/java/javase/21/language/pattern-matching-switch-expressions-and-statements.html#GUID-E69EEA63-E204-41B4-AA7F-D58B26A3B232

Switchs

# COMMIT

# JDK 20: Record Pattern en Java

- Définition : Utilisation des Records dans le Pattern Matching pour décomposer et manipuler les composants des records.

- Utilisation :

Vérifie le type et extrait directement les champs des Records dans un switch ou un if.

Syntaxe simple et claire.

# Exemple Record Matching

```
} public static void printUserInfo(User user) {
    // Switch utilisant Record Pattern
    switch (user) {
        case User(String nom, int age, int id) when age < 30 ->
                System.out.println(nom + " est un jeune adulte de " + age + " ans avec
                l'ID " + id);
        case User(String nom, int age, int id) when age >= 30 && age < 40 ->
                System.out.println(nom + " est un adulte de " + age + " ans avec l'ID
                " + id);
        case User(String nom, int age, int id) when age >= 40 ->
                System.out.println(nom + " est un senior de " + age + " ans avec l'ID
                " + id);
    default ->
        System.out.println("Utilisateur non reconnu");
    }
}
```

JDK 21

# Exercice : Pattern Matching et Record Pattern

Implémentez une méthode describeObject(Object obj) qui analyse dynamiquement un objet du CRM pour en déterminer le type et ses caractéristiques à l'aide du Pattern Matching et des Record Patterns.

Types à gérer :

- User (nom, prenom, age, id)
- Product (nom, prix, stock)
- Order (id, utilisateur, montantTotal)
- Autres : String, Integer, ou tout autre objet non spécifié

Conditions supplémentaires :

- Integer :
  - Si positif → "Identifiant positif"
  - Si négatif → "Identifiant négatif"
- String :
  - Plus de 10 caractères → "Nom long"
  - Moins ou égal à 10 → "Nom court"

# COMMIT

# 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.

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(2) 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.

# Commit

# Temporal Adjuster

- TemporalAdjusters : Outils pour modifier des objets Temporal (ex. LocalDate, LocalDateTime).
- Méthode with() :

  Accepte un TemporalAdjuster.
  Applique un ajustement sur la date/heure

```
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 : Interface utilisée pour obtenir des résultats à partir d'objets Temporal (ex. LocalDate, LocalDateTime).

- Méthode query() :

  Permet d'extraire des données spécifiques d'un objet Temporal.
  Accepte une instance de TemporalQuery pour définir le type de résultat souhaité.

- Utilisation :

Pratique pour obtenir des informations comme le jour de la semaine, le mois, ou d'autres propriétés.

```
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.

# Commit

DateTime

# Period & Duration

Period

Représente : Années, mois, jours
Utilisation : Manipuler des dates
Exemple : Period.of(1, 2, 15) (1 an, 2 mois, 15 jours)

Duration

Représente : Heures, minutes, secondes
Utilisation : Manipuler des heures
Exemple : Duration.ofHours(5) (5 heures)

```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);
```

- 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.

# Commit

# 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.

# JShell

# Introduction

- REPL : Évaluer du code Java ligne par ligne, sans nécessiter un fichier source complet.
- Test rapide : Idéal pour tester des expressions, des algorithmes ou des méthodes individuelles.
- Apprentissage interactif : Outil puissant pour apprendre et expérimenter Java en temps réel.
- Feedback instantané : Résultats visibles immédiatement après l’exécution d'une commande.
- Complétion automatique : Propose des suggestions pour les méthodes, classes, etc.
- Commandes spécifiques : (/vars, /methods, /list, /help) pour gérer les définitions de variables, méthodes et exécuter des commandes.

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.

```
jshell> int factorielle(int n) {
   ...>     if (n == 0 || n == 1) {
   ...>         return 1;
   ...>     } else {
   ...>         return n * factorielle(n - 1);
   ...>     }
   ...> }
```

# Commit

# 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 ...)

# API Process

# Introduction

A quoi sert l'Api process?

- Lancer : Démarrer des processus.
- Gérer : Surveiller et contrôler les processus.
- Rediriger : Manipuler leurs sorties.

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 ). ( avec .destroy() )

# 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.

```
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.

# Commit

# 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

# HTTP2 > HttpURLConnection

Limitation de protocole

- HttpURLConnection supporte uniquement HTTP/1.1, limitant l'envoi d'une seule requête par connexion.

Performance améliorée avec HTTP/2.0

- HTTP/2.0 permet d'envoyer plusieurs requêtes simultanément sur une seule connexion, améliorant ainsi les performances.

Mode synchrone

- HttpURLConnection fonctionne en mode synchrone, ce qui peut entraîner des blocages pendant l'attente de réponses.

## JDK 9 : autres nouveautés

# HTTP 2

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}
```

# 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");
    }
}
```

# Commit

# 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();
```

# 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.

# 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

- A contrario, passer tous les types en var peut plus gêner le développeur 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

# Commit

# Autres améliorations de la JDK 10

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

# 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.

# Commit

# 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).

# Commit

# Classes internes

Définition : Classes définies à l'intérieur d'une autre classe.

Types :

- Classes internes non statiques : Accès aux membres de la classe externe.
- Classes internes statiques : Ne peuvent pas accéder directement aux membres non statiques de la classe externe.

Utilisation :

- Encapsulation : Regroupe des fonctionnalités proches dans une seule structure.
- Lisibilité : Améliore l'organisation et la lisibilité du code.

# 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

# 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());
```

# 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";
```

# 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 ....

# 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>
```

# Commit

# 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

# Clarifications sur la NullPointerException

La clarification du NullPointerException dans JDK 14 consiste à fournir des messages d'erreur plus explicites qui indiquent la cause précise de l'exception, facilitant ainsi le débogage en précisant quel objet était null lors de l'accès à ses méthodes ou propriétés.

JDK 14

# Live Monitoring

Qu'est-ce que c'est ?

Outil permettant de surveiller les applications Java en temps réel.

Principales fonctionnalités :

- Collecte de données dynamiques : Surveille les performances et l'état des applications sans nécessiter de redémarrage.
- Visibilité accrue : Permet d'analyser les métriques en temps réel pour une meilleure compréhension du comportement de l'application.

Outils intégrés :

- JFR (Java Flight Recorder) : Enregistre des événements et des performances à l'exécution.
- JMC (Java Mission Control) : Outil d'analyse pour examiner les données de JFR et optimiser les performances.

Avantages :

- Détection rapide des problèmes : Facilite l'identification et la résolution rapide des problèmes de performance.
- Optimisation continue : Permet une amélioration continue des applications basées sur des données en temps réel.

JDK 14

# Nouvel instanceof

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

# Commit

# 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

- Définition : Les classes scellées permettent de contrôler quels autres classes peuvent étendre une classe spécifique.
- Syntaxe : Utilisation du mot-clé sealed, suivi de permits pour spécifier les sous-classes autorisées.
- Utilité :
- Garantir une hiérarchie de classes bien définie.
  Améliorer la sécurité et la lisibilité du code.

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

# 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

JDK 15

# Interfaces scellées

Définition : Les interfaces scellées permettent de restreindre quelles classes peuvent implémenter une interface donnée.

Syntaxe : Utilisation du mot-clé sealed, suivi de permits pour spécifier les classes autorisées.

Utilité :

Offrir une plus grande sécurité dans les contrats d'interface.
Aider à la structuration d'architectures logicielles complexes.

```
sealed interface Animal permits Chien, Chat {}
final class Chien implements Animal {}
final class Chat implements Animal {}
```

# Fonctionnalités dépréciées

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

# JDK 16

JDK 16

# jpackage

Définition : jpackage est un outil inclus dans le JDK (Java Development Kit) à partir de Java 14, qui permet de créer des packages d'applications Java prêts à être distribués et installés sur diverses plateformes (Windows, macOS, Linux).

Fonctionnalités :

Création de Packages : Génère des exécutables natifs (.exe, .app, .deb, .rpm) pour l'application Java.

Inclusion de dépendances : Emballe l'application avec ses bibliothèques et fichiers de ressources nécessaires.

Personnalisation : Permet de définir des propriétés de l'application comme le nom, l'icône, et la version.

# JDK 17 LTS

https://docs.oracle.com/en/java/javase/17/language/index.html

# JDK 18

JDK 18

# Nouveaux Outils et API

## Serveur Web Minimaliste (JEP 408)

- Commande jwebserver : Démarrage d'un serveur HTTP simple en une ligne
- Destiné au développement rapide et au prototypage: téléchargement fichier, API web statique, tests HTML/CSS

```
jwebserver -d /chemin/vers/le/repertoire
```

## Foreign Function & Memory API (JEP 419)

- Accès sécurisé et performant à la mémoire native.
- Facilite les appels à des bibliothèques externes (C/C++).

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

# Exercice

Lancer un jwebserver à la racine du projet nouveautés et télécharger le pom.xml

# Commit

JDK 18

# Performance et Sécurité

API Vector (JEP 417)

Utilisation des instructions SIMD (Single Instruction, Multiple Data)
→ Exécute la même opération sur plusieurs éléments de données en parallèle.

Optimisation des opérations sur les tableaux de données
→ Permet d'accélérer des calculs lourds sur des grandes quantités de données (comme des vecteurs ou des matrices).

Phase d'incubation
→ Encore en développement, mais déjà disponible pour expérimenter des gains de performance sur des calculs parallèles.

# Performance et Sécurité

## BoringSSL

BoringSSL est une bibliothèque SSL développée par Google, dérivée de OpenSSL, conçue pour une utilisation dans des environnements critiques et axée sur la sécurité.

### Performances Optimisées

Réduction de la latence lors de l'établissement de connexions TLS (protocole de sécurité).
Améliorations des algorithmes cryptographiques pour des opérations plus rapides.

### Sécurité Renforcée

Support des dernières versions de TLS (1.3) pour une sécurité améliorée.
Suppression des fonctionnalités obsolètes et vulnérables (ex : SSLv3, RC4).

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

# JDK 19

# Exemple: Virtual Threads

Virtual Threads : Threads légers pour le modèle de concurrence

- Nouvelle API pour gérer des milliers de threads légers de manière plus efficace.
- Simplifie la gestion des tâches concurrentes sans changer le modèle de programmation actuel (pas besoin de réécrire les applications).
- Meilleures performances pour les applications intensives en I/O ou réseau.

Avantages :

- Réduction de la surcharge liée aux threads natifs.
- Améliore la scalabilité des applications modernes.

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

# Commit

# Améliorations de la Mémoire et de la Sécurité

## Foreign Function & Memory API

- Évolution de l'API pour accéder aux fonctions natives et gérer la mémoire externe de manière plus performante.
- Permet de travailler directement avec des structures de données externes tout en minimisant les erreurs de gestion de mémoire.
- Simplifie l'interopérabilité avec des bibliothèques C/C++.

Améliorations de la Sécurité

- Désactivation progressive de l'API de sécurité Security Manager, jugée obsolète.
- Focus sur l'utilisation de nouvelles approches de sécurité plus robustes et modernes: conteneurisation(Docker, Kubernetes), virtualisation (Firecracker, Gvisor), utilisation de frameworks (Spring Securit, JEE Security)

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

# JDK 20

# Améliorations de la Performance et des Outils

**Améliorations de la Performance de la JVM**

- Optimisations du compilateur, en particulier avec GraalVM.
- Améliorations de l'exécution des programmes, permettant des gains de performance pour les applications Java.

**API pour les Éditeurs et Outils de Développement (JEP 432)**

- Introduction de nouveaux outils pour améliorer l'expérience des développeurs.
- Facilite la création de bibliothèques et d'outils qui tirent parti des nouvelles fonctionnalités du langage.

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

# Préparation pour les Futures Évolutions

**Incubation de la fonction Virtual Threads**

- Les threads virtuels sont encore en incubation, permettant une gestion plus légère des tâches concurrentes.
- Favorise l'écriture d'applications hautement concurrentes avec une meilleure gestion des ressources.

**API de Contrôle des Dépendances**

- Améliore la gestion des dépendances dans les projets Java.
- Simplifie l'intégration et la résolution des dépendances pour les applications Java modernes.

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

# JDK 21

# Améliorations de la Performance et de l'Écosystème

**Améliorations des Threads Virtuels**

- Les threads virtuels continuent d'être affinés pour améliorer la gestion de la concurrence.
- Permet aux développeurs d'écrire des applications hautement concurrentes de manière plus simple et plus efficace.

**API de Contrôle de l'Accès aux Fichiers**

- Introduit une API améliorée pour contrôler l'accès aux fichiers, facilitant la gestion des permissions de sécurité.
- Renforce la sécurité des applications en offrant un meilleur contrôle des accès.

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

# Prise en charge des Normes Modernes et des Outils

**Améliorations des Annotations**

- Les annotations peuvent désormais être utilisées avec des expressions lambda, améliorant la lisibilité et la gestion des métadonnées dans le code.

**Mises à jour de l'API de Collecteurs**

- Ajout de nouveaux collecteurs pour les flux, permettant des opérations de regroupement plus puissantes et flexibles.

**Améliorations de l'API de Réflexion**

- Optimisation des performances et simplification de l'utilisation de l'API de réflexion pour les applications modernes.

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

# Dossier pédagogique

- Feuilles d’émargement signées pour chaque journée.
- Feuilles d’émargement signées pour les passages de certifications (si certifications)
- Évaluations formateur

# Questions / réponses

- Revenons sur les questions hors plan de cours que vous m’avez posé durant la formation pour y répondre

# Formation suivante

- **Java - Programmation avancée**

  - Approfondir les traitements parallèles, le multi-thread, les promesses…
  - Approfondir vos connaissances sur le Classloader et la JVM
  - Test Driven Development appliqué à Java avec Junit
  - Optimisation: outils de mesure et d’analyses pour améliorer le code
  - Gestion mémoire

  https://www.m2iformation.fr/formation-java-programmation-avancee/JAV-AV/

# Merci d’avoir suivi cette formation M2I et à très bientôt !

# Bilan formation et remerciements

- Merci d’avoir participé à cette formation M2I.
- Envoie du Bilan formation.

# Votre formateur

Victor, Dauphin
Formateur externe M2I
Victor.dauph@gmail.com
https://www.linkedin.com/in/victor-dauphin-83430840/

Et encore merci !