Les Bases de la Programmation en Langage Java Concepts Théoriques et Applications Pratiques

Moussa Keita

Les Bases de la Programmation en Langage Java

Concepts Théoriques et Applications Pratiques

Moussa KEITA
Développeur Big Data
Consultant à la Caisse des Dépôts, Société Générale et EDF

Paris

Document version 1.0

(Juin 2023)

Contact Email : keitam09@ymail.com

TABLE DES MATIERES

1 INTRODUCTION

1.1 Le langage Java : ses origines et ses particularités

Java est un langage de programmation servant de base au développement de nombreux logiciels et applications informatiques ainsi que divers outils technologiques de traitement de l'information.

Le projet de création du langage Java a été initié en 1991 au sein Sun Microsystems par une équipe d'ingénieurs dirigée par James Gosling. Le projet visait, au départ, à proposer un langage permettant d'écrire (sous forme de codes embarqués) des programmes pour faire communiquer des appareils électroniques: téléviseurs, télécommandes, décodeurs, petits appareils électriques, etc. La première version du langage a été proposée en 1995. Même si le projet initial n'a pas connu le succès escompté, le langage s'est tout de même révélé adapté à de nombreux autres domaines d'utilisation en particulier le domaine de la programmation Web. Par exemple, il a servi de base au développement du navigateur HotJava, un autre projet de Sun Microsystems. Il a également été intégré au projet du navigateur Netscape. Cette adaptabilité du langage a grandement contribué à son essor et à sa popularité au fil des années. Depuis le rachat de Sun Microsystems par Oracle en 2009, la langage Java est devenu la propriété de Oracle qui assure désormais sa maintenance. Les versions successives du langage sont consultables sur ce lien.

Grâce à une bibliothèque très dense et des fonctionnalités riches et variées, Java est devenu, aujourd'hui, un langage incontournable dans le domaine de l'industrie informatique, du génie logiciel, des technologies de l'information et de la communication mais également dans le domaine de traitement de données en particulier dans le domaine du Big Data.

L'une des spécificités du langage Java est sa portabilité. En effet un même code source peut être porté et exécuté dans n'importe quel environnement disposant d'une *Machine Virtuelle Java* (JVM) sans aucune autre contrainte particulière. Java utilise ainsi le concept de Machine Virtuelle qui était déjà utilisé par le langage Pascal UCSD crée en 1977. La portabilité du code Java est assurée par le fait que le code source est d'abord compilé en un format spécifique appelé *bytecode* qui est exécutable sur tout environnement disposant d'un interpréteur Java. Le rôle d'interpréteur est joué par la JVM notamment sa composante JRE (Java Runtime Environnent). La compilation du code source en bytecode et la disponibilité de l'interpréteur rend le langage Java agnostique à l'environnement d'exécution (Système d'exploitation hôte).

Une autre particularité du langage Java est son formalisme Orienté-Objet. En effet, le projet Java s'est inscrit, dès sa création, dans l'approche de Programmation Orientée-Objet (POO)[1] initiée en 1970 par le langage SmallTalk. A noter, qu'à cette époque, la Programmation Procédurale (PP) était l'approche de programmation dominante, utilisée par de nombreux langages comme par exemple C++ (langage crée en 1985). Bien que Java

[1] Voir ci-dessous les détails sur la Programmation Orientée-Objet.

ait suivi le formalisme Orientée-Objet, sa syntaxe d'écriture est tout de même restée très proche de celle du langage C++.

1.2 La Programmation Orientée-Objet

1.2.1 Le concept de Programmation Orientée-Objet (POO)

Java adopte une démarche de programmation dite Programmation Orientée-Objet (POO) à la différence de l'approche classique dite Programmation Procédurale (PP). Rappelons que l'approche PP est une approche qui est exclusivement centrée sur l'écriture de fonctions (encore appelées procédures) qui visent à traiter les données prises en entrée du programme. Dans une approche PP, le programme est structuré autour d'un ensemble de procédures (fonctions) et d'un ensemble de données qui vivent indépendamment des procédures.

A la différence de la PP, la POO est une démarche où les procédures d'accès ou de traitement (désormais appelées méthodes) sont appliquées aux données qui sont préalablement "*encapsulées*" dans des enveloppes appelées *Objets*. Un objet est une entité qui regroupe un ensemble de données et aussi un ensemble de méthodes permettant d'accéder à ces données ou de les traiter. L'encapsulation des données dans un objet implique que pour agir sur les données, il faut nécessairement passer par les méthodes associées à l'objet. Les méthodes jouent donc le rôle d'intermédiaire entre l'utilisateur et les données. En POO, le programmeur définit d'abord les objets de telle sorte que chaque objet représente une entité bien identifiée du système. Ensuite, il écrit les fonctions d'accès et de traitement associé à chaque objet de sorte à pouvoir accéder aux données et les traiter.

En somme, la particularité de l'approche POO par rapport à d'autres approches est qu'elle permet de regrouper dans un même objet les données et les traitements qui s'y appliquent.

1.2.2 Les concepts d'entités, de Classe et d'Objet

1.2.2.1 Identification des entités

La mise en place, suivant l'approche POO, d'un programme en tant solution à une problématique donnée dans un système nécessite d'abord de modéliser ce système et d'identifier clairement toutes les entités pertinentes qui le composent. La programmation consistera alors simplement à matérialiser par du code informatique chaque entité, ses différents états, ses liens et ses interactions avec les autres entités du système. Par exemple, dans une entreprise commerciale, les entités pertinentes peuvent être les employés, les clients, les commandes, les opérations de caisse. Dans un établissement scolaire, les entités pertinentes peuvent être les classes, les élèves, les professeurs, les matières, les examens, les notes, etc.. La pertinence d'une entité est évaluée en fonction de son apport dans la construction de la solution proposée par le programme à écrire. Par exemple, pour écrire un programme de gestion de prêts de livres dans une bibliothèque universitaire, les entités

pertinentes peuvent être les étudiants, les professeurs et les classes. L'entité « terrain de sport », même si elle fait partie du système du campus universitaire, ne peut pas être considérée comme pertinente dans le cas présent.

Notons aussi que les entités ne sont pas nécessairement toutes indépendantes. Une entité peut être une partie intégrante d'une autre entité. Par exemple, pour une entreprise commerciale, l'entité commande client est dépendante aussi bien de l'entité client que de l'entité produit. Car une commande est un fait évènementiel qui met en rapport un client avec un ou plusieurs produits. Elle fait également intervenir une entité employé (ex : préparateur de commande, caissiers, etc..) Ainsi, lorsque la commande client est considérée comme une entité à part entière dans le système, elle conservera donc une dépendance avec les entités client, produits et employé.

1.2.2.2 Distinction entre Classe-Objet

Dans la Programmation Orientée-Objet, une entité représente une classe. La classe est donc une notion plus conceptuelle et plus générale. Par exemple, reprenant l'exemple de l'entreprise commerciale, l'entité CLIENT forme la classe *Client* ; l'entité PRODUIT forme la classe *Produit*. La classe est une structure générique. L'objet en est sa matérialisation concrète par des données. Par exemple, si l'entité PRODUIT est une classe, une bouteille de lait qui en est une matérialisation de cette classe est considérée comme un objet. De même, si l'entité ELEVE est une classe, Juliette, élève en classe de 3^{ième} est un objet de cette classe. Et plus encore, si ANIMAL est une classe, alors chien, chat et lion qui en sont des matérialisations peuvent être considérés comme des objets de cette classe.

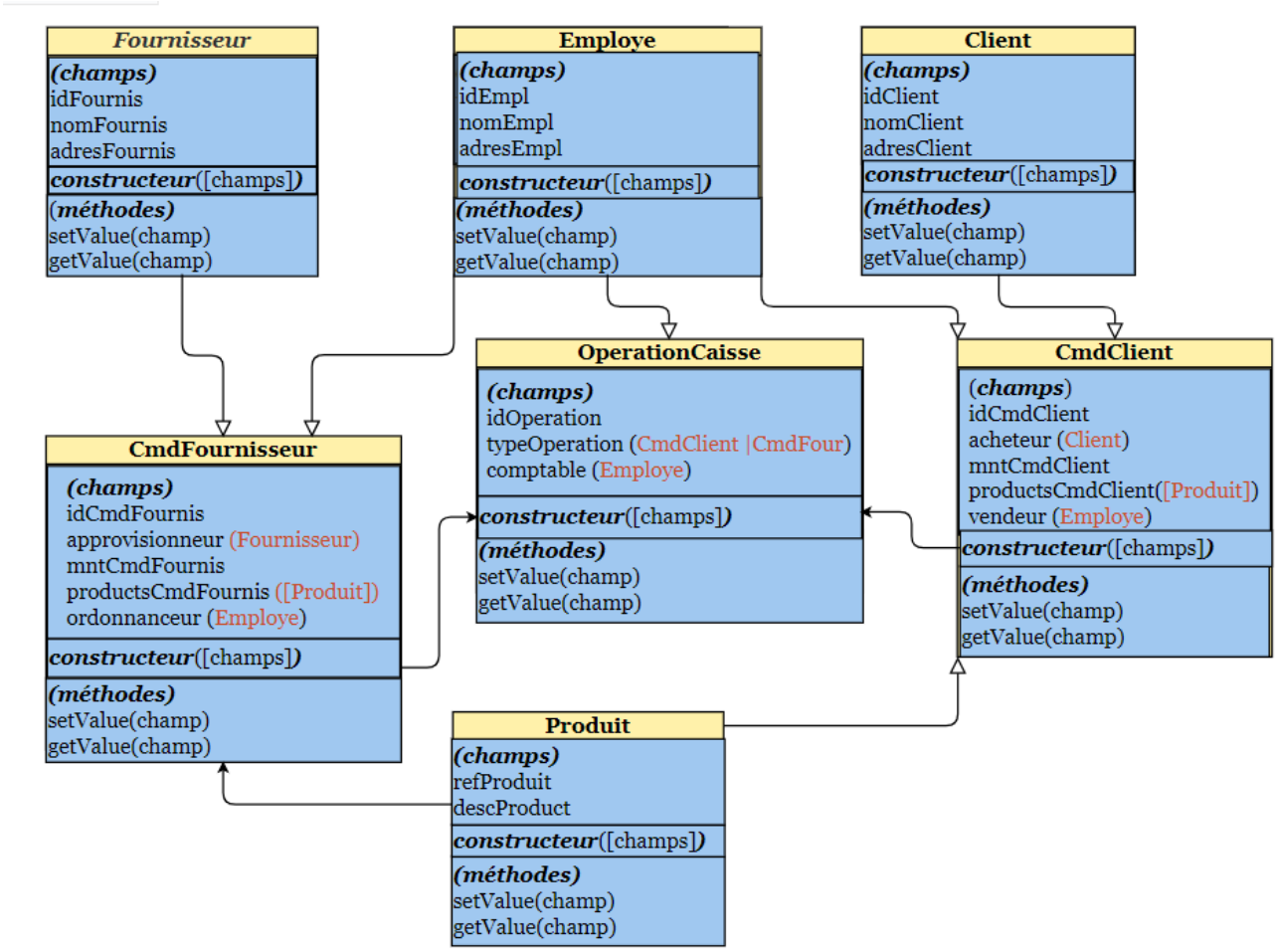
Pour résumer la différence entre classe et objet, procédons à quelques analogies. Par exemple, en prenant le cas d'une base de données relationnelle (BDR), une table représente une classe, car elle définit la structure générale des données. Chaque ligne dans cette table représente un objet, car chaque ligne de données matérialise de manière concrète la structure générale définie par la table. Prenons l'exemple d'un constructeur automobile comme Citroën. Le modèle Citroën C3 est une classe car il correspond à une spécification générale et générique. Mais la voiture de Citroën C3 dans laquelle je roule est un objet car elle correspond à la matérialisation de la spécification du modèle générique C3.

1.2.3 Illustration concrète de la notion de classe

Pour mieux illustrer la notion de classe, prenons le cas d'une entreprise commerciale qui nous sollicite pour développer une application de gestion de ses activités. Notre première tâche consisterait d'abord à répertorier l'ensemble des entités pertinentes entrant dans le cadre des activités de cette entreprise. Pour des besoins d'illustration, nous avons supposé que les entités pertinentes de cette entreprise sont les employés (entité EMPLOYE), les fournisseurs (entité FOURNISSEUR), les clients (CLIENT), les produits (PRODUIT), les commandes-fournisseurs(CMD_FOURNISSEUR), les commandes-clients (CMD_CLIENTS), la trésorerie (OPERATION_CAISSE), etc. Dans une approche Orientée-

Objet, chacune de ces entités est représentée par une Classe. Par exemple, l'entité EMPLOYE est représentée par la classe *Employe*, l'entité FOURNISSEUR est représentée par la classe *Fournisseur*, l'entité CLIENT par la classe *Client*, les commandes-fournisseurs représentées par la classe *CmdFournisseur*, les commandes-clients représentées par la classe *CmdClient*, l'entité PRODUIT représentée par la classe *Produit* et la trésorerie représentée par la classe *OperationCaisse*. La représentation de toute ces entités sous forme de classes ainsi que leur interaction sont présentées sur la figure ci-dessous.

Figure 1 Représentation des entités d'une entreprise commerciale sous forme de classes et leurs interactions



Une classe est une structure générale permettant de représenter les propriétés et les comportements des entités d'un système. Une classe ne contient pas de données, elle se limite à définir les structures qui serviront à accueillir les données en vue de matérialiser les objets. Une classe est caractérisée par quatre types d'information : son nom, ses champs, ses constructeurs et ses méthodes.

- **Nom de la classe**

Chaque classe est nommée de sorte à pouvoir identifier de manière unique dans le système l'entité qu'elle représente. Ex : la classe correspondant à l'entité CLIENT est nommée *Client*. Celle correspondant à l'entité FOURNISSEUR est nommée *Fournisseur*. La classe

correspondant aux commandes client est nommée *CmdClient*. A noter que deux classes ne peuvent pas avoir le même nom dans un système. Aussi, une remarque doit être faite au sujet du nommage des classes en Java. D'abord, les noms des classes commencent toujours par une lettre majuscule. Ensuite, lorsque le nom est composé de plusieurs mots, le premier caractère de chaque mot doit être écrit en lettre majuscule. Exemples : *CdmClient, CmdFournisseur, OperationCaisse*.

- **Les champs (fields)**

Les champs (fields) traduisent les propriétés qui caractérisent les entités. Ils servent à accueillir et à stocker les données pour représenter chaque objet. Par exemple, pour la classe ***Client***, les champs retenus sont *IdClient*, *nomClient*, *adresClient*. Le choix des champs à retenir dépend des besoins du programme pour résoudre le problème posé. Par exemple, pour un programme de gestion de campagnes de marketing, il est possible qu'on retienne les champs comme *ageClient*, *sexeClient*, etc…

Les champs sont plus couramment de types standards (nombres, chaînes de caractères ou alphanumériques). Mais les champs peuvent également être des objets, c'est-à-dire la matérialisation d'autres classes présents dans le système. Cela arrive surtout lorsqu'il y a une dépendance entre les entités ; ce qui est le plus souvent dans un système. Par exemple, dans la figure 1, dans la classe CmdClient, le champs *acheteur* est un objet de la classe Client car ce champ sert à représenter le client qui a passé la commande. De même, le champ vendeur est un objet de la classe *Employe* car il représente l'employé qui a réalisé la vente.

Par ailleurs, à la différence des noms de classes qui commencent toujours par une lettre majuscule, les noms des champs commencent par une lettre minuscule. Et lorsque le nom du champ est composé de plusieurs mots, la première lettre de chaque mot doit être écrit en lettre majuscule (à l'exception du premier mot). Exemples : *nomClient*, productsCmdClient, etc.

- **Le constructeur**

Le constructeur d'une classe est une fonction spéciale qui permet d'initialiser les valeurs des champs et de créer un nouvel objet de cette classe. Chaque fois qu'on appelle le constructeur d'une classe, on dit qu'on instancie la classe, c'est-à-dire qu'on crée un nouvel objet de cette classe. L'instanciation de la classe consiste en fait à attribuer des valeurs initiales aux champs et à les encapsuler dans un objet. Et cet objet représente une instance de la classe, c'est-à-dire une copie concrète de la classe qui encapsule les données réelles. En théorie lorsque les données sont encapsulées dans un objet, il n'est plus possible pour un utilisateur d'accéder directement à ces données, ni de les modifier. Il doit passer par des fonctions dédiées appelées méthodes (voir ci-dessous).

- **Les méthodes**

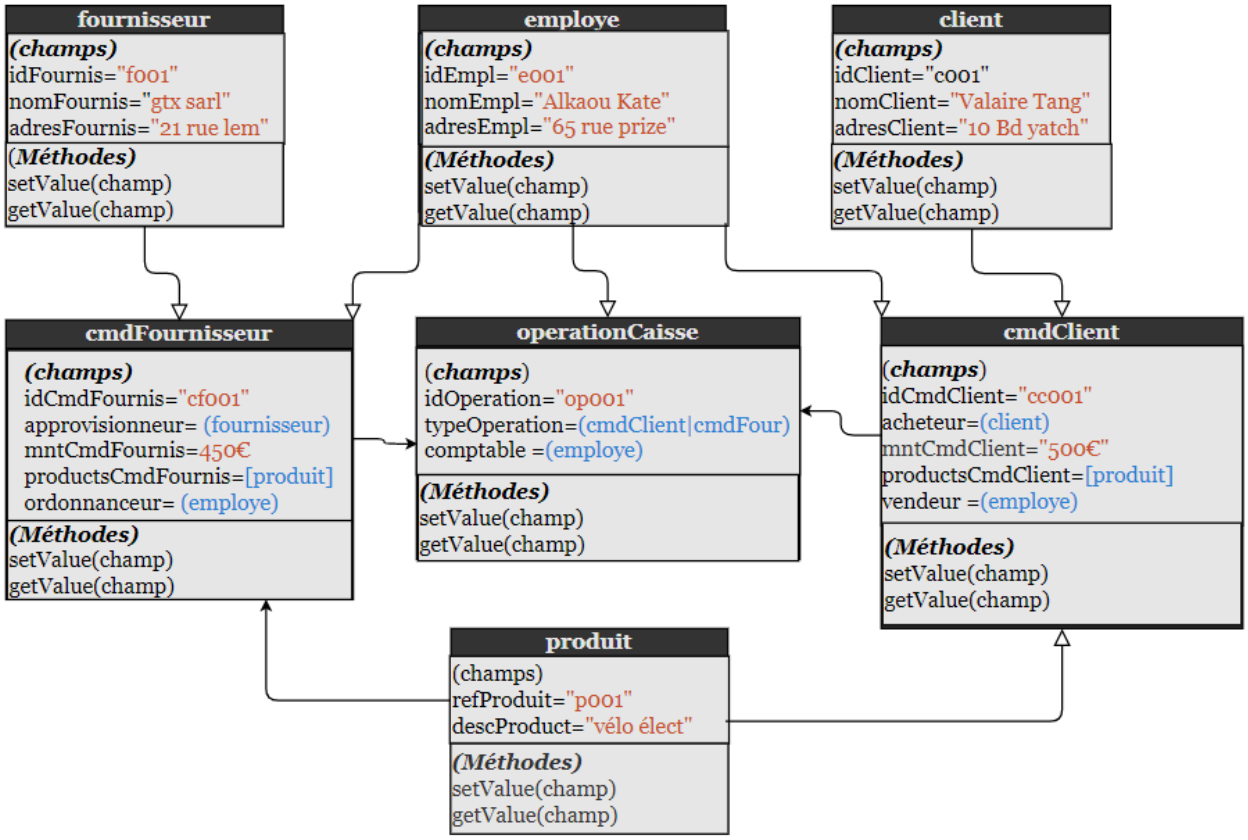
Les méthodes sont des fonctions prévues dans la classe pour permettre à l'utilisateur d'accéder aux données encapsulées dans l'objet instancié et de les modifier si nécessaire. Contrairement à la programmation procédurale, dans la POO, aucun champ n'est défini en

dehors des classes. Et pour agir sur un champ spécifique, il faut nécessairement passer par une méthode. On distingue deux catégories de méthodes : les méthodes dites *getters* et les méthodes dites *setters*. Un getter est une méthode permettant d'accéder à la valeur d'un champ dans un objet. Tandis qu'un setter est une méthode permettant de modifier la valeur d'un champ dans l'objet. Au moins un setter et un getter sont prévus pour chaque champ dans une classe. C'est pourquoi dans la figure 1, pour chaque classe, nous avons deux templates de méthodes : setValue(champ) qui représente le setter pour le champ considéré et getValue(champ) qui représente le getter pour le champ considéré.

1.2.4 Illustration concrète de la notion d'objet

Comme indiqué précédemment, un objet est une instanciation d'une classe c'est-à-dire une matérialisation d'une classe en attribuant des valeurs aux champs qui caractérisent la classe. Les objets sont construits suite à l'appel des constructeurs de classes prévus à cet effet. La figure 2 ci-dessous montre quelques exemples d'objets pouvant être instanciés à partir des classes qui ont présentées dans la Figure 1 (voir section précédente).

Figure 2 : Représentation des objets construits à partir des classes



Comme, on peut le constater sur la figure 2, les objets sont des représentations concrètes des entités. Contrairement à la classe qui est une représentation générale de l'entité, l'objet représente un élément bien identifié de cette entité. Par exemple, le client référencé *c001* nommé *Valaire Tang* et résidant à l'adresse *10 Bd Yatch* est une représentation concrète

de l'entité CLIENT. De même le produit référencé *p001* avec la dénomination *vélo électrique* est une instance de l'entité PRODUIT.

La figure 2 permet de faire quelques remarques sur les objets par rapport aux classes.

D'abord contrairement aux noms de classes, les noms des objets commencent toujours par une lettre minuscule. Et lorsque le nom de la méthode est un nom composé de plusieurs mots, la première lettre de chaque mot doit être écrite en lettre majuscule (à l'exception du premier mot du nom). Exemples : *cmdClient*, operationCaisse, etc.

Enfin, il est important de rappeler le principe suivant. Lorsque le champ d'une classe est de type Objet, pour créer un objet de cette classe et définir la valeur du champ de type objet, il faut d'abord créer une instance de la classe correspondant à l'Objet. Pour être plus précis, prenons le cas de la classe *CmdClient* (voir Figure 1). Cette classe contient un champ nommé *acheteur* qui représente le client qui a passé la commande ; le client étant par ailleurs représenté par la classe *Client*. Pour pouvoir instancier la classe *CmdClient* et définir tous ses champs, il faut au préalable instancier la classe Client. L'instance (l'objet) ainsi obtenue pourra être passée au constructeur de la classe *CmdClient* pour enfin définir le champ *acheteur*. Ce principe s'applique à tous les champs de type objet comme le champ vendeur qui est de type *Empoye* et le champ *productsCmdClient* qui est un champ de type liste de *Produit*.

2 PRÉPARATION DE L'ENVIRONNEMENT DE PROGRAMMATION EN JAVA

Cette section vise à décrire les procédures pour mettre en place les outils nécessaires pour commencer le développement Java. Il s'agit en particulier de la procédure d'installation du Kit de Developpement Java (JDK) mais aussi de la procédure d'installations des Environnements de Développement Integrés (IDEs).

2.1 Installation du Kit de Développement Java (JDK)

2.1.1 Présentation du Java Developement Kit (JDK) et du Java Runtime Environment (JRE)

Le Java Development Kit (JDK) est une infrastructure logicielle qui regroupe un ensemble de bibliothèques et d'outils permettant de compiler et d'exécuter des codes écrit en langage Java. Le JDK fournit un ensemble de composants permettant le débogage de codes, la gestion des ressources mémoire et le monitoring de performance des exécutions. L'installation du JDK sur l'environnement de travail est donc prérequis important pour pouvoir faire du développement Java.

Rappelons tout de même que lorsque l'environnement est destiné uniquement à exécuter du code Java mais pas à développer, ni à compiler, l'installation du JDK n'est pas obligatoire. En effet, lorsque votre code est déjà compilé en bytecode, vous avez simplement besoin de l'infrastructure d'exécution appelé *Java Runtime Environment* (JRE). Le JRE est une version allégée du JDK qui implémente seulement le *Java Virtual Machine* (JVM) en y ajoutant les utilitaires nécessaires à l'exécution du code Java. Bien entendu, le JRE peut être installé en dehors du JDK surtout lorsque l'on n'a pas besoin de développer du code Java sur notre environnement. Mais le JDK est plus complet car il embarque aussi le JRE. La disponibilité du JDK (ou du JRE) permet de garantir la portabilité et l'interopérabilité du code Java.

2.1.2 Procédure d'installation du JDK

2.1.2.1 Installation sur Windows

- Rendez-vous sur la page : https://www.oracle.com/java/technologies/downloads/
- Choisir la version de Java que vous voulez installer. Par exemple, choisir Java 20.
- Cliquer sur l'onglet correspondant à Microsoft Windows[2]

[2] NB : Cette procédure d'installation a été testée uniquement sur le système d'exploitation Microsoft Windows 11.

Pour l'installation sur les systèmes Linux, consulter la page :

- Choisir soit le fichier *x64 Installer* : https://download.oracle.com/java/20/latest/jdk-20_windows-x64_bin.exe
- ou le fichier *x64 MSI Installer* : https://download.oracle.com/java/20/latest/jdk-20_windows-x64_bin.msi

Nous avons choisi le fichier *x64 MSI Installer* car il est directement compatible avec l'installateur de package MSI propre à Microsoft Windows.

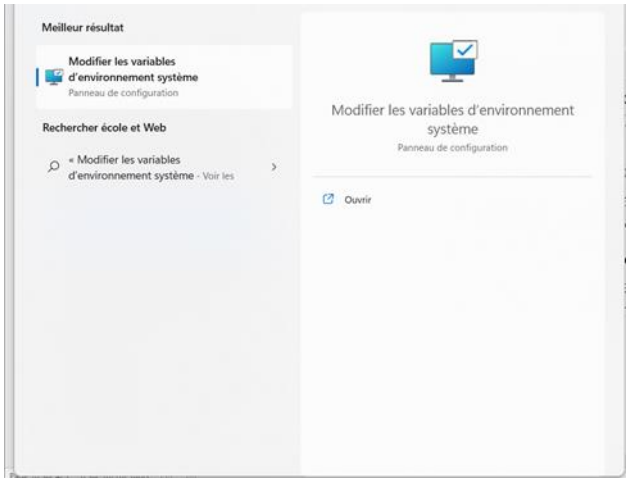
- Cliquer sur l'url du fichier choisi pour télécharger le fichier.
- Après le téléchargement, lancer l'installation de Java en double-cliquant sur le fichier.
- Choisir votre répertoire d'installation. Ex : *C:\Program Files\Java\jdk-20*.
- Poursuivre l'installation.

A la fin, vérifier bien que le dossier *jdk-20* est bien créé dans le répertoire d'installation que vous avez choisi. Ex : *C:\Program Files\Java\jdk-20* est bien créé.

2.1.2.2 Définir la variable d'environnement pour le Java installé

Pour pouvoir pointer sur la version Java installée lors de l'exécution des commandes shell, nous allons d'abord définir la variable d'environnement. Pour cela :

- Ouvrir la barre de recherche depuis le menu Démarrer de votre PC
- Copier et coller dans la barre de recherche la phrase : « Modifier les variables d'environnement système ».



- Cliquer sur « Modifier les variables d'environnement système » pour ouvrir la fenêtre du Système.
- Dans la petite fenêtre qui apparaît, cliquer « Variables d'environnement ».

---

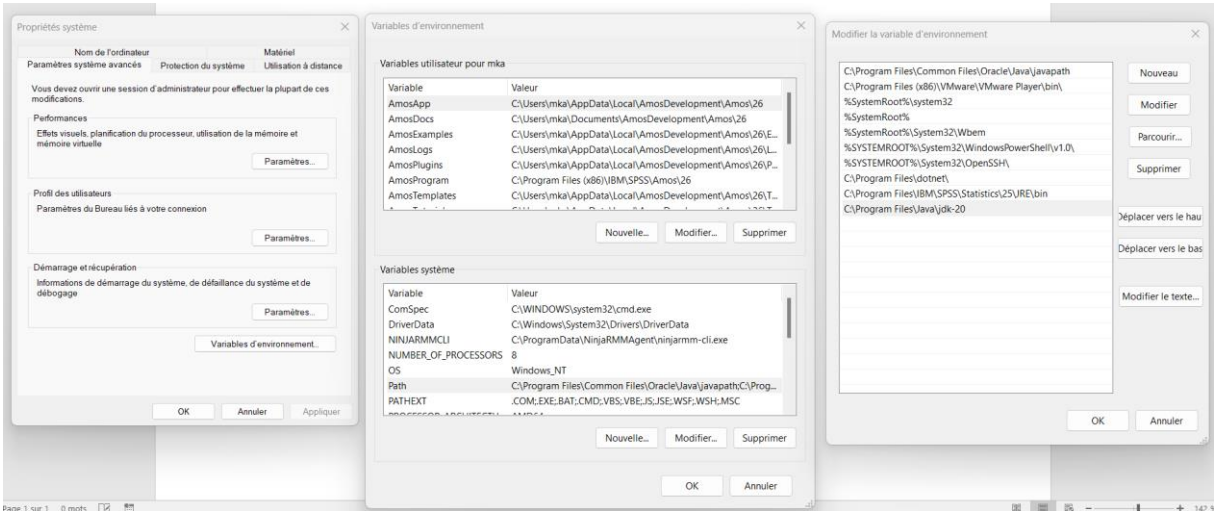
https://docs.oracle.com/en/java/javase/20/install/installation-jdk-linux-platforms.html

Et pour l'installation sur le système Mac Os, consulter la page :

https://docs.oracle.com/en/java/javase/20/install/installation-jdk-macos.html

- Une deuxième fenêtre apparaît. Au niveau du champ « Variables système », sélectionner Path, et cliquer sur modifier. Dans la fenêtre qui apparaît, cliquer sur nouveau et ajouter le lien vers votre installation Java : C:\Program Files\Java\jdk-20.

NB : Assurez-vous qu'une autre installation Java n'est pas renseignée dans ce Path au risque de créer un conflit de version. Si une autre version Java est renseignée dans ce Path, il serait judicieux de la supprimer sauf si cette version est utilisée spécifiquement utilisée par une application bien identifiée.



- Terminer la définition de la variable d'environnement
- Clique sur OK successivement sur toutes les fenêtres ouvertes.

Il est fortement conseillé de redémarrer le PC pour que l'installation soit bien prise en compte, notamment la définition de la nouvelle variable d'environnement Java.

2.1.2.3 Vérifier l'installation

- Dans le menu Démarrer, dans la barre de recherche taper CMD et taper Entrée.

La fenêtre de l'invite Commande Windows Apparaît.

- Tester la version de Java en tapant la commande shell :

java –version

Si la variable d'environnement est bien définie, cette commande devrait renvoyer le résultat suivant :

```
java version "20"
Java(TM) SE Runtime Environment (build 20+36-2344)
Java HotSpot(TM) 64-Bit Server VM (build 20+36-2344, mixed mode,
sharing)
```

Cela signifie que la version est 20 est bien installée et fonctionnelle sur l'environnement Microsoft Windows.

2.2 Choisir et installer un Environnement de Développement Intégré (IDE)

Qu'il s'agisse du langage Java ou de tout autre langage de programmation, il existe plusieurs outils qui facilitent l'écriture de codes aux développeurs. Ces outils sont appelés Environnement de Développement Intégrés, en anglais *Integrated Development Environments* (IDE). Bien que les IDEs ne soient pas nativement nécessaires pour écrire un code, ils sont néanmoins très importants au point d'être devenus incontournables dans le processus de développement. Un IDE est un logiciel avec une interface graphique qui offre plusieurs facilités au développeur : organier et structurer le projet ; proposer des corrections de syntaxes, faire des suggestions de codes, proposer des complétions de codes, aider à déboguer, compiler et exécuter le code, etc. L'ensemble de ces facilités génèrent un gain de temps très significatif lors du développement.

Il existe plusieurs dizaines d'IDEs sur le marché. Certains sont plus spécialisés sur des langages spécifiques alors que d'autres plus généralistes. La page suivante présente un classement actualisé de la popularité des IDEs dans la communauté des développeurs : https://pypl.github.io/IDE.html

Les IDEs les plus utilisés dans les projets Java sont Eclipse IDE, Intellij IDEA et NetBeans IDE. Eclipse et NetBeans sont totalement gratuits alors que Intellij IDEA propose deux offres : une version payante et une version Community.

Dans ce tutoriel, nous présentons la procédure d'installation et d'utilisation de ces trois IDEs à savoir : Intellij, Eclipse, et NetBeans. Encore une fois, les procédures présentées dans ce document ont été testées uniquement sur Microsoft Windows 11.

2.2.1 Installation et configuration de Intellij IDEA

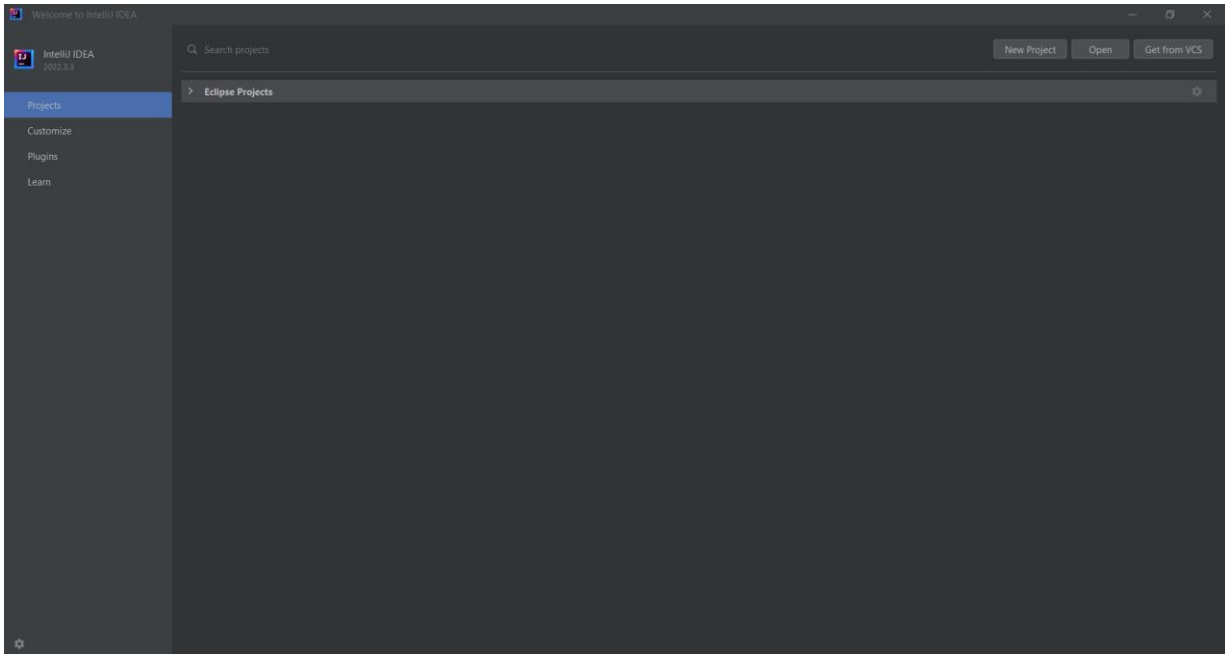
2.2.1.1 Télécharger et installer Intellij IDEA

- Télécharger et installer Intellij depuis le site :
  https://www.jetbrains.com/idea/download/#section=windows

Attention : seule la version community est gratuite.

- Après installation, lancer Intellij

La fenêtre d'accueil se présente comme suit :



2.2.1.2 Initialiser un projet de test et configurer le JDK

- Cliquer sur New Project en haut à droite de la fenêtre d'accueil.

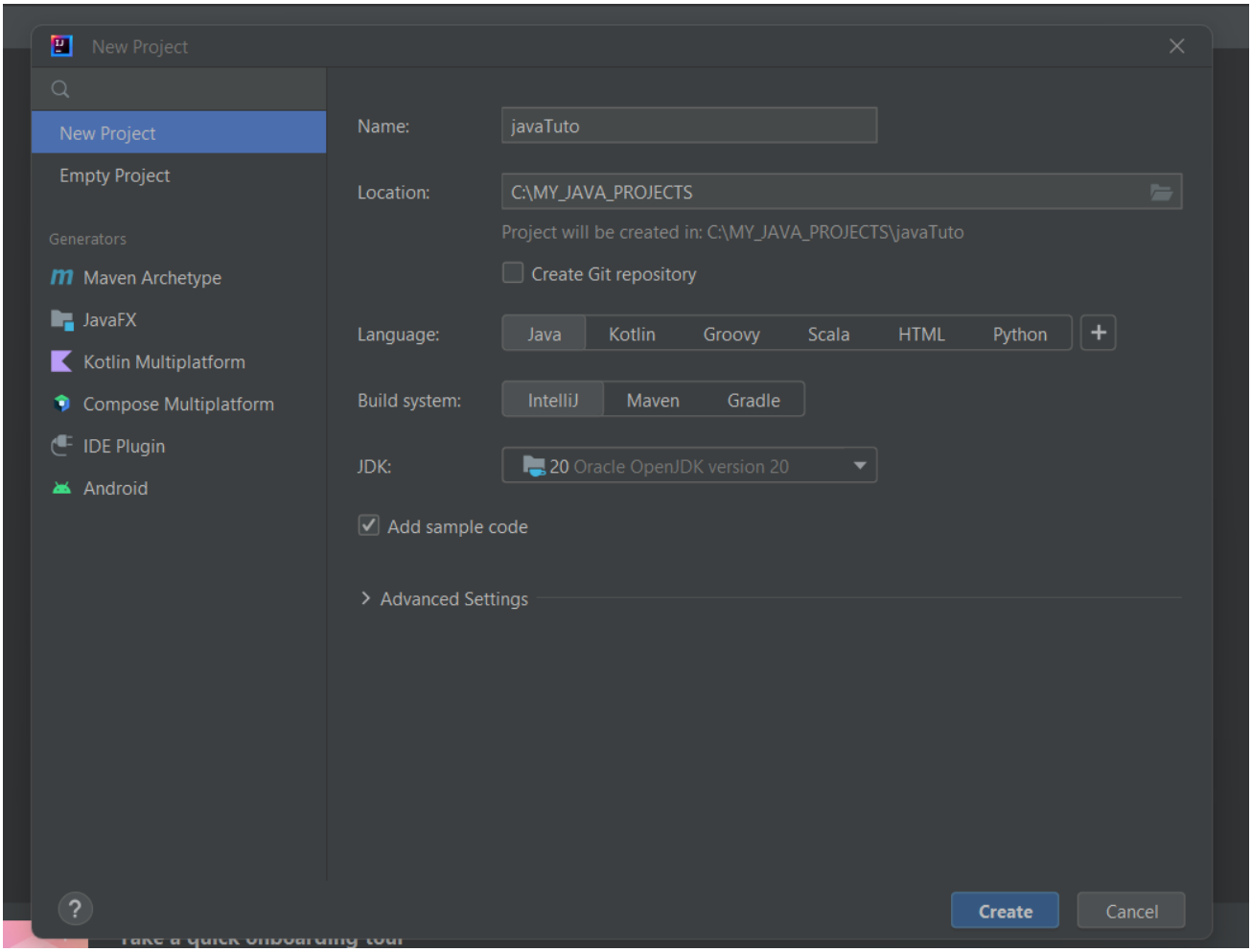
Une nouvelle fenêtre apparaît.

- Dans le champ Name, indiquer le nom du projet. Ex : javaTuto
- Dans le champ Location, indiquer votre workspace c'est-à-dire l'arborescence parent qui contiendra tous vos projets Java. Ex : C:\MY_JAVA_PROJECTS
- Dans language, choisir Java.
- Dans l'option Build System, choisir Intellij.
- Laisser cocher, Add sample code
- Dans le champ JDK, dérouler la liste et cliquer sur Add JDK

Indiquer le chemin vers le JDK que vous avez préalablement installé (au besoin voir la section décrivant la procédure d'installation du JDK).

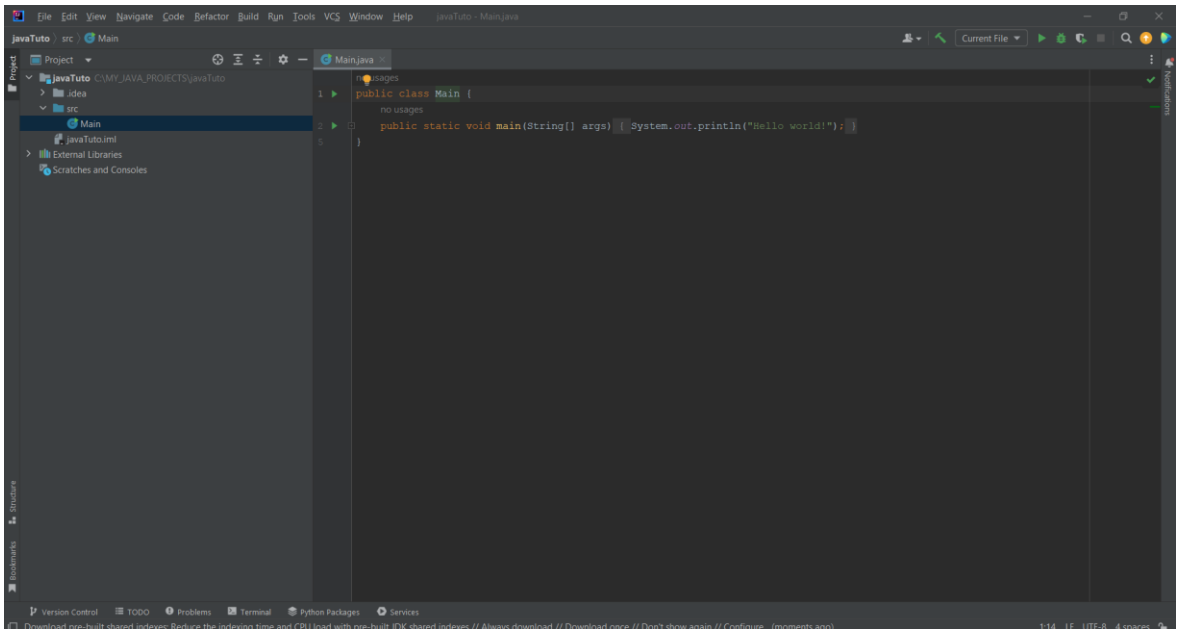
Le chemin du JDK que nous avons installé pour ce tutoriel est : C:\Program Files\Java\jdk-20

- Coller ce chemin et cliquer sur Ok.



- Cliquer sur Create.

Le projet javaTuto est maintenant initialisé et Intellij ajoute automatiquement un template de code java dans le dossier src.



2.2.1.3 Tester Intellij : compiler et exécuter le code de test

1. *Ajouter le code de test*

- Dans le dossier src, cliquer droit sur le fichier Main. Choisir Refactor>Rename.
- Remplacer le nom Main par le nom Tuto. Et cliquer sur Refactor.
- Copier le bout de code ci-dessous et remplacer le contenu du fichier Tuto.java

```
package com.tuto;
class Tuto{
public static void main(String args[]){
System.out.println("Bonjour. Nous allons commencer à faire du Java");
    }
}
```

- Faire CTRL+S pour enregistrer.
- Après avoir collé ce bout de code, Intellij envoie une alerte au niveau de la ligne package com.tuto et la souligne en rouge.
- En faisant passer la souris au dessus de la ligne, Intellij nous fait la suggestion suivante : Package name 'com.tuto' does not correspond to the file path "

- Cliquer sur la proposition Move to package com.tuto.

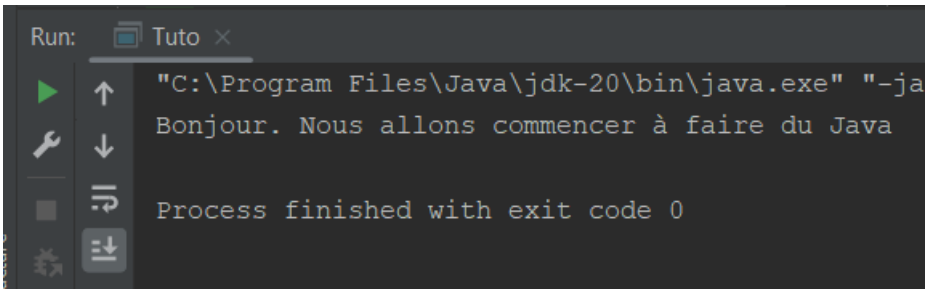
Intellij crée automatiquement le package com.tuto et déplace le fichier Tuto.java dans ce package. Pour info, un package est simplement un sous-répertoire ou un ensemble de sous-répertoires situé dans le dossier racine src. Le package permet d'isoler les codes dans leur propre dossier. Nous reviendrons plus tard sur les notions de package dans le document.

2. *Exécuter le code de test*

Pour vérifier que Intellij est bien installé et prêt pour développer le code Java, nous allons tester l'exécution du bout de code que nous venons d'ajouter. Pour cela :

- Cliquer sur le menu Run et choisir Run Tuto.java.

Si l'exécution s'est correctement déroulé, on devrait voir le message dans le console tel qu'affiché ci-dessous.

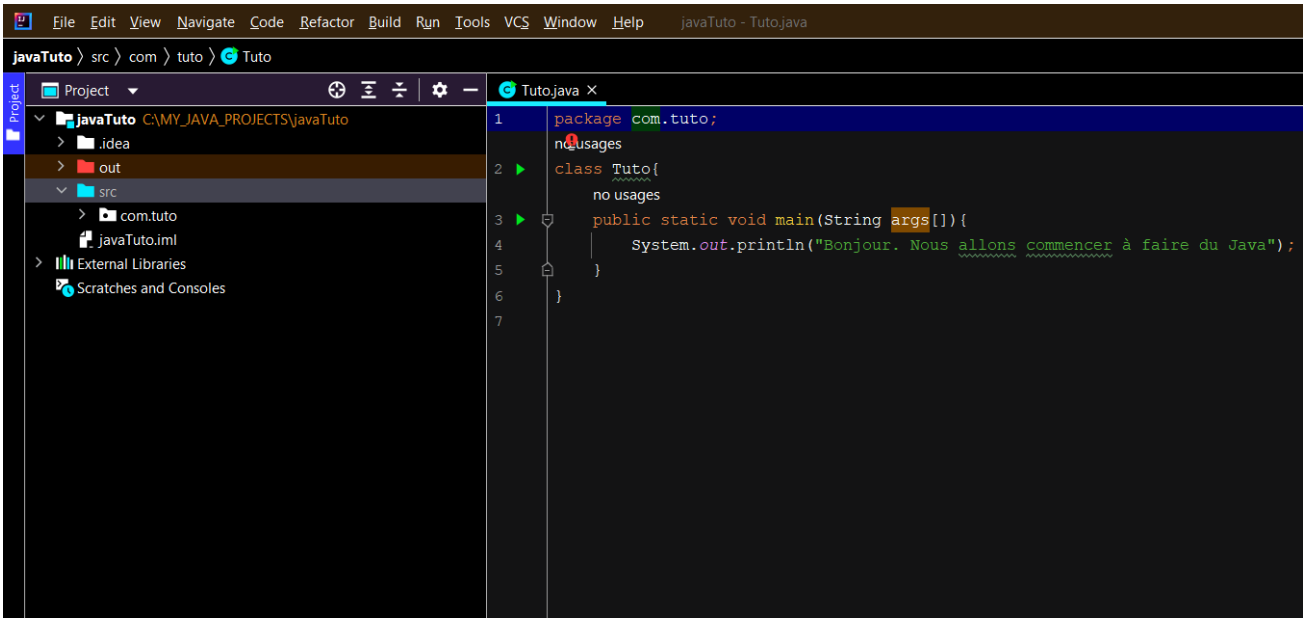


2.2.1.4 Changer la mise en forme du code : fond d'écran et police et taille

Intellij offre la possibilité de configurer le style de présentation de votre code. Voir les étapes ci-dessous.

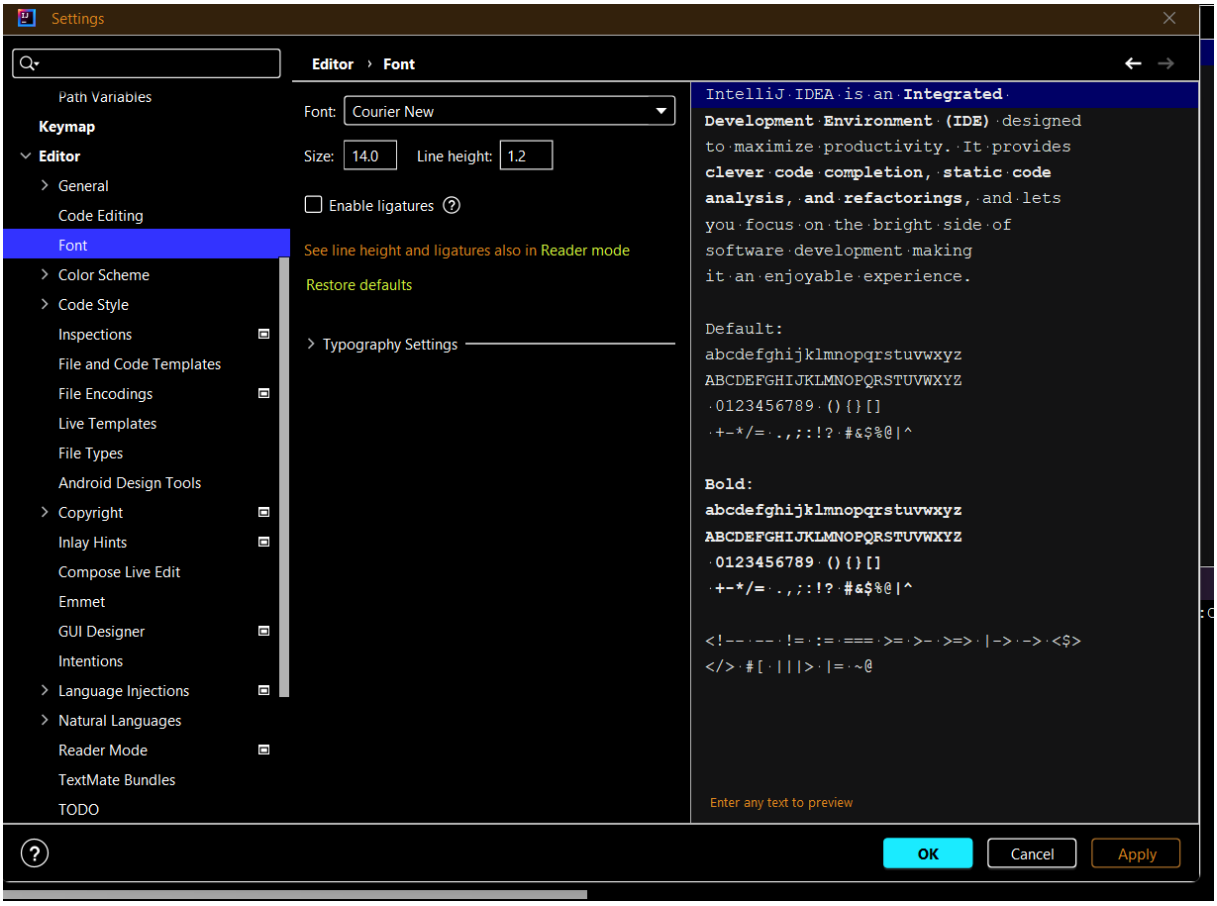
Changer la couleur de fond de l'écran

- Cliquer dans le menu File>Settings>Appearence & Behavior> Appearence.
- Dans le champ Theme, choisir le thème que vous préférez et cliquer sur Apply pour voir l'effet. Par exemple, choisir High constrast et cliquer sur Ok. On obtient cette couleur de fond ci-dessous.



Changer la police et la taille

- Cliquer dans le menu File>Settings>Appearence & Behavior> Editor > Font.
- Dans le champ Font, dérouler et choisir la police que vous souhaitez. Et dans Size, taper la taille que vous souhaitez. Et finir par cliquer Ok. Voir image ci-dessous.



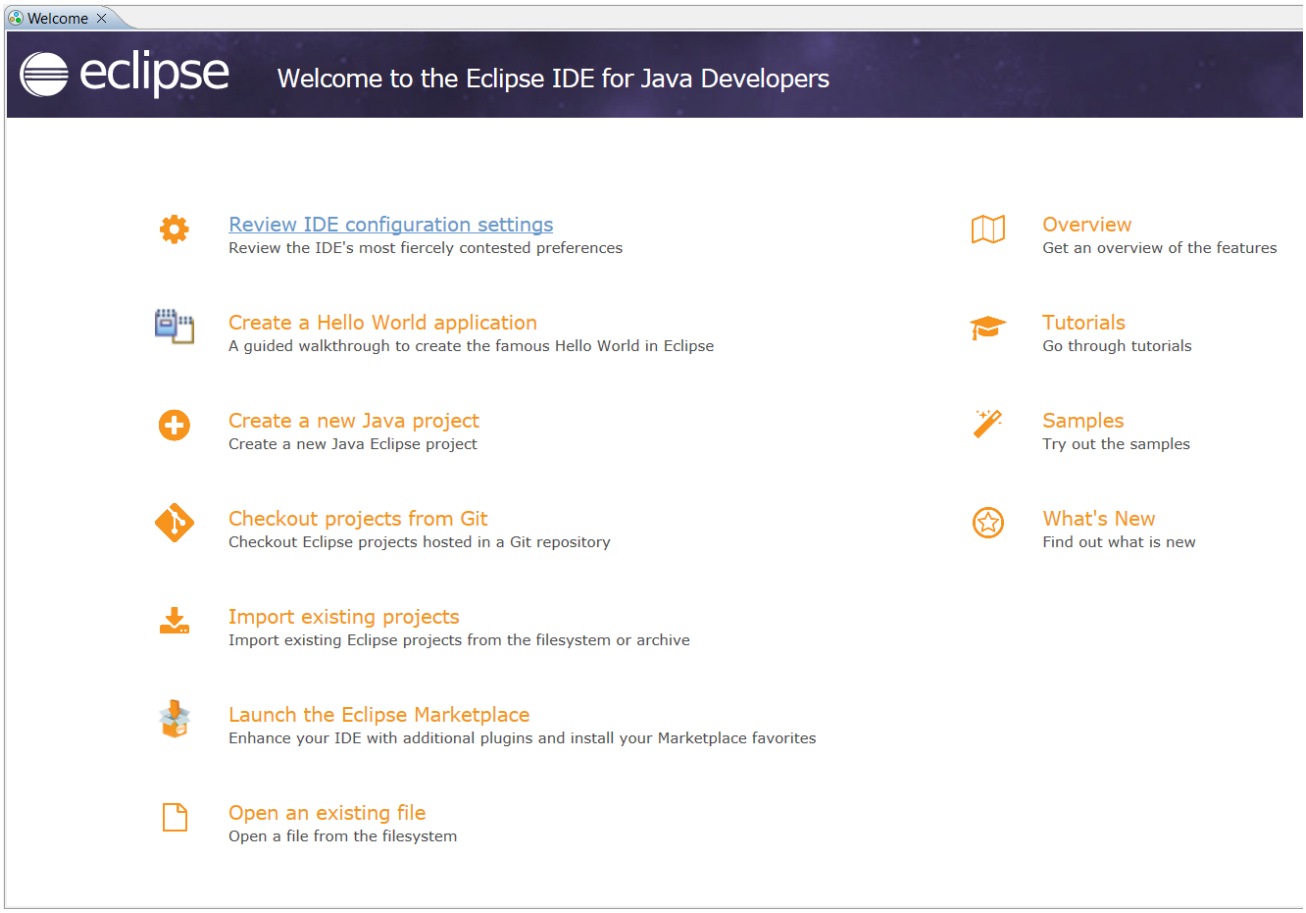
2.2.2 Installation et configuration de Eclipse IDE

2.2.2.1 Télécharger et installer Eclipse

- Télécharger et installer éclipse via le lien : https://www.eclipse.org/downloads/
- Après l'installation, lancer Eclipse

Au premier lancement, il vous sera demandé de spécifier un espace de travail appelé workspace. Le workspace est un répertoire censé contenir l'ensemble de vos projets de développement. Choisir un répertoire. Ex : C:\MY_JAVA_PROJECTS

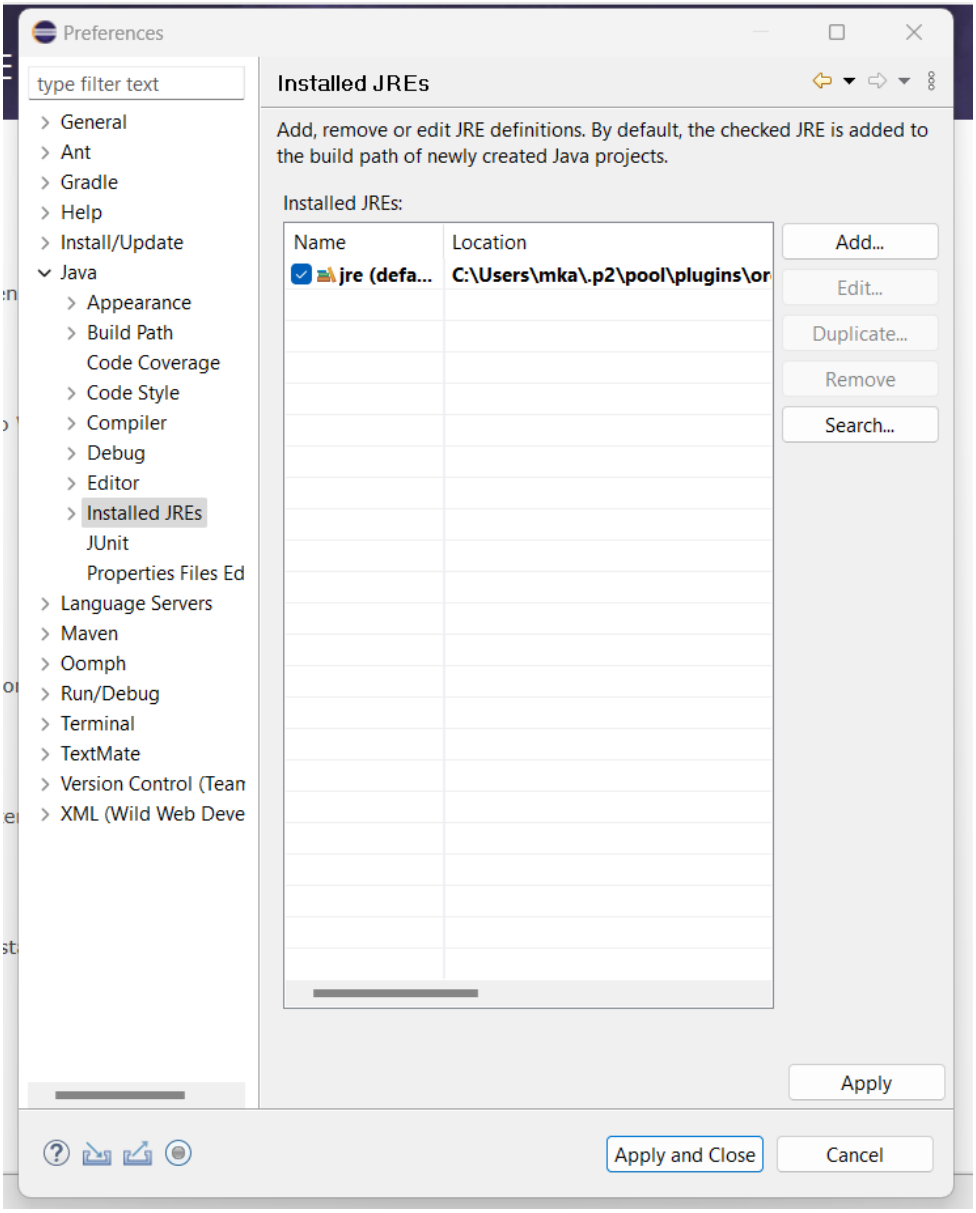
- Cliquer sur OK et la page d'accueil s'ouvre et se présente comme suit :



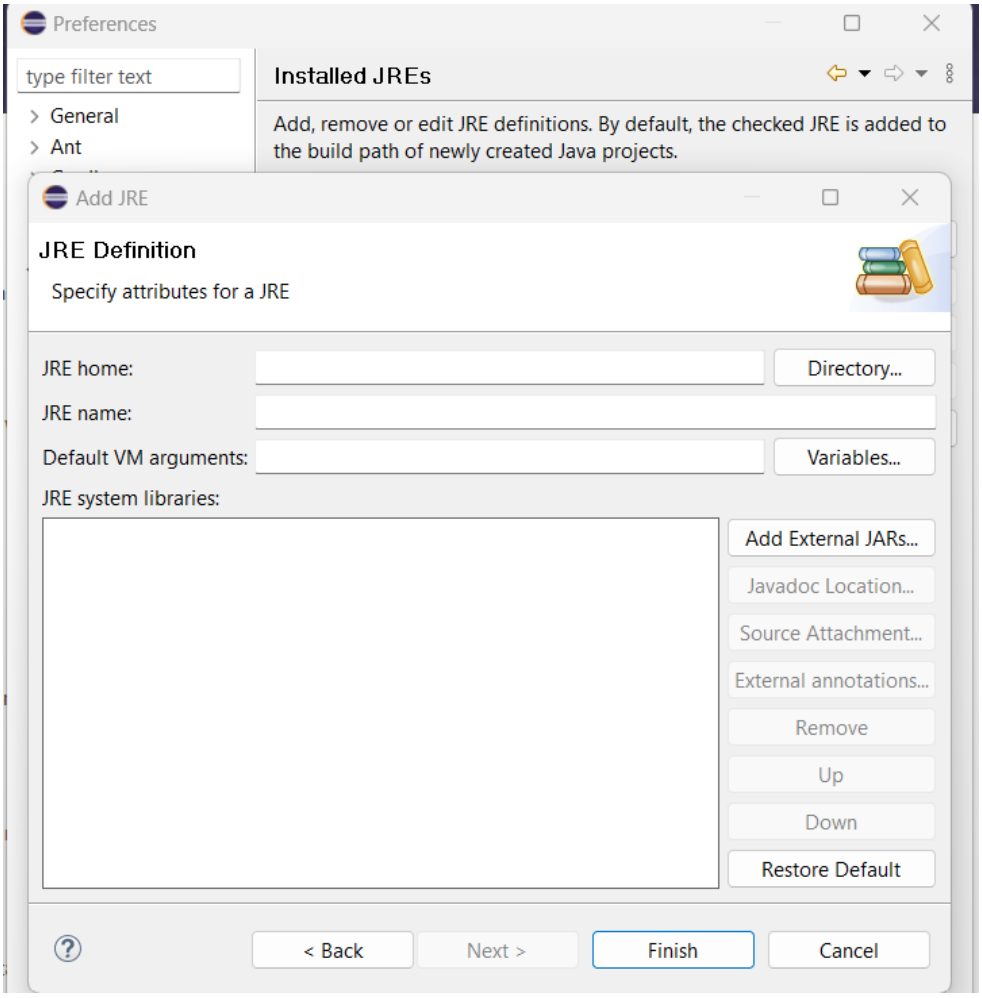
2.2.2.2 Pointer Eclipse sur le JDK déjà installé

A minima, pour pouvoir exécuter du code Java, Eclipse a besoin du Java Runtime Environment (JRE). Mais comme nous allons surtout développer du code, il nous faut utiliser le JDK pour pouvoir profiter de toutes les fonctionnalités qu'il offre. Pour pointer Eclipse sur le JDK que vous avez déjà installé, suivre les étapes décrites ci-dessous

1. Cliquer sur le menu Window >Preferences>Java>Installeds JRE

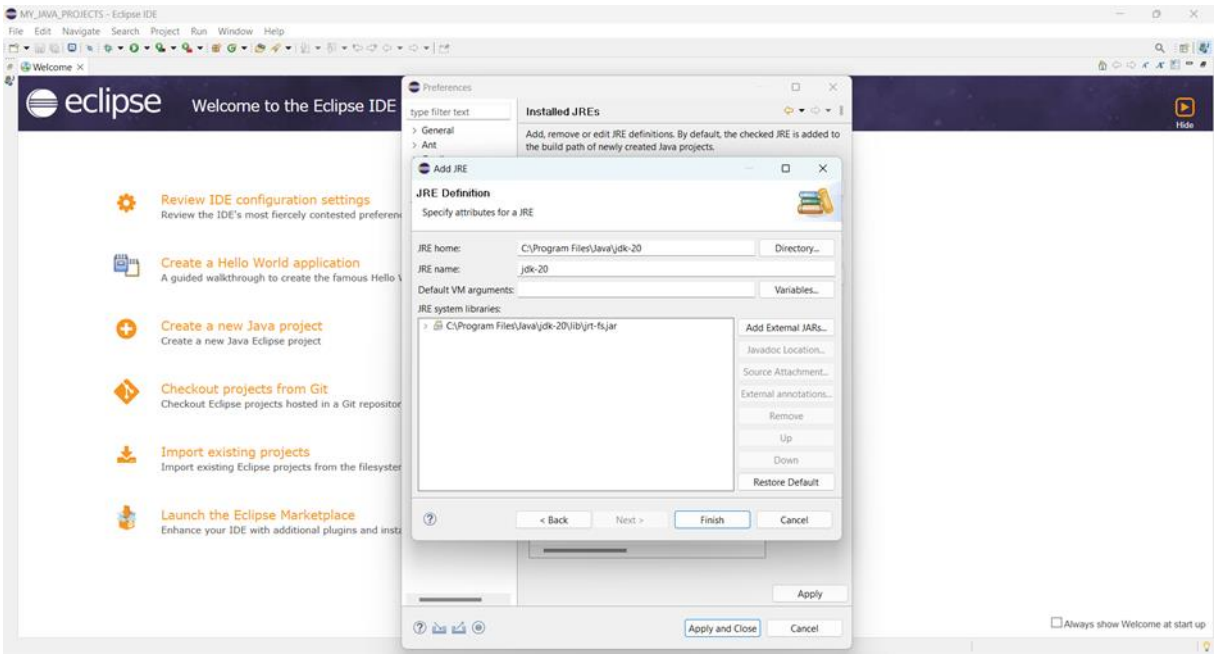


2. Cliquer sur Add, sélectionner l'option Standard VM et cliquer sur Next



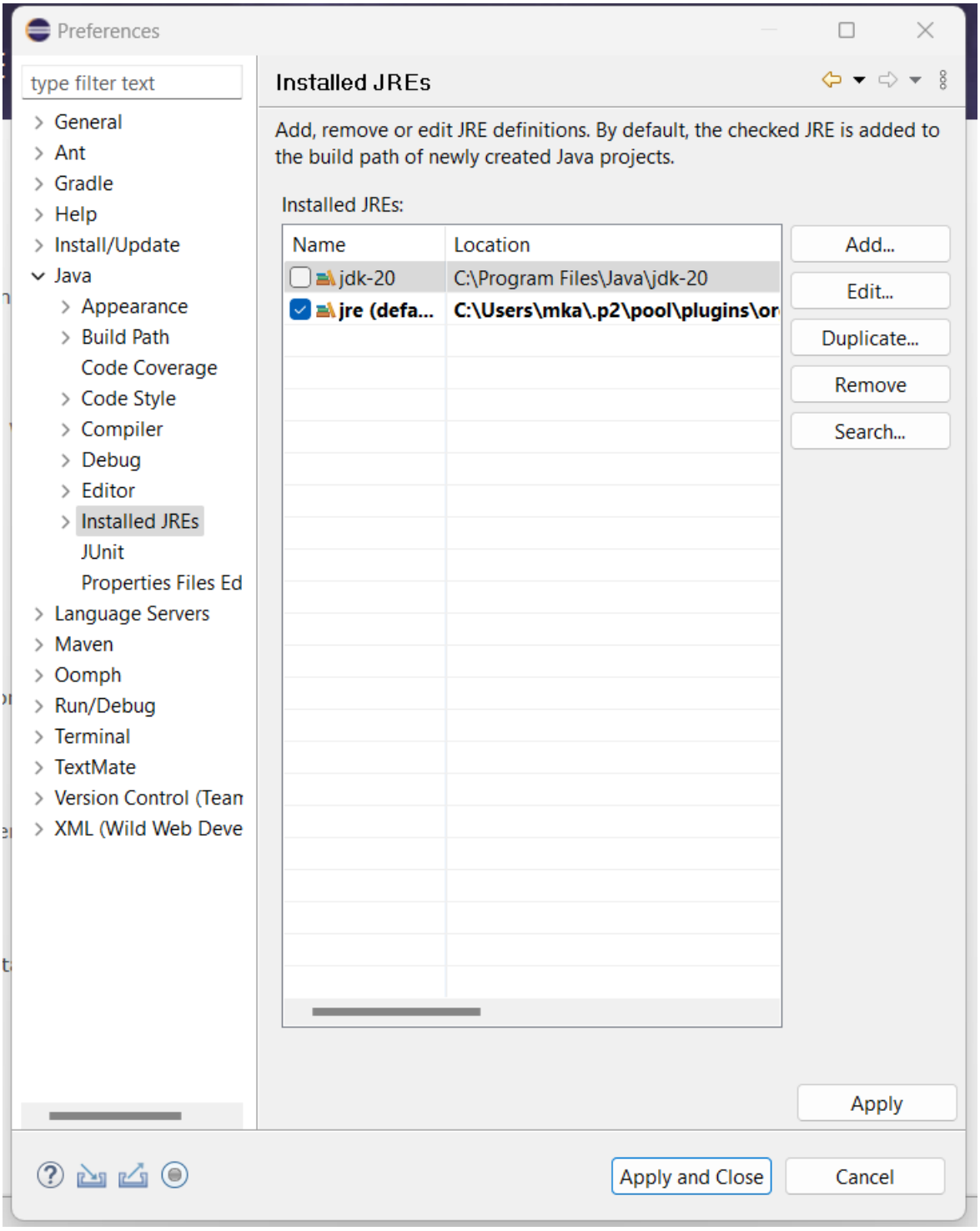
3. Devant JRE Home, cliquer sur Directory et indiquer le lien vers le répertoire où est installé votre JDK. Ex : C:\Program Files\Java\jdk-20

Et cliquer sur Sélectionner un dossier.



4- Cliquer sur Finish

5- Sélectionner le JDK que vous avez chargé et décocher les autres JDK disponibles.



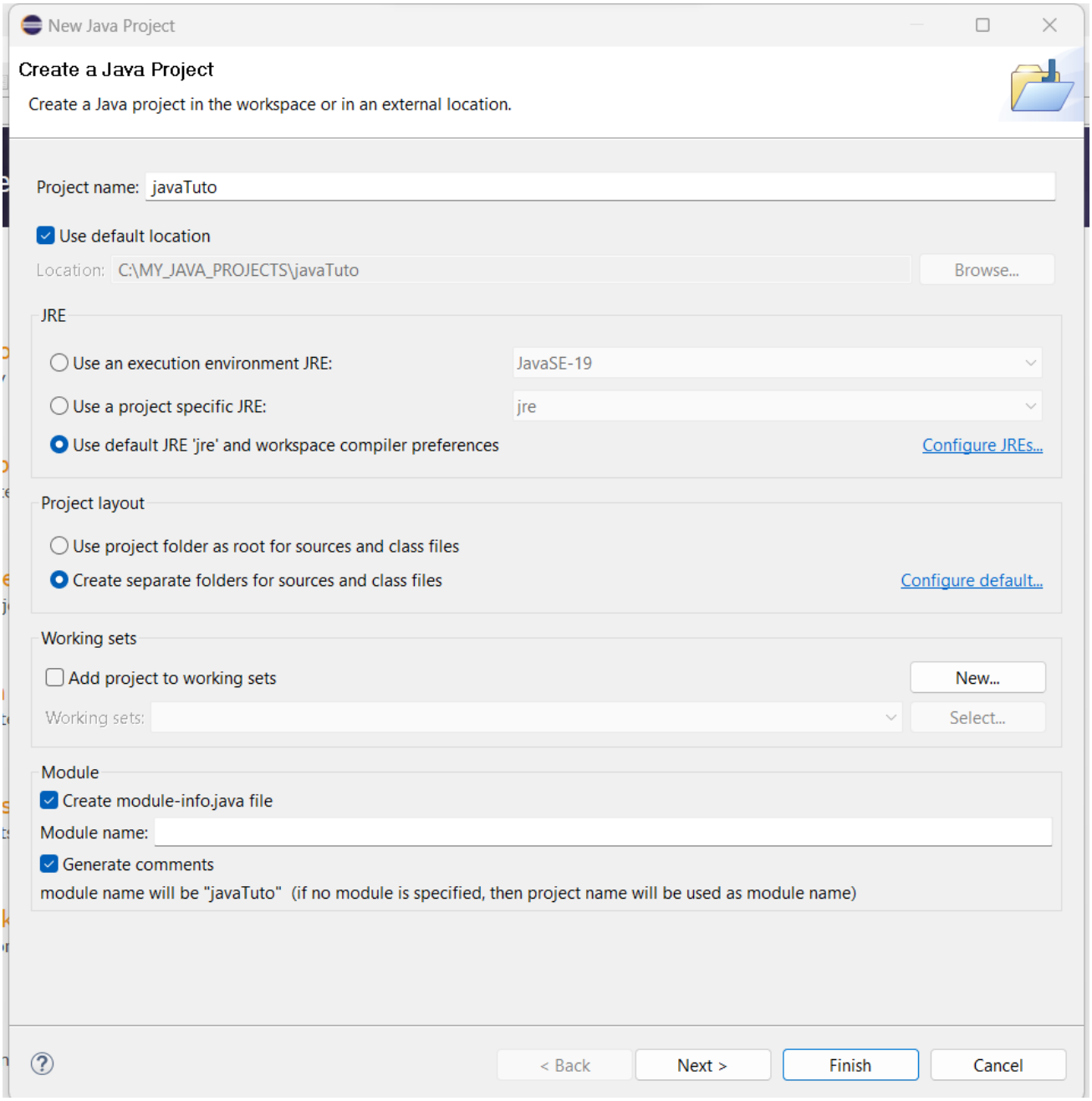
4. Cliquer sur Apply and Close

2.2.2.3 Tester Eclipse : Ecrire, compiler et exécuter un code de test

Pour tester que Eclipse est bien installé et fonctionne correctement, nous allons écrire, compiler et exécuter un code Java de test. Voici ci-dessous les étapes à suivre.

1. Créer un projet de test en faisant :

- Cliquer File>New>Java Project
- Dans Project name, indiquer le nom. Ex : javaTuto
- Dans la section JRE, choisir l'option Use default JRE



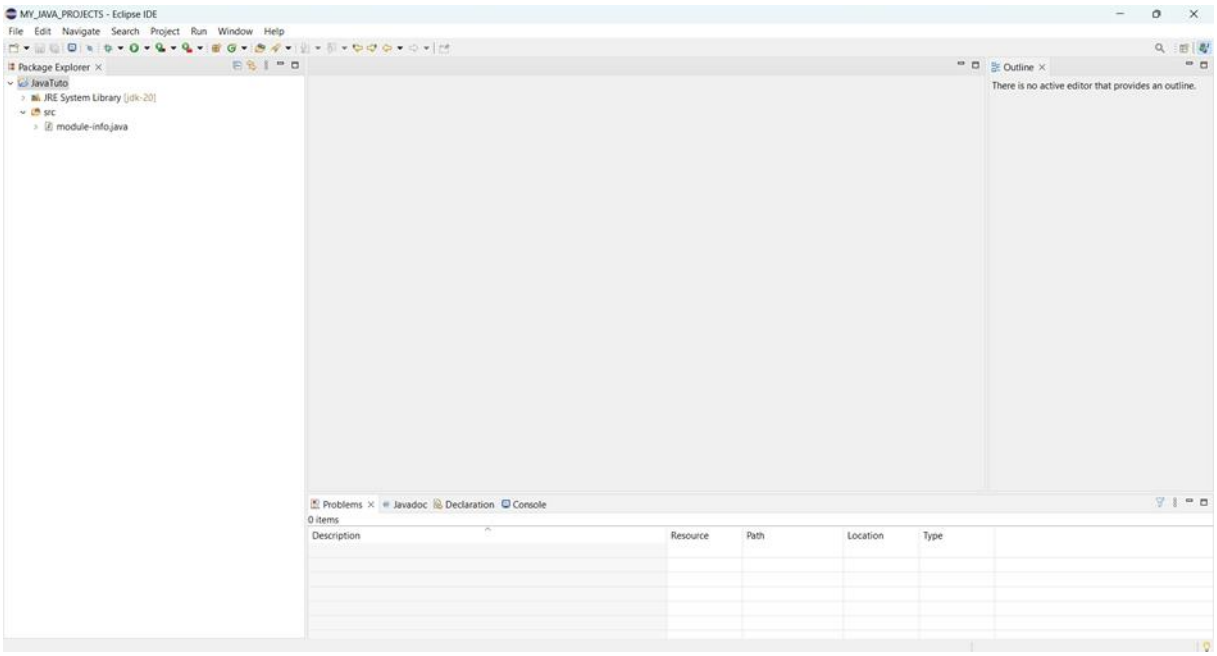
- Cliquer sur Finish.

La structure du projet est maintenant créée. Pour voir la structure du projet, cliquer en haut à gauche de l'explorateur.

Nous allons ajouter le code de test.

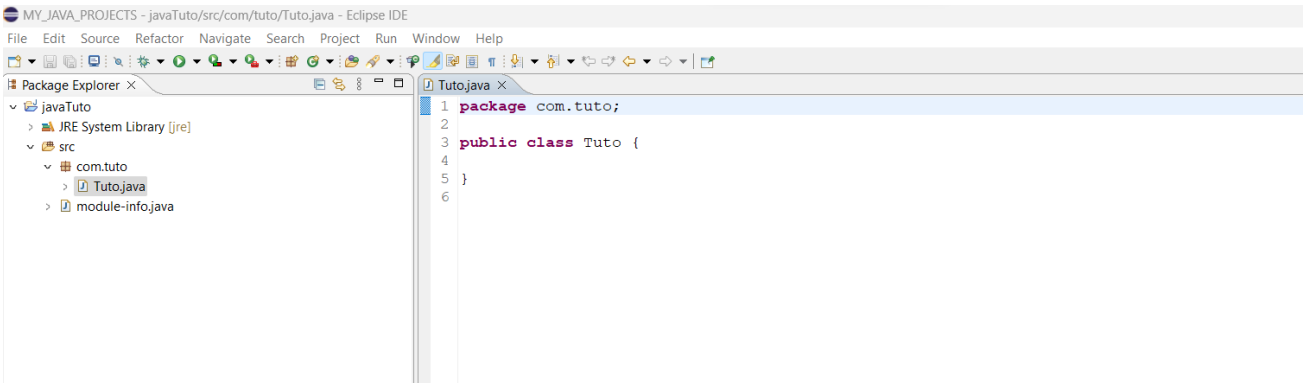
2. Ajouter le code de test

En haut à gauche, cliquer sur l'icône de la structure de votre projet. N'hésitez pas à fermer la page d'accueil de présentation d'Eclipse pour que votre projet s'affiche en pleine page.



- Cliquer droit sur le dossier src, cliquer New>Class.

  Dans le champ name, taper Tuto. Dans le champ package, taper *com.tuto*. Et cliquer sur Finish.



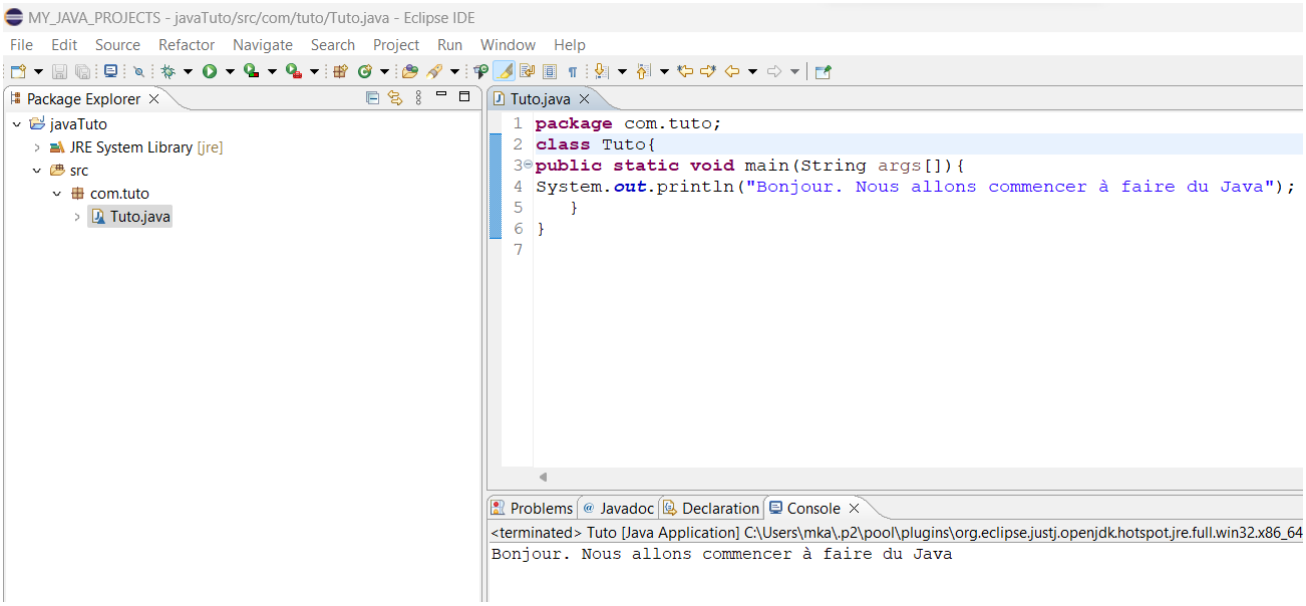
- Copier le bout de code ci-dessous et remplacer le contenu du fichier Tuto.java

```
package com.tuto;
class Tuto{
public static void main(String args[]){
System.out.println("Bonjour. Nous allons commencer à faire du Java");
    }
}
```

- Supprimer le fichier module-info.java
- Enregistrer le fichier Tuto.java en faisant CTRL+S

3. Exécution du code

Aller dans menu Run et cliquer sur run, cliquer sur Select all et cliquer sur Ok. Vous devez voir le message dans le console.

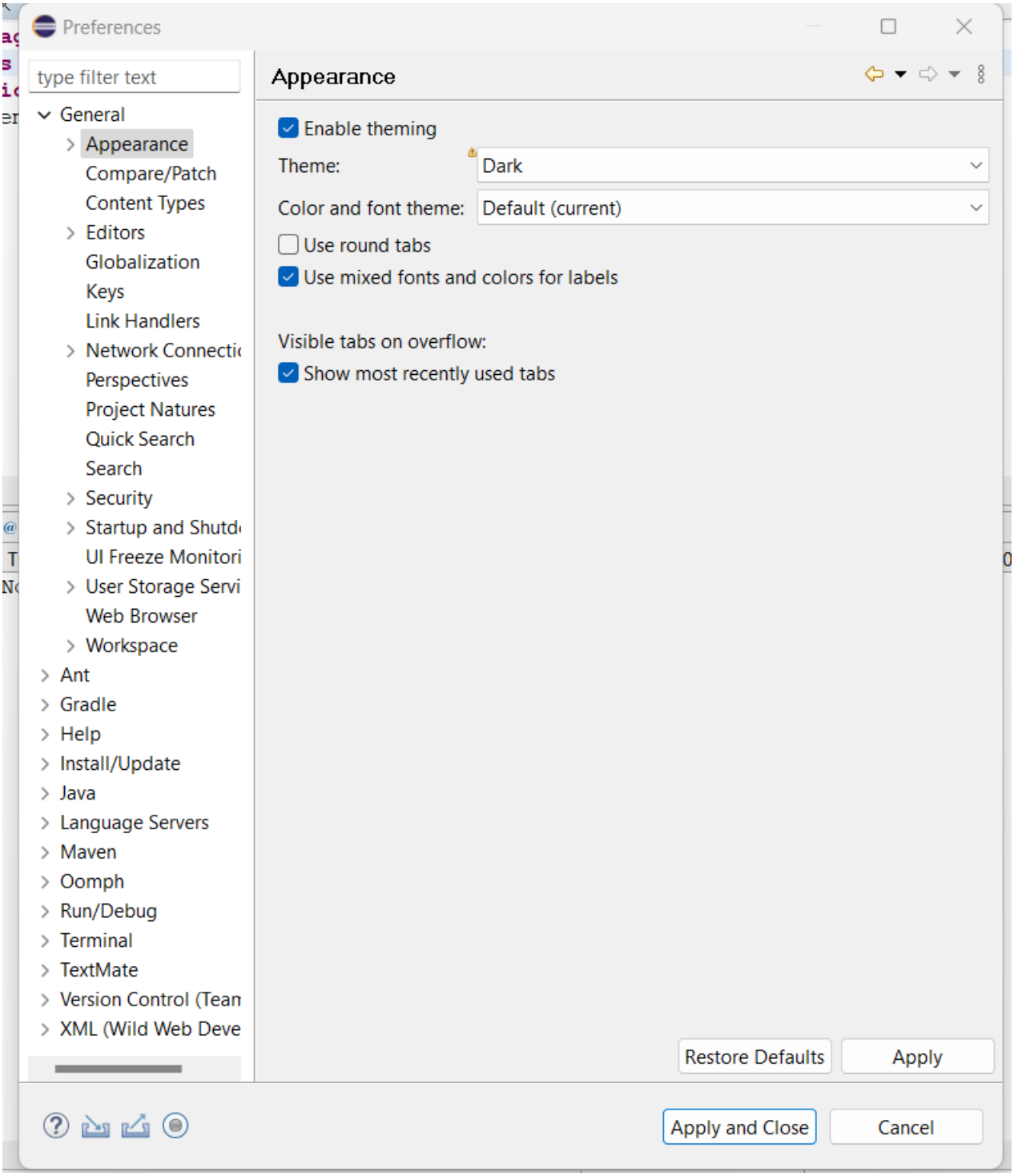


2.2.2.4 Changer la mise en forme du code : fond d'écran, police et taille.

Chaque développeur a sa préférence pour présenter son code qu'il s'agisse de la couleur du fond d'écran de l'IDE ou de la police d'écriture du code. Eclipse offre la possibilité de configurer le style de présentation de votre code.

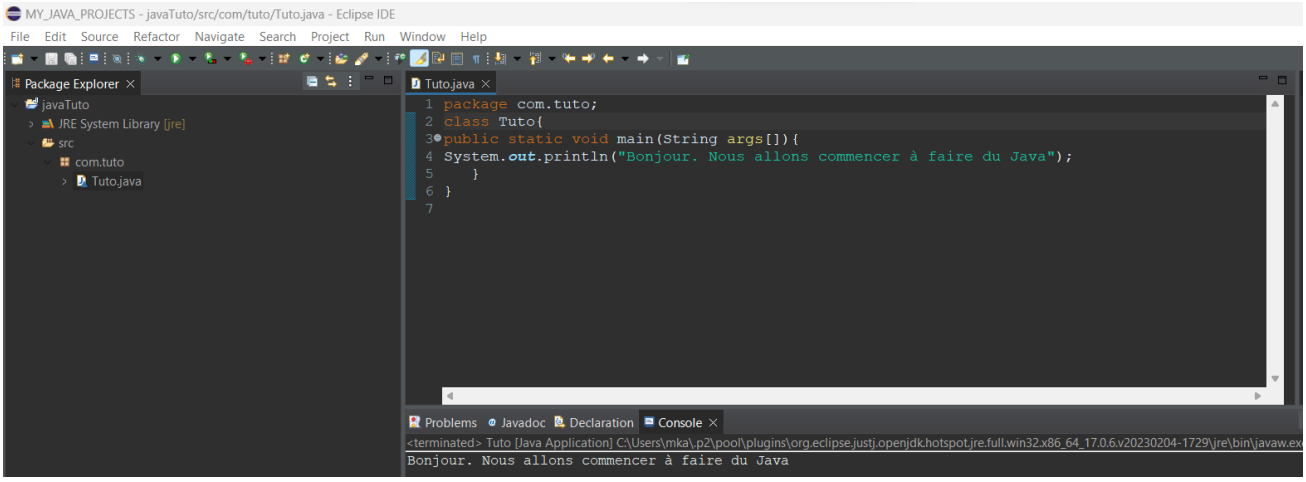
Changer la couleur de fond

- Cliquer dans le menu Window>Preferences>General>Appearence.



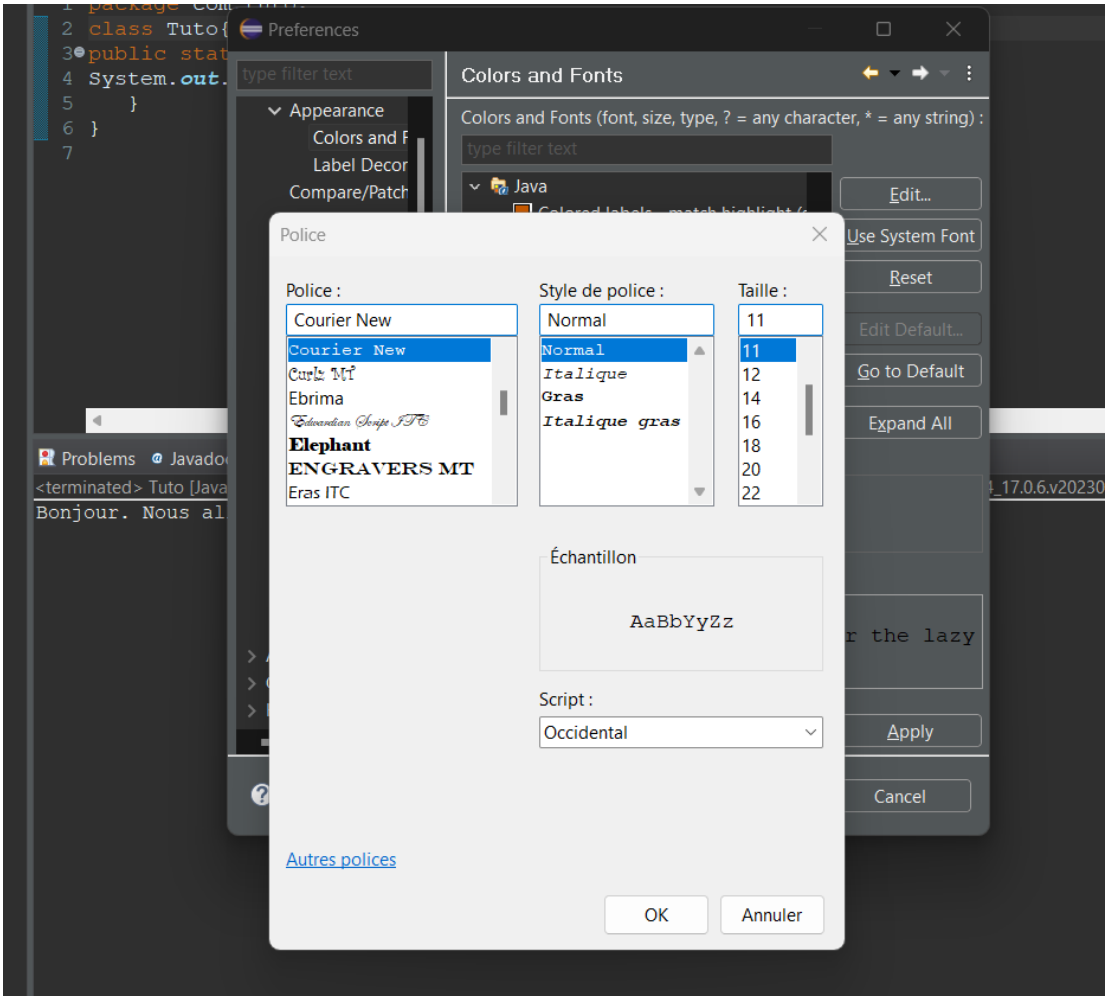
- Dans le champ Theme, choisir le thème que vous préférez et cliquer sur Apply pour voir.

Par exemple, choisir Dark et cliquer sur Apply and Close. Il n'est pas nécessaire de redémarrer si Eclipse vous le propose. On obtient cette couleur de fond ci-dessous.



Changer la police et la taille

- Cliquer sur Window>Prefrences>General>Appearence > Colors and Fonts >Java.
- Double-cliquer sur Java Editor text font. On peut ainsi choisir la police, le style et la taille de police que nous préférons. Voir capture d'écran ci-dessous.



2.2.3 Installation et configuration de Netbeans IDE

2.2.3.1 Télécharger et installer Netbeans

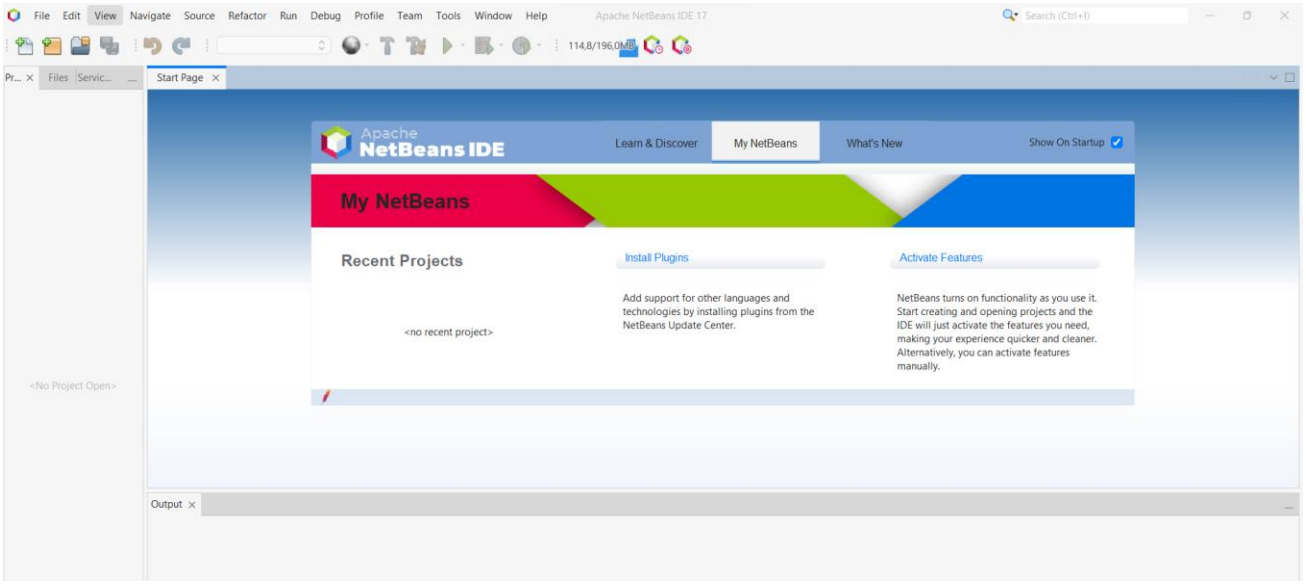
- Télécharger et installer NetBeans depuis ce lien :

https://netbeans.apache.org/download/index.html

Au démarrage de l'installation, NetBeans vous propose directement de choisir le JDK. Dans ce cas, choisissez le JDK que vous avez précédemment installé. Ex : C:\Program Files\Java\jdk-20

- Après l'installation, lancer NetbBeans

La fenêtre d'accueil de l'IDE se présente comme suit :



2.2.3.2 Initialiser un projet de test

- Dans le menu, cliquer File>New project. Choisir Java with Maven et à droite choisir Java Application. Cliquer sur Next

  Une nouvelle fenêtre apparaît.

- Dans le champ Project name, indiquer javaTuto.
- Dans le champ Project location, indiquer votre workspace.

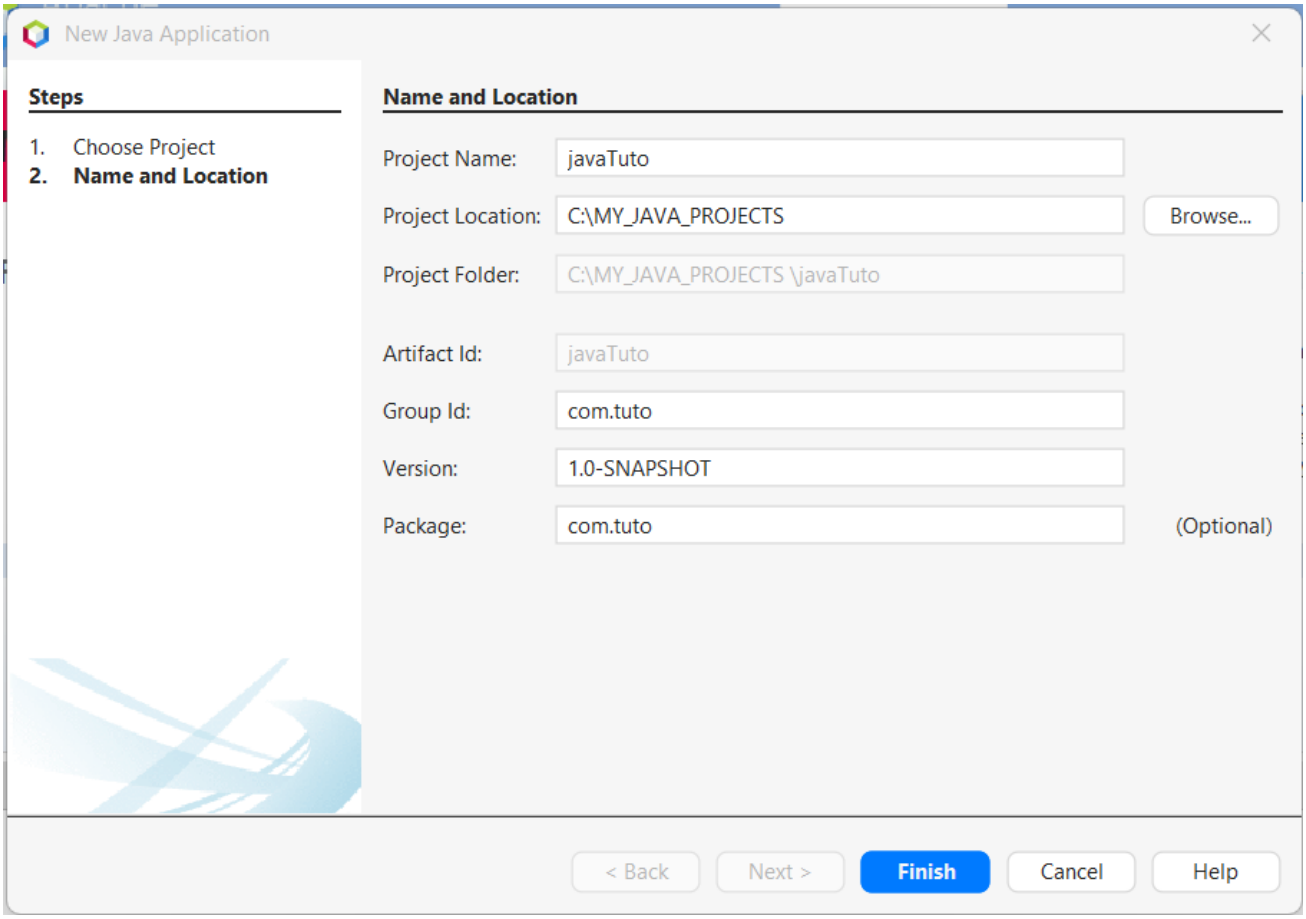
Le workspace est l'arborescence parent qui centralise tous vos projets Java. Ex : C:\MY_JAVA_PROJECTS

- Dans groupId indiquer *com.tuto*

Le groupId est l'identifiant du groupe de code. Techniquement, il s'agit du package de base de l'application.

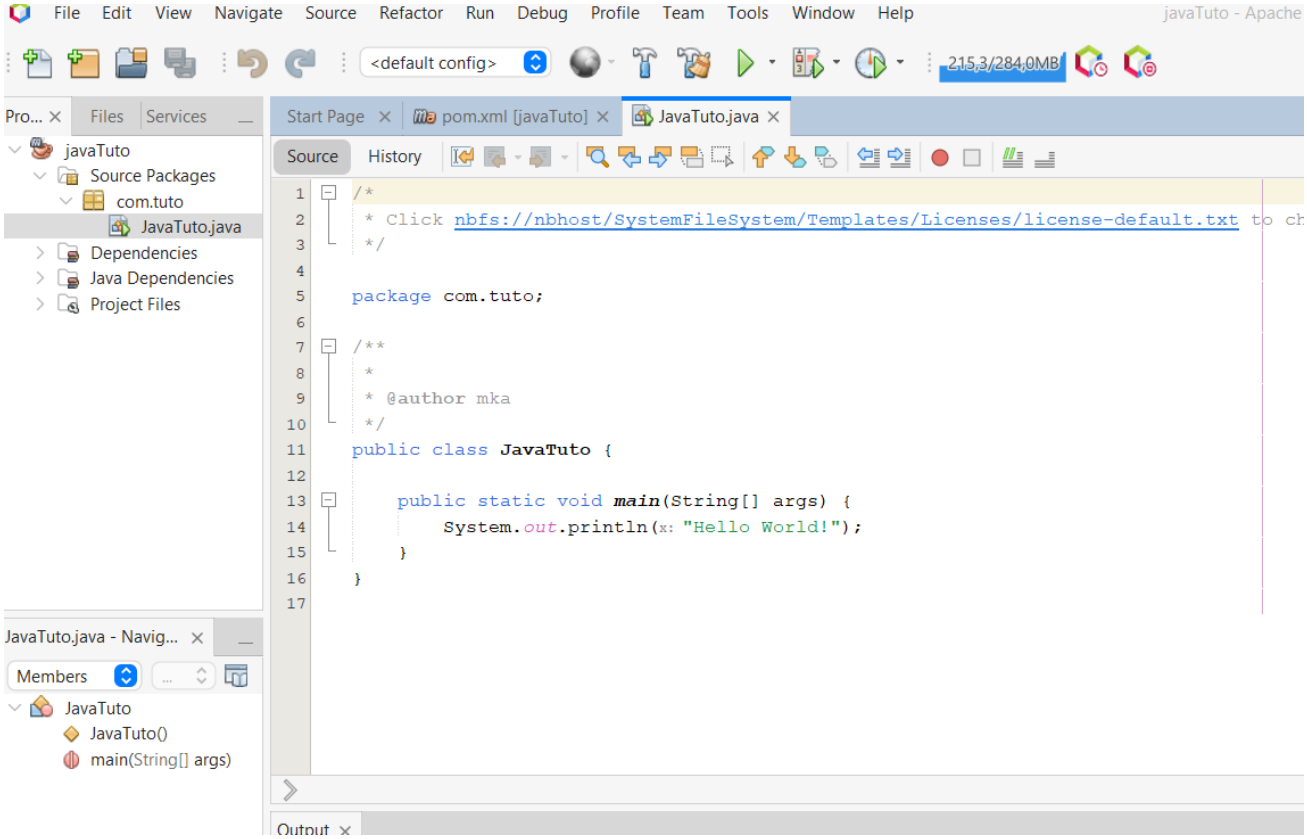
- Dans package, indiquer *com.tuto*

La valeur du package peut être égale à la valeur du groupId qui est le package de base. La valeur du package peut aussi être une extension du groupId c'est-à-dire une sous-arborescence du package de base. C'est dans le package que sont les situés les fichiers de code sources seront créés pour être isolés. Nous détaillerons plus tard sur la notion de package.



- Cliquer sur Finish

L'interface se présente alors comme suit :



2.2.3.3 Tester NetBeans : compiler et exécuter le code de test

3. *Ajouter le code de test*

Dans le sous-répertoire (package) *com.tuto*, Eclipse crée automatiquement un fichier Java nommé JavaTuto.java et ajoute un template de code. Mais nous allons changer ce fichier pour ajouter notre propre fichier et notre propre code source. Pour cela, suivre les étapes suivantes :

- Clique-droit sur le fichier JavaTuto.java, et choisir Refactor>Rename.
- Remplacer le nom JavaTuto par le Tuto. Et cliquer sur Refactor.
- Copier le bout de code ci-dessous et remplacer tout le contenu du fichier Tuto.java.

```
package com.tuto;
class Tuto{
    public static void main(String args[]){
        System.out.println("Bonjour. Nous allons commencer à faire du Java");
    }
}
```

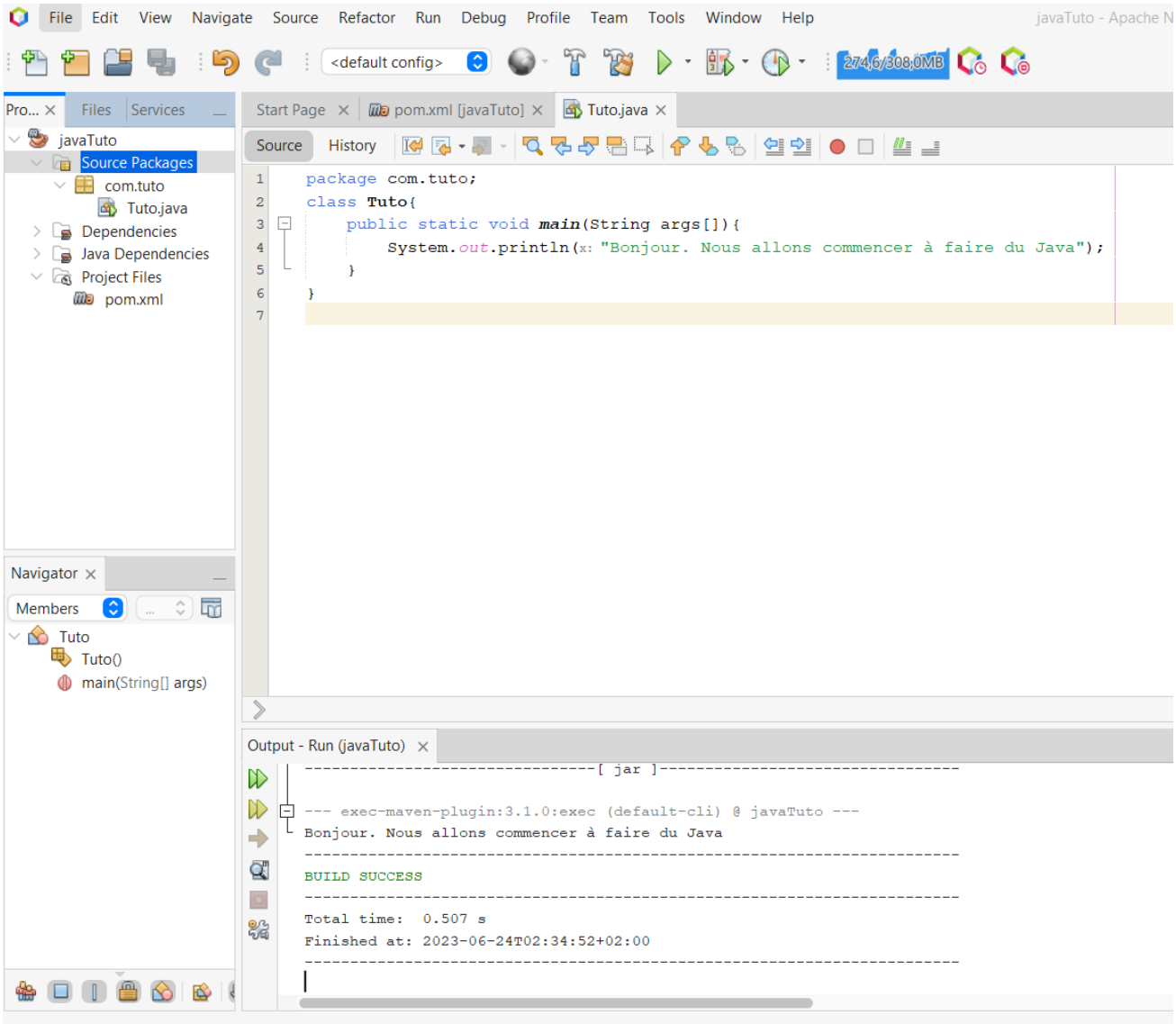
- Cliquer sur CTRL+S pour enregistrer la modification.

4. Executer le code de test

Pour vérifier que NetBeans est correctement configuré, nous allons tester l'exécution du bout de code que nous venons d'ajouter. Pour cela :

- Cliquer sur le menu Run et choisir Run project (javaTuto).

Si l'exécution s'est correctement déroulée, on devrait voir le message dans le console tel qu'affiché ci-dessous.



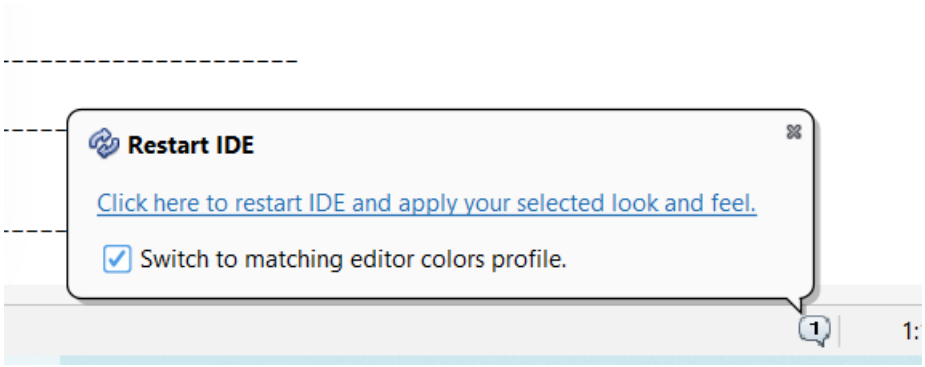
2.2.3.4 Changer la mise en forme du code : fond d'écran et couleur, police et taille

NetBeans offre la possibilité aussi de configurer la présentation des fonds d'écran et des polices d'écriture des codes. Ci-dessous les étapes pour changer les styles de présentation de votre code.

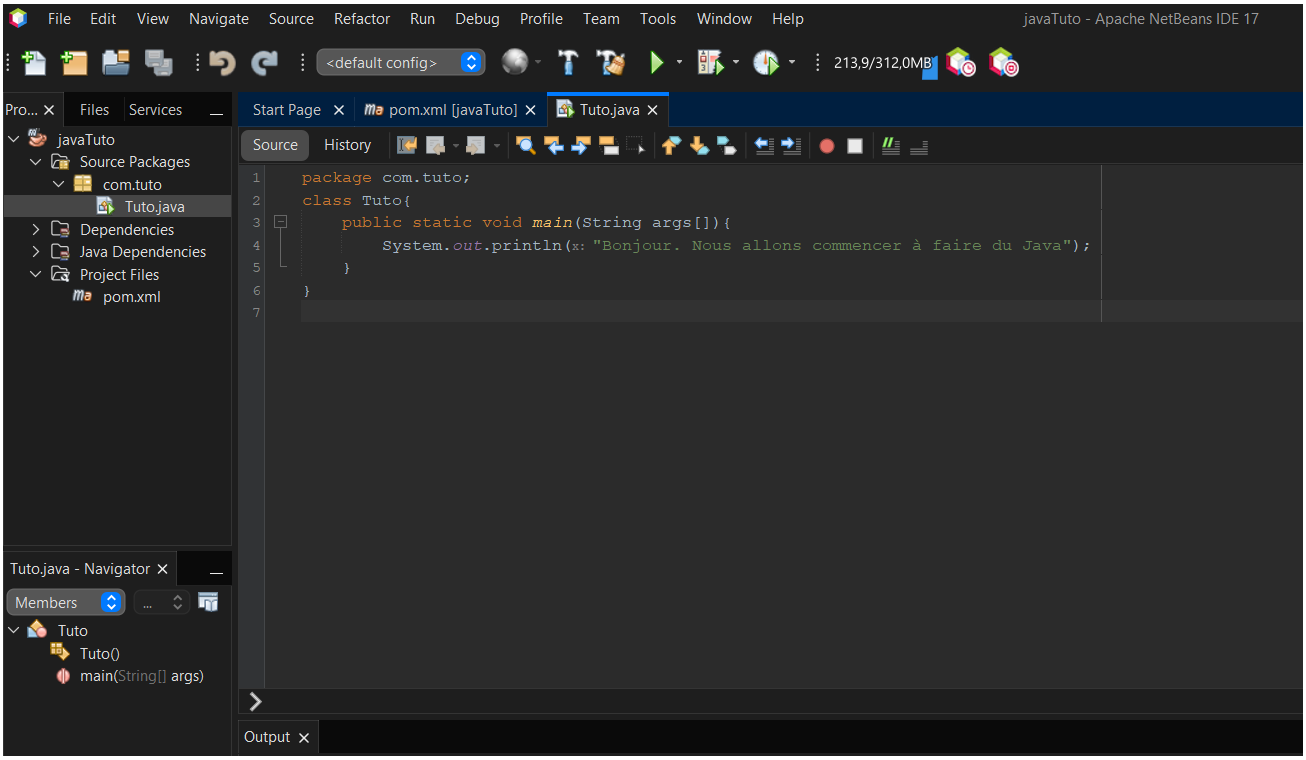
Changer la couleur de fond de l'écran

- Cliquer dans le menu Tools>Options.
- Dans la fenêtre qui apparaît, cliquer sur l'onlget Appearence>Look and Feel.
- Dérouler le champ Preferred look and feel, choisir le thème que vous préférez.

Par exemple choisir le thème Dark metal et cliquer sur Apply. Une petite fenêtre apparaît et vous invite à cliquer pour relancer Netbeans pour que les modifications puissent prendre effet.

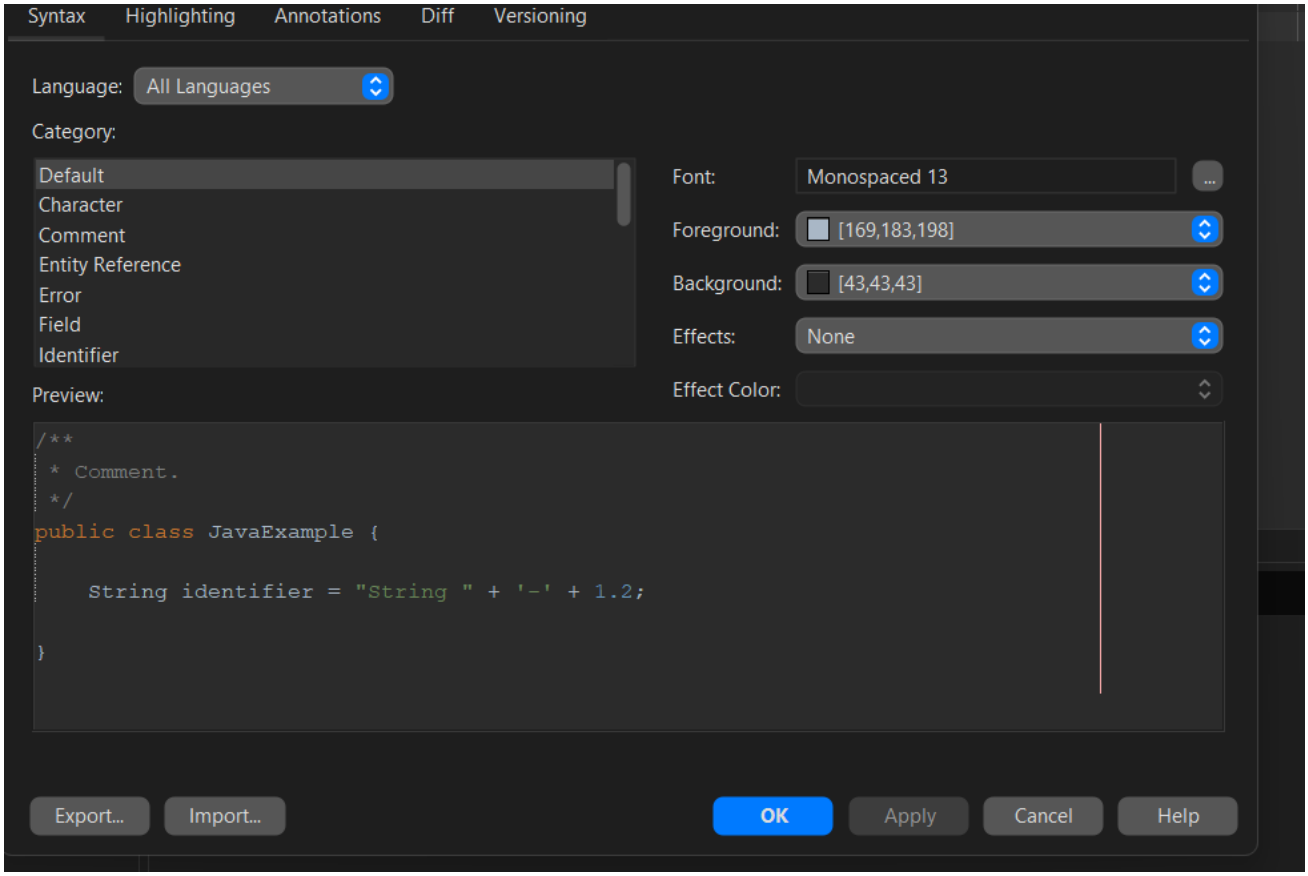


Après le redémarrage de NetBeans, l'interface se présente comme suit :



Changer la police et la taille

- Pour changer la police, cliquer dans le menu Tools>Options> Font and Colors.



Devant le champ Font, cliquer sur les trois points … Et choisir la police et la taille que vous souhaitez. Ensuite cliquer sur Ok pour les appliquer.

3 LES ÉLÉMENTS DE BASE DU LANGAGE JAVA

Ce chapitre présente les différents éléments qui forment l'ossature du langage Java. Il s'agit notamment des variables, la syntaxe d'écriture des blocs d'instruction, les différents types des variables, les opérateurs, les structures de contrôle mais également les mots réservés du langage.

3.1 Les instructions et les blocs d'instructions

Une instruction est une action ou un opération matérialisée par un code et visant à réaliser une tâche bien définie. En Java, les instructions sont toujours terminées par un point-virgule « ; ».

```
int i=0;
```

Un bloc d'instructions est une succession d'instructions. Il est souvent délimité par des accolades {… }. Ex :

```
{ int i=0;  String j="Hello world"; }
```

Les blocs d'instructions peuvent contenir aussi bien des instructions simples mais aussi des instructions plus complexes comme des listes de choix, des boucles ou mêmes d'autres blocs d'instructions. Ex :

```
{ int i=0;
    while (i<10) {
        System.out.println("Hello world");
        i=i+1;
    }
}
```

Ce bloc d'instructions fait une boucle et affiche dix fois l'expression « Hello world ».

3.2 Commentaires de codes

A la différence des instructions, les commentaires sont des corps de texte que le développeur utilise pour documenter son code. Les commentaires sont très utiles car ils permettent de faciliter la lecture et la compréhension du code. En Java, il existe deux façons d'ajouter un commentaire. Voir l'exemple ci-dessous.

```
{
    int i=0;// On initialise la variable i.
    /* Ici on fait une boucle et on affiche Hello world
       tant que la valeur de i est inférieure à 10. */
    while (i<10) {
        System.out.println("Hello world");
```

```
            i=i+1; // On incrémente la valeur de i.
        }
}
```

Pour ajouter un commentaire, on utilise le double slash // lorsque le commentaire tient sur une seule ligne. Et on utilise le /* */ lorsqu'il s'agit d'un bloc de texte qui peut s'étendre sur plusieurs lignes.

3.3 Les variables

En Java, une variable est tout identificateur permettant de stocker ou de référencer une information dans le programme. Une variable est caractérisée notamment par son nom et son type. Comme nous allons le voir plus tard, les variables Java peuvent être de plusieurs types, allant des plus simples (comme des chiffres ou des lettres) aux plus complexes comme des classes, des objets ou des collections d'objets. Mais ici, pour illustrer la notion de variable nous nous limitons d'abord aux cas simples.

3.3.1 Définition d'une variable : déclaration et assignation

La déclaration d'une variable nécessite de définir un nom et de spécifier un type. Ex :

```
int myVar1=234;
long myVar2=183946386434L;
double myVar3=12.123456789;
float myVar4= 12.12345F;
String myVar5="Hello world";
boolean myVar6=true ;
int myVar7;
int myVar8= null;
```

Cet exemple présente huit cas de définition de variables. Un typage est déclaré pour chacune des variables définies. Par exemple, myVar1 est de type numérique entier (int), myVar5 est de type chaîne de caractères alors que myVar6 est de type booléen.

En Java il est obligatoire de déclarer le type lors de la définition d'une variable. En effet, Java est un langage typé, c'est-à-dire que le compilateur vérifie la cohérence des types des valeurs avant de compiler le code. C'est la raison pour laquelle le type de chaque variable doit être connu à l'avance. En revanche, Java permet de déclarer une variable sans l'assigner une valeur. C'est le cas de la variable myVar7 dans l'exemple ci-dessous. Il s'agit d'une simple déclaration. La valeur pourra être assignée plus tard dans le programme.

Notons par ailleurs, qu'il est possible de déclarer une variable et de lui assigner une valeur nulle. C'est le cas de la variable myVar8 dont la valeur est fixée à nulle (voir exemple ci-dessous).

En définitive, une variable est caractérisée aussi bien par son nom et son type mais également par la cohérence entre le type déclaré et la valeur assignée.

3.3.1 Interdire la modification d'une variable : usage du mot-clé final

Lorsqu'une variable est déclarée, il est possible de lui assigner des valeurs et de modifier et réassigner de nouvelles valeurs autant de fois qu'on souhaite dans le programme. Mais dans certaines situations, on souhaite que la valeur d'une variable reste figée et qu'il ne soit pas possible de modifier sa valeur dès qu'elle est initialisée pour la première fois. Java offre cette possibilité avec le mot-clé final. L'exemple ci-dessous montre le cas d'utilisation du mot-clé final.

```
final int myVar9=10; // Déclaration et première assignation
myVar9=20 ; // Modification rejetée
```

Dans cet exemple, on déclare la variable myVar9 et on lui assigne la valeur 10. Par la suite, nous souhaitons modifier la variable en lui assignant la valeur 20. Cette modification est tout simplement rejetée car myVar9 a été déclarée avec le qualificateur final.

3.3.2 Règles et conventions de nommage des variables

Il existe quelques règles et conventions applicables au nommage d'une variable en Java. Ci-dessous le rappel de quelques-unes.

Convention de nommage

- Le nom doit commencer par une lettre en minuscule.
- Lorsque le nom d'une variable est formé de mots composés, la première lettre de chaque mot sera écrit en majuscule (à l'exception du premier mot).. Ex : *myVar1, idClient, montantTotal*.
- Le nom d'une variable doit être informatif mais il doit aussi rester le plus court possible. Par exemple pour créer une variable pour le montant de la commande, on peut utiliser soit *montantCmd*, soit mntCommande. On peut aussi utiliser *mntCmd* même si cette spécification est moins intuitive.
- Le nom d'une constante doit être écrite en majuscule de préférence et les mots composés peuvent être éventuellement séparés par le caractère underscore _. Ex : MAX_VALUE.

Règles de nommage

- Le nom d'une variable ne doit contenir aucun caractère spécial. Les noms suivants ne sont pas autorisés : _myVar, $myVar, my.Var, my$Var.
- Le nom d'une variable ne doit pas commencer par un chiffre. Ex : 4myVar

- Le nom d'une variable ne doit pas être un mot réservé du langage Java. : Ex : final, continue, goto, interface, etc.. (voir Tableau 1 ci-dessous pour la liste des mots réservés).
- …

Tableau 1 : Les mots réservés en langage Java

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
extends	final	finally	float	for
goto	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	super	switch
synchronized	this	throw	throws	transient
try	void	volatile	while	

Ces mots réservés représentent les pièces qui forment le socle du langage Java. Ces mots clés ne doivent donc pas être utilisés pour nommer des identificateurs (variables ou objets) au risque d'avoir de mauvaises surprises.

3.4 Type des variables

Les types sont des formats de représentation dans lesquels sont stockées les informations. Java est un langage dit « typé ». Car, il exige qu'on déclare le type de tous les objets manipulés dans le programme : variables et tout autre identificateur.

En Java, on distingue deux catégories de types : les types primitifs et les types non primitifs. Les types primitifs sont des formats basiques universellement reconnus dans lesquels sont stockées les valeurs des variables. Ex : entier, décimal, chaîne de caractères, booléen, etc. Les types non primitifs, quant à eux, sont des types ayant un certain degré d'abstraction par rapport aux types primitifs. Les types non primitifs sont généralement des évolutions des types primitifs. Cette section vise à fournir un aperçu général sur le typage des variables en Java.

3.4.1 Les types primitifs

On distingue quatre groupes de types primitifs :

- ✓ les nombres entiers : byte, short,int et long
- ✓ les nombres flottants : float, double
- ✓ les caractères : char

- ✓ les booléens : boolean

Le tableau ci-dessous fournit les détails sur chaque type primitif.

Tableau 2: Les types primitifs Java

Type	Description	Taille en octets (en bits)	Valeur minimum	Valeur maximum	Classe Enveloppe (wrapper)
byte	Octet	1 (8 bits)	-128	127	Byte
short	Entier court	2 (16 bits)	-32 768	32 767	Short
int	Entier	4(32 bits)	-2 147 483 648	2 147 483 647	Integer
long	Entier long	8(64 bits)	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	Long
float	Décimal flottant à simple précision	4(32 bits IEEE 754 floating point)	-1.40239846E-45	3.40282347E38	Float
double	Décimal flottant à double précision	8(64 bits IEEE 754 floating point)	4.9406564584124654E-324	1.797693134862316E308	Double
char	Caractère	2 (16 bits)	0	65 536	Character
boolean	Booléen	Indéterminé			Boolean

Même si les types primitifs ne sont pas des classes, mais Java offre la possibilité de les utiliser sous forme de classes et d'instancier des types primitifs à partir de ces classes. On les appelle les classes enveloppes (wrapper classes). La dernière colonne du tableau indique la classe enveloppe correspondant à chaque type primitif.
L'avantage des classes enveloppes est qu'elles permettent de définir des variables typées sous forme de classe et de réaliser certaines opérations plus avancées parfois impossibles avec une variable typée sous sa forme primitive. Dans l'exemple ci-après, la variable m offre plus de possibilité de traitement et de manipulation que la variable k.

```
int k= 100;
Integer m=new Integer("100");
```

3.4.2 Les types non primitifs (types références ou types classes)

A la différence des types primitifs qui sont des formats bruts de stockage d'information, les types non primitifs sont plutôt des références. Ce sont de types plus évolués construits au-dessus des types primitifs. Par exemples les classes enveloppes associées aux types primitifs sont des cas particuliers des types non primitifs. Car elles permettent de présenter les types primitifs sous une forme plus évoluée.

Les types non primitifs sont généralement des classes ou des références de classes. A noter que les types non-primitifs sont toujours adossés à un type primitif. Cette section vise à présenter quelques types non-primitifs du langage Java. Plus particulièrement nous présentons le type String et le type Array qui sont les types non primitifs standards les plus couramment utilisés.

3.4.2.1 Le type String

Définir une variable de type String

Le type String est un type non primitif destiné à stocker les chaînes de caractères. A la différence du type primitif char, le type String est une classe. Il ne faut pas confondre le format caractère représenté par le type char et le format chaîne de caractères représenté par le type String. En effet, le type char sert à représenter un seul caractère alors que le type String peut représenter une séquence de caractères. Il peut contenir représenter zéro ou plusieurs caractères.

Le type String est représenté par la classe String de l'API Java. Ainsi pour créer une variable de type String, il suffit simplement d'appeler cette classe String() en lui passant la valeur de la chaîne de caractères que vous souhaitez représenter. Les exemples ci-dessous fournissent des illustrations.

```
String myVar10= new String ("Ceci est une chaîne de caractères");
String myVar11= new String (""); /* Définit une chaîne vide */
String myVar12= new String(); /* Définit également une chaîne vide */
String myVar13= null; /* Définit également une chaîne vide */
```

Cet exemple fournit trois cas de création de variables de type String. myVar10 est une variable de type String dont la valeur est « *Ceci est une chaîne de caractères* ». myVar11 et myVar12 sont également des variables de type String à la seule différence que leur valeur est une chaîne de caractères vide. **Attention** à ne pas confondre un String vide et un String de valeur nulle. En effet, une chaîne de caractères vide est une séquence de 0 caractère, c'est-à-dire de longueur 0. Alors qu'une chaîne de caractères nulle est une séquence de caractères qui n'a pas d'existence matérielle. Elle est représentée par la valeur null (voir la variable 13 dans l'exemple).

Concaténer deux ou plusieurs chaînes de caractères

Il est possible d'« additionner » deux ou plusieurs chaînes de caractères pour former une seule chaîne. Cette opération s'appelle concaténation. C'est l'opérateur « + » qui permet de concaténer deux ou plusieurs chaînes de caractères. L'exemple ci-dessous fournit une illustration.

```
String mot= new String ("Bonjour");
String prenom= new String ("Christine");
String nom= new String ("Latour");
String espace = new String (" ");
String salutation= mot+espace+prenom+espace+nom;
System.out.println(salutation);
```

En exécutant ce bout de code la valeur affichée de la variable salutation est :

```
Bonjour Christine Latour
```

Les principales méthodes de la classe String

La Classe String offre plusieurs méthodes permettant le traitement des chaînes de caractères. Par exemple la méthode length() renvoie le nombre total de caractères qui forment une chaînes (espace compris). La méthode toLowerCase() convertit toute la chaîne

de caractères en minuscule. Et la méthode toUpperCase() convertit toute la chaîne en majuscule. Voir exemple ci-dessous.

```
String salutation= new String ("Bonjour Madame Christine Latour");
System.out.println(salutation.length());
System.out.println(salutation.toLowerCase());
System.out.println(salutation.toUpperCase());
```

L'exécution de ce bout de code renvoie :

```
31
bonjour madame christine latour
BONJOUR MADAME CHRISTINE LATOUR
```

L'ensemble des méthodes fournies par la classe String sont consultables à ce lien : https://docs.oracle.com/javase/10/docs/api/java/lang/String.html

Utilisation de la classe StringBuffer

Il faut faire remarquer qu'une valeur String n'est pas modifiable. On dit que le String est immuable. En effet, chaque fois qu'une variable de type String est modifiée, Java crée automatiquement une nouvelle instance de la classe String pour recevoir la valeur modifiée. Même si cette nouvelle instance prend le même nom que l'ancienne variable, la valeur initiale est reste intacte. Java offre néanmoins une classe spécifique de traitement de chaînes qui offre la possibilité de modifier la valeur d'une chaîne. Il s'agit de la classe *StringBuffer*. La classe StringBuffer découpe la chaîne d'origine en une séquence de caractères unitaire. Chaque caractère étant alors identifiée par sa position dans cette séquence. Grâce à cette disposition des caractères, il est alors possible de modifier une chaîne d'origine : ajouter ou retirer des caractères à la séquence. Avec la classe StringBuffer on peut par exemple transformer le mot « Attention » en mot « Intention ». Pour cela, il faut remplacer le premier caractère par i et le second caractère par n en utilisant la méthode setCharAt(). Ex :

```
String string1 = "Attention" ;
System.out.println (string1) ;
StringBuffer myBuffer = new StringBuffer (string1) ;
myBuffer.setCharAt (0, 'I');
myBuffer.setCharAt (1, 'n');
String string2= myBuffer.toString();
System.out.println (string2) ;
```

```
Attention
Intention
```

Dans cet exemple, on utilise la méthode setCharAt() pour déposer le caractère « I » à la première position (position 0) où se trouve déjà le caractère A. Ensuite, on dépose le caractère « n » à la deuxième position où se trouvait le premier caractère « t». Après cette transformation, on regroupe l'ensemble des caractères de la séquence pour reconstituer la valeur String en utilisant la méthode toString().

La classe StringBuffer offre plusieurs autres méthodes de traitement. Les plus couramment utilisées en plus de setCharAt() sont insert() et append(). La méthode insert() permet d'insérer un caractère ou un bout de chaîne de caractères dans un buffer à une position bien spécifiée. Quant à la méthode append, elle permet d'ajouter un caractère ou un bout de chaîne de caractères en fin d'un buffer. L'exemple ci-dessous montre l'utilisation de ces deux méthodes.

```
String s1 = "Bonjour Latour" ;
System.out.println (s1) ;
StringBuffer myBuffer = new StringBuffer (s1) ;
myBuffer.insert (8, new StringBuffer("Madame "));
String s2= myBuffer.toString();
System.out.println (s2);
myBuffer.append (", comment allez vous ?");;
String s3= myBuffer.toString();
System.out.println (s3) ;
```

```
Bonjour Latour
Bonjour Madame Latour
Bonjour Madame Latour, comment allez vous ?
```

Dans cet exemple, la chaîne de caractères en entrée est s1 dont la valeur est « Bonjour Latour ». Nous avons utilisé la méthode *insert()* pour ajouter la chaîne de caractères « Madame ». Cette chaîne est insérée à la position 8 (position initialement occupée par la lettre L). L'ajout de cette chaîne modifie le buffer initial. Nous récupérons cette valeur modifiée sous forme de String dans la variable s2. Ensuite, nous continuons toujours de modifier le buffer en ajoutant une nouvelle chaîne. Mais cette fois en mode *append()*. Il s'agit de la phrase *«, comment allez vous ?* » A remarquer que lorsqu'on utilise la méthode *append()*, il n'est pas nécessaire de convertir en buffer le bout de chaîne à ajouter. Mais cela peut être obligatoire pour la méthode *insert()* lorsque le bout de chaine comporte deux ou plusieurs caractères.

Le StringBuffer offre également d'autres fonctionnalités pour modifier les chaînes de caractères qui, autrement, n'auraient pas été possible avec la classe String. La page ci-dessous fournit plus de détails sur la classe StrinBuffer : https://docs.oracle.com/javase/10/docs/api/java/lang/StringBuffer.html

Utilisation de la classe StringBuilder

La classe StringBuilder est une autre classe permettant la manipulation et le traitement des valeurs de type String. La classe StringBuffer est la version asynchrone de la classe StringBuffer. En effet, la classe StringBuffer traite les valeurs Strings de manière synchronisée. Le traitement synchronisé des données est souvent utile dans les environnements multithreads. Mais parfois le prix à payer de la synchronisation est la baisse de performance. Lorsque nous n'avons pas besoin d'un traitement synchronisé, la classe StringBuilder apporte plus de performance que la classe StringBuffer. L'exemple ci-dessous illustre l'utilisation de la classe StringBuilder.

```
String sb1 = "Bonjour Latour" ;
System.out.println (sb1) ;
StringBuilder myBuilder = new StringBuilder(sb1);
myBuilder.insert (8, "Madame ");
String sb2= myBuilder.toString();
System.out.println (sb2);
myBuilder.append (", comment allez vous ?");;
String sb3= myBuilder.toString();
System.out.println (sb3) ;
```

Comme on peut le constater la classe StringBuilder fonctionne sur le même principe que la classe StringBuffer. La méthode *insert()* permet d'ajouter une chaîne de caractères à une position donnée dans la chaîne en entrée. Dans l'exemple ici, nous avons ajouté le String « *Madame* ». Nous avons aussi utilisé la méthode *append()* qui ajoute un String à la fin de la chaîne en entrée. Ici nous avons ajouté la chaîne de caractères *« , comment allez vous ?* ». Remarquons aussi que pour renvoyer le String retraité, il suffit d'utiliser la méthode *toString()* sur l'objet StringBuilder tout comme on l'a fait pour l'objet StringBuffer.

La classe *StringBuilder* offre également plusieurs méthodes de traitement de chaînes de caractères. Les détails sur chacune de ces méthodes se trouvent sur cette page :

https://docs.oracle.com/javase/10/docs/api/java/lang/StringBuilder.html

3.4.2.2 Le Type Array (le format tableau)

Le type Array est un type de deuxième degré qui représente sous forme de tableau une séquence de valeurs de même type (type primitif ou type classe).

Définir une variable de type Array

Une variable de type Array se définit en deux temps. D'abord on déclare la variable en spécifiant son nom et sa dimension. Ensuite, on assigne les valeurs.

Déclaration de la variable

A la différence du type String, le type Array n'est pas obtenu en instanciant une classe spécifique. Pour déclarer une variable de type Array, il suffit d'indiquer les symboles des crochets [ ] précédés par les types des valeurs qu'il va recevoir. Les exemples ci-dessous illustrent quelques cas de définition de variables de type Array.

```
String[] noms = new String[5]; // un Array de String à 5 éléments
int[] numeros = new int[5]; // Array de type int à 5 éléments
float[] poids = new float[5]; // Array de type float à 5 éléments
```

Dans l'exemple ci-dessous, on définit trois variables noms, numeros et poids qui sont toutes de type Array. La première variable contient 5 éléments de type String. La deuxième contient des éléments de type int et la troisième des éléments de type de float. Notons que la dimension d'un Array est fixée lors de la déclaration. Une fois définie, cette dimension, n'est plus modifiable car les Arrays tout comme les Strings sont des objets immuables.

Assignation des valeurs

La dimension et les types des éléments étant définis, on peut maintenant assigner les valeurs des éléments. Il faut savoir que pour accéder à un élément d'un Array, il faut indiquer la position de cet élément appelé indice. En effet, l'Array se présente comme une séquence de valeurs où chaque élément est identifiable par son indice. Le premier élément est identifié par l'indice 0 et le dernier élément est identifié par l'indice n-1 où n est la dimension de l'Array.

Les exemples ci-dessous initialisent les variables précédemment déclarées.

```
// Assigner les valeurs de la variable noms
noms[0] = "Alex" ; // Assigne la valeur à l'indice 0 ( premier élément)
noms[1] = "Florent" ;
noms[2] = "Khalil" ;
noms[3] = "Ismael" ;
noms[4] = "Jonathan" ;
// Assigner les valeurs de la variable numeros
numero[0]=2367;
numero[3]= 9073;
numero[4]=1423;
// Assigner les valeurs de la variable numeros
poids[1] = 72.5F;
poids[3] = 80.0F;
```

Comme on peut le constater le type Array offre plus de flexibilité, car il n'exige pas à ce que tous les éléments de la séquence soient assignés et connus d'un coup. On peut les définir au fur et à mesure. C'est le cas de la variable numeros et de la variable poids où les éléments n'ont pas été définis. Mais l'espace reste disponible pour les autres éléments et ils pourront être définis à n'importe quel moment dans le programme.

A noter qu'on peut aussi déclarer et assigner les valeurs d'un Array dans la même instruction. Les exemples ci-dessous sont des illustrations de cette approche.

```
String[] noms = {"Alex" , "Florent" , "Khalil" , "Ismael" ,"Jonathan" };
int[] numero= {2367, 5637, 8495, 9073, 1423};
float[] poids = {60.4F, 72.5F, 102.8F, 80.0F, 61.90F};
```

Contrairement à l'approche de définition en deux étapes, dans cette nouvelle approche, tous les éléments doivent être connus dès la définition de la variable.

Accéder aux éléments

Pour accéder à un élément d'un Array, il suffit d'indiquer l'indice de cet élément. L'exemple ci-dessous illustre la récupération des valeurs des éléments à partir d'un Array.

```
String name= noms[3] ; // Récupère Khalil et stocke dans la variable name
int num= numero[4] ; // Récupère 1423 et stocke dans la variable n
float p=poids[0] ;// Récupère 60.4F et stocke dans la variable p
```

Notons aussi qu'il est possible d'élaborer une boucle pour accéder à chaque élément d'Array. Par exemple, pour la variable noms, on peut écrire comme suit :

```
for (int nom: noms) {
   System.out.println(nom);
}
```

Mais on peut aussi afficher l'Array en utilisant une librairie spécialisée de l'API Java comme illustré sur l'exemple ci-dessous.

```
import java.util.Arrays; // Importer la librairie Arrays.
System.out.println(Arrays.toString(noms));
```

3.4.2.3 Le type matrice (Array multidimensionnel)

En Java, une variable de type matrice s'obtient en déclarant un type Array à plusieurs dimensions. Les types Arrays que nous venons d'étudier sont toutes de type Array unidimensionnel. Mais Java offre aussi la possibilité de créer des Arrays à plusieurs dimensions encore appelés Arrays multidimensionnels. Les Arrays multidimensionnels sont des Arrays dont les éléments sont également des Arrays. Par exemple pour déclarer un Array à deux dimensions on utilise le symbole [ ] [ ]. L'exemple ci-dessous illustre la définition d'une variable de type Array multidimensionnel.

```
int[] score_joueur1 = {5, 7, 6, 4};
int[] score_joueur2 = {1, 3, 1, 8};
int[] score_joueur3 = {9, 2, 3, 5};
//array de arrays
int[][] infos_joueurs= { score_joueur1, score_joueur2, score_joueur3};
```

Tout comme l'Array unidimensionnel, tous les éléments constituant une matrice doivent également de même type (qui peut être primitif ou de type classe).

Aussi, tout comme pour les Arrays unidimensionnels, on accède aux éléments d'un Array multidimensionnel en parcourant les éléments par leur indice. Cependant les éléments de premier niveau étant eux-mêmes des Arrays, il faut élaborer une boucle pour accéder aux éléments singuliers de l'Array. Une première boucle pour parcourir les Arrays suivant leur indice. Et une deuxième boucle pour parcourir les éléments individuels constituant chaque Arrays. L'exemple ci-dessous illustre comment retrouver les éléments individuels de la matrice précédemment définie.

```
for (int [] joueur: infos_joueurs) {
    for (int score: joueur) {
        System.out.println(score);
    }
}
```

3.4.2.4 Le type Enum

Définir une variable de type Enum

Comme son nom l'indique, le type Enum est un objet qui permet d'énumérer un certain nombre de valeurs représentées sous formes de séquences. En réalité Enum n'est pas un type au sens proprement parler. Il s'agit en fait d'une séquence prédéfinie de valeurs ayant le même type (primitif ou non). Les valeurs de cette séquence peuvent être appelées à tout moment dans le programme selon les besoins.

A la différence des Arrays (matrices) pour lesquels le nombre d'éléments peut être élevé, les Enum ont un nombre très limité de valeurs. Par exemple, on peut construire un objet Enum pour les 7 jours de la semaine ou les 12 mois de l'année. Les Enums peuvent être définies pour d'autres séquences de valeurs comme le niveau de satisfaction d'un client dont les valeurs peuvent être : « non satisfait », « moyenne satisfait » et « très satisfait ». L'exemple ci-dessous illustre la définition de quelques variables de type Enum.

```
enum Chemise { S, M, L, XL, XXL }
enum Couleur {  BLEU, BLANC , ROUGE, JAUNE, VERT}
enum Semaine { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE };
```

Quelques remarques sont à faire au sujet de la définition des variables de type Enum

- A la différence des variables standards, les variables Enum sont nommées avec le premier caractère en majuscule. En fait les Enums sont plutôt des objets de type Classe. Pour ce faire on adopte la même règle de nommage que pour la classe.
- Aussi les valeurs des Enums sont assignées sans utiliser le symbole « = ».
- Les valeurs des Enum sont indiquées sans les quotes même s'il s'agit des valeurs alphanumériques.

Accéder aux valeurs d'une variable Enum

Etant donné que les Enums se comportent comme une classe, l'accès à ses éléments se fait comme l'accès à un attribut global ou une attribut d'une classe static. Il suffit simplement d'utiliser le nom de l'Enum et la valeur souhaitée en utilisant le symbole « . ». L'exemple ci-dessous illustre l'accès aux valeurs d'une Enum.

```
Semaine jour = Semaine.JEUDI ; // Renvoie Jeudi
Chemise taille = Chemise.L ; // Renvoie L
Couleur favori=Couleur.BLEU; // Renvoie BLEU
```

Java offre plusieurs méthodes pour exploiter les variables de type Enum. Consulter la page ci-dessous. https://docs.oracle.com/javase/10/docs/api/java/lang/Enum.html

3.4.3 Conversion de type

En Java, pour convertir une variable d'un type A en un type B lorsque le type A est compatible avec le type B, on utilise l'opérateur de cast symbolisé par les parenthèses (). Même si le principe de cast reste le même pour les variables de type primitifs et les variables de type non primitifs, pour des raisons de pédagogie, il est judicieux de distinguer les deux cas.

3.4.3.1 Conversion des types primitifs

Comme indiqué précédemment, pour convertir un type primitif en un autre type, on utilise l'opérateur de cast (). L'exemple ci-dessous montre plusieurs cas de conversion de types primitifs.

```
int myIntVar= 200;
long myLongVar= 24353738398L;
// Conversion explicite
float myFloatVar1= (float) myIntVar; // Cast du type int en float
double myDoubleVar1= (double) myLongVar; // Cast du long int en double
double myDoubleVar2= (double) (myLongVar/myIntVar); // Convertit du rapport
// Conversion implicite
float myFloatVar2=  myIntVar; // Cast implicite du type int en float
double myDoubleVar3=  myLongVar; // Cast implicite du long int en double
double myDoubleVar4= (myLongVar/myIntVar); // Cast implicite du rapport
```

Dans cet exemple, nous définissons d'abord deux variables *myIntVar* et *myLongVar* qui sont respectivement des variables de type *int* et *long*. Ensuite, nous réalisons quelques opérations de cast. Par exemple la variable *myFloatVar1* est obtenue en convertissant en type *float* la variable *myIntVar*. La variable *myDoubleVar1* est le cast en type *double* de la variable *myLongVar*. Et la variable *myDoubleVar2* est le cast en type *double* de la division entre *myLongVar* et *myIntVar*, qui par défaut aurait été en type *long*. Pour chacun des trois exemples de cast, nous avons effectué ce qu'on appelle un cast explicite, car nous avons explicitement indiqué l'opérateur cast (). Mais Java offre la possibilité d'effectuer un cast implicite. C'est-à-dire sans spécifier l'opérateur (). C'est le cas par exemples des variables *myFloatVar2* (qui convertit le type *int* en type *float*), *myDoubleVar3*(qui convertit le type *long* en type *double*) et la variable *myDoubleVar4* (qui convertit le rapport entre le type *long* et le type *int* en type *double*).

Signalons cependant que le cast de type doit être utilisé avec beaucoup de précaution car il peut générer de l'imprécision dans le résultat de sortie, surtout dans les cas où le type de sortie est d'un rang plus faible que le type d'entrée. Par exemple, lorsqu'on convertit un type *double* (qui est un décimal en double précision) en type *byte*, il est clair qu'il y aura perte de précision très significative. Dès lors, pour réaliser un cast, qu'il soit implicite ou explicite, il faut s'assurer qu'un certain nombre de conditions de sécurité soient respectées. La première condition est que le type en sortie doit être plus général que le type en entrée. Pour les types primitifs, l'ordre croissant des types est définit comme suit : *byte*, *short*, *int*, *char*,

long, *float*, *double*. Le type *booléen* est un type spécial qui n'est pas compatible avec les autres types (voir Tableau 2 pour les détails sur les types).

En règle générale, pour qu'un cast soit possible il faut que le type d'entrée et le type de sortie soient compatibles. Et lorsque les deux types sont compatibles, il faut s'assurer que le cast se fait d'un type moins général vers un type plus général. Dans le cas contraire, c'est-à-dire le cast d'un type général vers un type moins général, il faut s'attendre à une perte de précision, parfois même à une troncature lorsque les types sont trop éloignés.

3.4.3.2 Conversion des types non primitifs

Comme nous l'avons évoqué plus haut, les types non primitifs sont des classes. A ce stade du document, nous n'allons pas développer encore les notions de classes afin de pouvoir illustrer les opérations de cast sur des objets. Des sections sont prévues à cet effet. Ici, nous allons seulement nous limiter à présenter les principes de base de la conversion des variables de type non primitif appuyées par des exemples conceptuels.

Pour rappel, les principes de cast applicables pour les variables de types non primitifs (encore appelés types références) sont les mêmes que ceux applicables aux variables de types primitifs. En effet, pour convertir un type A vers un type B, il faut d'abord que le type A et le type B soient compatibles. Dans le cas des types références, pour garantir la compatibilité entre le type A (classe A) avec le type B (classe B), il faut qu'il existe une relation d'héritage entre les deux classes. Par exemple, la classe B doit hériter de la classe A et vice-versa. Pour illustrer ces propos, supposons par exemple une classe appelée Fleur() et une classe appelée Rose(). Ces deux classes sont compatibles car la classe Rose() est une sous-classe de la classe Fleur(). Dans ce cas, il est possible d'appliquer l'opérateur cast sur un objet de la classe Rose() pour le convertir en un objet de type Fleur(). Ce cas est illustré dans l'exemple ci-dessous.

```
class Fleur { String racine; String tige; String feuille;};
class Rose extends Fleur { String couleur="Rouge vif";};

Rose maRose=new Rose() ;// Crée un objet à partir de la classe Rose
Fleur maFleur =(Fleur) maRose ; // Convetir l'objet en type Fleur
```

Dans cet exemple, on convertit un objet de type particulier (Rose) en un objet de type plus général (Fleur). Etant donné qu'il y a une relation d'héritage entre les deux objets, on pouvait effectuer la conversion de manière implicite, c'est-à-dire sans spécifier l'opérateur de cast (). Le cast implicite se présente alors comme suit :

```
Fleur maFleur =maRose ; // Cast implicite de l'objet maRose en type Fleur
```

Notons que lorsque l'on convertit un objet de type particulier vers un objet de type général comme cela est le cas ici, l'objet convertit obtenu en sortie perd effectivement ses particularités. Par exemple, en convertissant l'objet *maRose* en objet de type *Fleur*, on perd le champ couleur, qui n'est pas un attribut défini dans la classe *Fleur()*. L'attribut couleur

est en effet spécifique à la classe *Rose()*, qui, on le rappelle, est une sous-classe des fleurs. Cela est matérialisé par le qualificateur *extends* dans sa définition.

Notons aussi que dans certaines rares situations peuvent vous amener à convertir un objet d'un type plus général en un objet de type particulier. Dans ce cas, le cast n'est pas toujours possible même si les deux objets sont compatibles. Par exemple, on peut disposer d'un objet de type Fleur() et on veut le caster en type Tulipe() qui est une sous-classe particulière de fleur. Cette conversion peut se faire comme suit.

```
class Fleur{ String racine; String tige; String feuille;};
class Tulipe extends Fleur{ String couleur="blanche";};

Fleur maFleur=new Fleur() ;// Créer un objet de classe Fleur
Tulipe maTulipe =(Tulipe) maFleur ; // Cast en type Tulipe
```

Ce cast n'est pas possible car vouloir ramener un objet de type Fleur en un objet de type Tulipe signifie qu'on personnalise le type Fleur en le réduisant en type Tulipe en identifiant même sa couleur. Ce qui n'est pas possible.

En somme, pour la conversion des types référence, il est fortement recommandé de ne convertir que les types particuliers en des types généraux et non l'inverse.

3.5 Les opérateurs Java

Cette section présente les différents types d'opérateurs utilisés en langage Java.

3.5.1 Les opérateurs arithmétiques

Le tableau ci-dessous présente les opérateurs arithmétiques standards.

Tableau 3: Les opérateurs arithmétiques

Opérateur	Description	Exemple d'usage	Commentaire Résultat
Opérateurs arithmétiques standards			
+	Addition	`int x= 9; int y=3; int z= x+y;`	z=12
-	soustraction	`int x= 9; int y=3; int z= x-y;`	z=6
*	Multiplication	`int x= 9; int y=3; int z= x*y;`	z=27
/	Division	`int x= 9; int y=3; int z= x/y;`	z=3
%	Modulo	`int x= 9; int y=3; int z= x%y;`	z=0

3.5.2 Les opérateurs unaires

Les opérateurs unaires sont un ensemble d'opérateurs (arithmétiques ou autres) qui s'appliquent à un seul opérande. Le tableau 4 ci-dessous liste les opérateurs unaires.

Tableau 4: Les opérateurs unaires

Opérateur	Description	Exemple d'usage	Commentaire Résultat
-	Négativation	`int x = 200; int y = -x;`	*y=-200*
+	Positivation	`int x = 200; int y = +x;`	*y=200*
++	Incrémentation	`int x = 6; int y=++x;` `int x = 6; int y=x++`	*x=7 , y=7* *x=7, y=6*
--	Décrémentation	`int x = 6; int y= --x;` `int x = 6; int y=x--;`	*x=5, y=5* *x=5, y=6*
!	Complément booléen	`boolean x = true;` `boolean y = !x;`	x=true, y=false

3.5.3 Les opérateurs relationnels

Les opérateurs relationnels sont des opérateurs (arithmétiques ou autres) qui permettent de comparer la valeur de deux opérandes. Le tableau 5 ci-dessous fournit la liste des opérateurs relationnels.

Tableau 5: Les opérateurs relationnels

Opérateur	Description	Exemple d'usage	Commentaire Résultat
==	Egal à	`int x= 5; int y=5; boolean z=x==y`	z=true
<	Strictement inférieur à	`int x= 7; int y=8; boolean z=x<y;`	z=true
<=	Inférieur ou égale à	`int x= 9; int y=3; boolean z=x<=y;`	z=false
>	Strictement supérieur à	`int x= 3; int y=7; boolean z=x>y;`	z=false
>=	Supérieur ou égale à	`int x= 9; int y=3; boolean z=x>=y;`	z=true
!=	Différent de	`int x= 9; int y=3; boolean z=x !=y;`	z=true

3.5.4 Les opérateurs conditionnels

Les opérateurs conditionnels permettent vérifier une succession de conditions. Le tableau 6 ci-dessous illustre les opérateurs conditionnels ainsi que leur mode d'utilisation.

Tableau 6: Les opérateurs conditionnels

Opérateur	Description	Exemple d'usage	Commentaire Résultat
&	Et (joint) : vérifie les deux conditions	`int x= 9; int y=3; boolean z= (x<10 & y>0);`	z=true
&&	Et (disjoint) : Ne vérifie pas la deuxième condition si la première n'est pas satisfaite	`int x= 9; int y=3; boolean z= (x>10 && y>0);`	z=false
|	Ou (joint) : vérifie les deux conditions	`int x= 9; int y=3; boolean z= (x<10 | y>0);`	z=true
||	Ou (disjoint) : Ne vérifie pas la deuxième condition si la première est déjà satisfaite	`int x= 9; int y=3; boolean z= (x>10 || y>0);`	z=false

3.5.5 Les opérateurs d'assignation

Un opérateur d'assignation sert à définir la valeur d'une variable. En Java, l'opérateur d'assignation de base reste le symbole « = ». Mais il existe d'autres variantes de cet opérateur qui sont en fait des opérateurs d'assignation dynamiques combinant d'autres types d'opérateurs. Le tableau 7 ci-dessous présente les opérateurs d'assignations les plus couramment utilisés.

Tableau 7: Les opérateurs d'assignation

Opérateur	Description	Exemple d'usage	Commentaire Résultat
=	Assignation directe	`int x= 9;`	
+=	Addition incrémentale	`int x= 9; x+=5; // Equivalent à x=x+5`	x=14
-=	Soustraction incrémentale	`int x= 9; x-=5; // Equivalent à x=x-5`	x=4
*=	Multiplication incrémentale	`int x= 9; x*=5; // Equivalent à x=x*5`	x=45
/=	Division incrémentale	`int x= 10; x/=5; // Equivalent à x=x/5`	x=2
%=	Modulo incrémental	`int x= 10; x%=5; // Equivalent à x=x%5`	x=0

3.5.6 Les opérateurs sur bits (Bitwise operators)

Les Bitwise operators servent à la manipulation de bits individuels d'un nombre. Ils sont généralement utilisés lors de l'exécution d'opérations de mise à jour et de requêtage des

arborescences binaires indexées. Le tableau 8 ci-dessous montre les Bitwise operators les plus couramment utilisés.

Tableau 8 : Les Bitwise operators (opérateurs sur bits)

Opérateur	Description	Exemple d'usage	Commentaire Résultat
&	Et Bitwise	`int x = 5;int y = 7;` `int z=x & y;`	z=5
|	Ou Bitwise	`int x = 5;int y = 7;` `int z=x | y;`	z=7
^	Ou exclusif (XOR)	`int x = 5;int y = 7;` `int z=x ^ y;`	z=2
~	Complément bitwise (valeur inverse des bits)	`int x= 10;` `int y=~x;`	x=10, y=-11

3.5.7 Les opérateurs de décalage de bits (Shift opérators)

Les Shift operators (opérateurs de décalage) sont des opérateurs permettant de manipuler les bits. On distingue trois principaux shift operators. Les détails de chacun des opérateurs sont présentés dans le tableau 9 ci-dessous.

Tableau 9: Shift operators (opérateurs de décalage)

Opérateur	Description	Exemple d'usage	Commentaire Résultat
<<	Décale les bits à gauche	`byte x = 32; byte y = (byte) (x << 1);`	y= 64
>>	Décale les bits à droite	`byte x = 32; byte y = (byte) (x >> 2);`	y= 8
>>>	Décale les bits à droite en remplissant les positions restées vides par 0	`byte x = 32; byte y = (byte) (x >>> 2);`	x=8

3.6 Le structures de contrôle

Les structures de contrôle sont des expressions syntaxiques qui permettent d'exécuter des bloc d'instructions soit de type conditionnelle (*if.. else*), de type boucle (*for, do.. while*) ou de type choix multiples (*switch*). Cette section présente les syntaxes et les cas d'utilisation des structures de contrôle en langage Java.

3.6.1 Les structures conditionnelles : if...else

Un bloc d'instructions conditionnelles est un bloc dans lequel les instructions à exécuter sont définies suivant des conditions. On peut distinguer trois formes de bloc d'instructions conditionnelles : les structures à deux conditions, les structures à plusieurs conditions et les structures conditionnelles imbriquées.

3.6.1.1 Structure à deux conditions

La syntaxe de base d'un bloc d'instructions conditionnelles à deux conditions se présente comme suit :

```
if (condition de base) {
    instructions si la condition est satisfaite ;
} else{
    instructions si la condition est non satisfaite ;
}
```

L'instruction *if (condition de base)* définit la condition de base tandis que l'instruction *else* représente la condition alternative à la condition de base.

L'exemple ci-dessous vérifie si un nombre est pair ou impair et affiche un message différent pour chaque situation.

```
int j=20;
if(j%2==0){
    System.out.println("j est un nombre pair");
}else{
    System.out.println("j est un nombre impair");
}
```

On a d'abord défini une variable j dont la valeur est 20. Pour savoir si un nombre est pair, le reste de sa division par 2 doit être égal à 0. La condition servant dont à définir la structure de contrôle est donc *j%2==0*. Et *if(j%2==0)* renvoie une valeur booléenne *true* lorsque la condition est satisfaite et *false* lorsque la condition n'est pas satisfaite. Et une instruction est exécutée suivant chacune de ces valeurs. Dans cet exemple, la condition est satisfaite, du coup le message affiché sera : « *j est un nombre pair* ».

3.6.1.2 Structure à plusieurs conditions

Notons qu'un bloc d'instructions conditionnelles permet d'ajouter autant de conditions alternatives intermédiaires qu'on souhaite. Pour ajouter une condition alternative intermédiaire, il suffit d'ajouter une instruction *else if (condition alternative)*. La syntaxe ci-dessous définie représente la forme générale d'un bloc d'instructions conditionnelles.

```
if (condition de base) {
    instructions si condition de base satisfaite ;
}else if (condition alternative 1) {
    instructions si condition alternative 1 satisfaite ;
```

```
}else if (condition alternative 2) {
    instructions si condition alternative 2 satisfaite ;
}else if (...) {
            ... ;
}else if (condition alternative n) {
    instructions si condition alternative n satisfaite ;
}else{
    instructions pour tous les autres cas restants ;
}
```

L'exemple ci-dessous vérifie si un nombre est positif, négatif ou égal à 0. Pour chaque cas une instruction différente est exécutée (ici un message différent).

```
int j=15;
if(j>0){
    System.out.println("Nombre positif");
}else if(j<0){
    System.out.println("Nombre négatif");
}else{
    System.out.println("Zéro");
}
```

Dans cet exemple, il y a une seule condition alternative intermédiaire. Elle est traduite par la condition *else if(j<0)*. Les instructions correspondant à cette condition seront exécutées chaque fois que le nombre fournit en entrée est inférieur à zéro. Dans le cas contraire ce sont les instructions correspondant à d'autres conditions qui seront exécutées. Dans cet exemple, c'est l'instruction de la condition de base qui est exécutée car la valeur de j (15) est supérieure à 0. Le message est donc « Nombre positif »

3.6.1.3 Structures conditionnelles imbriquées

Une structure conditionnelle imbriquée est une forme dans laquelle des blocs d'instructions conditionnelles sont définies à l'intérieur des branches d'autres blocs d'instructions conditionnelles. La syntaxe ci-dessous une forme conditionnelle imbriquée.

```
if (condition de base niveau 1) {
        if (condition de base niveau 2) {
            instructions si condition de base niveau 1 satisfaite ;
        }else{
            instructions si condition de base niveau 2 non satisfaite ;
        }
}else{
    instructions si condition de base niveau 1 non satisfaite ;
}
```

L'exemple ci-dessous montre un cas d'utilisation de la structure conditionnelle imbriquée.

```
int age=20;
int poids=80;
if(age>=18){
    if(poids>50){
        System.out.println("Vous pouvez participer au don de sang");
    }else{
```

```
        System.out.println("Vous ne pouvez pas participer au don de sang");
    }
}else {
    System.out.println("Vous êtes encore mineur");
}
```

Dans cet exemple, on vérifie d'abord l'âge des individus. Ceux qui ont 18 ans ou plus, on vérifie leur poids. Et dans chaque situation, on exécute des instructions. En l'occurrence ici, on se limite à afficher des messages.

Rappelons qu'il est possible d'ajouter des conditions alternatives intermédiaires dans les structures imbriquées aussi bien dans les structures de premier niveau que dans les structures de niveau inférieur. Pour cela, il suffit d'utiliser l'instruction else if (). Par ailleurs, il est possible d'ajouter des niveaux d'imbrications autant que nécessaire.

3.6.2 Les structures itératives (les boucles)

Les structures itératives sont des expressions syntaxiques qui permettent soit d'exécuter plusieurs fois un même bloc d'instructions tant qu'une condition reste vérifiée ou selon une séquence de valeurs prédéfinies. Ces exécutions multiples de blocs d'instruction (itérations) sont appelées couramment des boucles. En Java, on distingue deux formes générales de boucles : les boucle de type *WHILE* et les boucles de type *FOR*. Les boucles *WHILE* exécutent plusieurs fois le même bloc d'instructions tant qu'une condition reste respectée. Et les boucles *FOR* exécutent les mêmes blocs d'instructions suivant une séquence de valeurs connue d'avance. Les sections ci-dessous présentent les détails sur chaque type de boucle et leur mode d'utilisation.

3.6.2.1 Les boucles WHILE

En Java, les boucles de type WHILE peuvent être exprimées dans deux formes syntaxiques différentes. La première est de la forme *while (condition) {instructions ;}* et la seconde est de la forme *do{ instructions} while (condition)* . Ci-dessous la syntaxe générale de chacune des deux formulations.

```
// formulation 1
while (condition) {
    instructions à exécuter tant que condition est vérifiée;
}
// formulation 2
do {
    instructions à exécuter tant que condition est vérifiée;
} while(condition);
```

Dans la première formulation, la condition est d'abord vérifiée avant de lancer la première itération et pouvoir poursuivre les autres itérations. Dans la deuxième formulation, la première itération est d'abord exécutée avant de vérifier la condition et ainsi choisir s'il faut continuer ou pas le reste des itérations.

Cas d'utilisation de la boucle while (condition) {instructions}

```
int i=0;
while(i<=10){
    System.out.println("La valeur de i est "+i);
    i++; // On incrémente la valeur de i.
}
```

Dans cet exemple on continue d'afficher la valeur de la variable i tant que cette valeur est inférieure ou égale à 10. La boucle s'arrête dès que la valeur de i devient supérieure à 10.

Cas d'utilisation de la boucle do {instructions} while (condition)

```
int i=0;
do{
    System.out.println("La valeur de i est "+i);
    i++; // On incrémente la valeur de i.
}while(i<=10);
```

Attention aux cas de boucles infinis

Dans la mise en place de boucle de type WHILE, une attention particulière doit être portée sur la définition des conditions de façon à ne pas générer de boucles infinies (aussi appelées boucles folles). Les boucles folles sont des boucles qui itèrent à l'infini car la condition d'arrêt n'est jamais satisfaite. L'exemple ci-dessous est un cas de boucle folle (Attention à ne pas l'exécuter !!!).

```
int i=0;
do{
    System.out.println("La valeur de i est "+i);
    i++; // On incrémente la valeur de i.
}while(i>0);
```

3.6.2.2 Les boucles FOR

Comme nous l'avons déjà indiqué, les boucles FOR sont des boucles qui exécutent plusieurs fois un même bloc d'instructions suivant une séquence de valeurs connues. La séquence de valeurs peut être une liste dont les valeurs des éléments sont dynamiquement calculées et le nombre d'éléments déterminé suivant une condition d'arrêt. La séquence de valeurs peut également être une liste statique de valeurs se présentant sous forme de Array. La spécification de la boucle FOR diffère légèrement selon qu'on soit dans l'un ou l'autre cas.

Cas d'une séquence dynamiquement calculée

Lorsque la séquence de valeurs est calculée dynamiquement la syntaxe de la boucle FOR se présente comme suit.

```
for (valeurInitiale; ConditionPoursuite; Incrementation) {
    Instructions tant que la condition de poursuite est vérifiée
}
```

Prenons un exemple concret pour illustrer cette syntaxe.

```
for (int i = 1; i <= 10; ++i) {
    System.out.println("La valeur de i est "+i);
}
```

Dans cet exemple la séquence de valeurs est définie de 1 (valeur initiale) à 10 (valeur maximum au-delà de laquelle l'itération s'arrête). Dans chaque itération la valeur de *i* est incrémentée de 1 grâce à l'expression *++i*.

Attention, l'apparition des boucles infinies est également possible dans les types de type FOR. Une situation qui survient lorsque la condition d'arrêt n'est jamais atteinte. Dans l'exemple ci-dessous, on génèrerait une boucle folle si par exemple la condition d'arrêt avait été *i > 0* au lieu de *i <= 10*. Attention, ne pas tester cette éventualité dans votre code.

Cas où la séquence est figée en entrée

Lorsque la séquence de valeurs est figée comme par exemple le cas d'un Array, la syntaxe de la boucle FOR se présente comme suit :

```
for (int i: myArray) {
    System.out.println("La valeur de l'élément est "+e);
}
```

Exemple concret d'illustration

```
int[] mySequence = {8, 2, 9, 6, 12, 4}; // Séquence prédéfinie de valeurs
for (int v: mySequence) {
    System.out.println("La valeur de v est "+v);
}
```

3.6.3 Les instructions *break* et *continue*

Les instructions *break* et *continue* sont deux instructions qui permettent de contrôler les comportements des boucles, qu'il s'agisse des boucles de type *FOR* ou des boucles de type *WHILE*.

L'instruction break permet d'arrêter les itérations même si la condition d'arrêt n'est pas encore atteinte. L'instruction *break* est généralement exécutée lorsqu'une condition intermédiaire spécifiée dans les bloc d'instruction est respectée. Par exemple, dans une boucle de tirage de boules prévu 100 fois, on peut dire d'arrêter les itérations dès qu'on tire la boule noire pour la première. C'est l'instruction *break* qui sert à définir ce genre de critère d'arrêt.

Quant à l'instruction *continue*, elle sert à skipper certaines itérations et à continuer le reste de la boucle jusqu'à atteindre la condition d'arrêt. Une itération est skippée en se basant sur une condition intermédiaire définie dans le bloc des instructions. Par exemple, on souhaite calculer le carré de tous les nombres pairs compris entre 1 et 50. Pour cela, on peut mettre en place une boucle itérative sur la séquence (1-50). Pour chaque nombre dans cette

séquence, on vérifie si le nombre est pair c'est-à-dire si le reste de sa division par 2 est égal à 0. Lorsque cette condition est vérifiée pour un nombre, alors on calcule son carré. Et la condition n'est pas vérifiée pour un nombre, on l'abandonne et on passe au nombre suivant. Tel est le principe sur lequel fonction l'instruction *continue*.

Dans cette section, nous allons présenter les cas d'utilisation des instructions *break* et *continue*.

3.6.3.1 L'instruction break

La syntaxe générale de l'instruction break est la suivante

```
// Cas d'une boucle FOR
for (valeurInitiale; ConditionPoursuite; Incrementation) {

        if (conditionBreak) {
                break; // Arrêt de la boucle si conditionBreak satisfaite
        }

        Instructions tant que la condition de poursuite reste vérifiée
}
// Cas d'une boucle WHILE
while (condition) {
    if (conditionBreak) {
        break; // Arrêt de la boucle si conditionBreak satisfaite
    }
    instructions à exécuter tant que condition est vérifiée;
}
```

Exemples d'illustration

```
// Cas d'une boucle FOR
for (int i = 1; i <= 10; ++i) {
    if (i == 5) {
        break;
    }
    System.out.println("La valeur de i est "+i);
}
// Cas d'une boucle WHILE
while (i<=10) {
    if (i==5) {
        break;
    }
    System.out.println("La valeur de i est "+i);
    i++;
}
```

Cette boucle vise à afficher les différentes valeurs de la variable *i*. Mais dès que la valeur de *i* est égale à 5, on arrête la boucle. Du coup, ce sont seulement les quatre premières itérations de la boucle qui seront exécutées.

3.6.3.2 L'instruction continue

La syntaxe générale de l'instruction continue est la suivante.

```
// Cas d'une boucle FOR
for (valeurInitiale; ConditionPoursuite; Incrementation) {

    if (conditionContinue) {
        continue; // Skip des instructions si conditionContinue satisfaite
    }

    Instructions tant que la condition de poursuite reste vérifiée
}
// Cas d'une boucle WHILE
while (condition) {
    if (conditionContinue) {
        break; // Skip des instructions si conditionContinue satisfaite
    }
    instructions à exécuter tant que condition est vérifiée;
}
```

Exemples d'illustration

```
// Cas d'une boucle FOR
for (int i = 1; i <= 10; ++i) {
    if (i == 5) {
        continue;
    }
    System.out.println("La valeur de i est "+i);
}
// Cas d'une boucle WHILE
int i= 1;
while (i<=10) {
    if (i==5) {
        continue;
    }
    System.out.println("La valeur de i est "+i);
    i++;
}
```

Dans ces exemples, toutes les valeurs de la variable *i* sont affichées sauf la valeur 5 et la boucle est exécutée jusqu'à la condition d'arrêt.

3.6.4 Les structures switch

La structure *switch* est une forme particulière des structures de contrôle qui exécute des blocs d'instructions suivant une liste de valeurs prédéfinies. La structure switch est très proche de la structure IF… ELSE dont elle constitue d'ailleurs une forme optimisée. En effet, la structure IF… ELSE, comme nous l'avons déjà montré précédemment, sert à exécuter des blocs d'instructions suivant une condition de base (*if*), une ou plusieurs conditions alternatives intermédiaires (*else if*) et une condition alternative restante (*else*). Le *switch* adopte le même principe mais les conditions sont définies sur une liste de valeurs prédéfinies. Cette section vise à présenter les structures *switch*.

La syntaxe générale de la structure switch se présente comme suit :

```
switch(expression) {
    case valeur1:
        Instructions à exécuter si expression== valeur1
        break;
    case valeur2:
        Instructions à exécuter si expression== valeur1
        break;

    case valeur...:
        Instructions à exécuter si expression== valeur...
        break;
    case valeurN:
        Instructions à exécuter si expression== valeurN
        break;
    default:
        Instructions à exécuter pour tous les autres cas restants
}
```

Dans cette syntaxe, *expression* représente généralement une variable ayant un nombre limité de valeurs. Et un bloc d'instructions est prévu pour chaque valeur. Lorsqu'on veut exécuter un bloc d'instructions pour une valeur spécifique de variable, on définit ce qu'on appelle un *case*. Il arrive que l'on souhaite définir des instructions spécifiques que pour seulement quelques *cases*. Et si pour les *cases* restants, on souhaite définir des instructions génériques alors ces instructions seront définis au niveau d'un *case* spécial dénommé *default*. A noter que la spécification du *case default* est optionnelle.

Par ailleurs, comme vous pouvez le constater, au niveau de chaque *case* apparaît une instruction *break*. Cette instruction permet l'arrêt des vérifications dès qu'un case est vérifié. En effet, le comportement par défaut de la structure switch est de vérifier toute les cases et de retenir le dernier case qui matche avec la valeur de *expression*. Pour éviter cette longue vérification, on s'arrête dès le premier match. C'est le rôle de l'instruction *break* qui est en fait optionnelle.

L'exemple ci-dessous illustre un cas d'utilisation de la structure *switch*.

```
int jourSemaine = 5;
switch (jourSemaine) {
    case 1:
        System.out.println("Lundi");
        break;
    case 2:
        System.out.println("Mardi");
        break;
    case 3:
        System.out.println("Mercredi");
        break;
    case 4:
        System.out.println("Jeudi");
        break;
    case 5:
        System.out.println("Vendredi");
        break;
    case 6:
```

```
            System.out.println("Samedi");
            break;
        case 7:
            System.out.println("Dimanche");
            break;
        default:
            System.out.println("Ne correspond à aucun jour de la semaine");
}
```

Dans cet exemple, l'expression est représentée par la valeur prise par la variable *jourSemaine*. Etant donné que la valeur de *jourSemaine* est égale à 5, alors ce sont uniquement les instructions définies au niveau du case 5 qui seront exécutées.

4 ETUDE DES CLASSES ET OBJETS JAVA

Dans le chapitre chapitre 1, nous avons déjà présenté et longuement discuté de manière conceptuelle la notion de Programmation Orientée Objet ainsi que des notions relatives aux classes et aux objets[3]. Ce présent chapitre est consacré à l'étude proprement dite des classes et leur instanciation sous formes d'objets en langage Java. Il s'agit en particulier de montrer comment concevoir et utiliser les classes Java pour répondre à un besoin concret.

4.1 Concevoir une classe

D'une manière générale, une classe est une spécification permettant de fournir une représentation générique d'un ensemble de données relatives à une entité dans un système (voir section 1.2.2 pour plus de détails sur la notion d'entité). Mais dans la pratique, dans un programme toutes les classes ne représentent pas nécessairement une entité. On peut aussi concevoir des classes qui ne représentent pas un objet spécifique mais plutôt une séquence de traitements réalisés sur un ensemble de classes. Ce sont par exemples des classes permettant de lancer les méthodes d'autres classes. Il peut s'agir aussi des classes utilitaires permettant de réaliser un certain nombre de traitements génériques. Ces types de classes, on peut les appeler « *Classes de traitement* ». Les classes qui représentent à proprement parler des entités du système modélisé, on les appelle « *classes d'entité* ». Il existe une troisième catégorie de classes représentée par une classe spécifique appelée « *classe Main* ». La classe *Main* est une classe contenant une méthode générique appelée « méthode *main()* » qui sert de point d'entrée à l'exécution de tout le programme Java. Dans cette section, nous présenterons chacun des trois types de classe en montrant leur mode d'utilisation.

4.1.1 Concevoir une classe d'entité

D'une manière générale, une classe représentant une entité est définie avec la syntaxe de déclaration suivante :

Syntaxe : S01

```
package nomPackage;

import nomLibrairies;

class NomClasse{

    /* Déclaration des champs */
    nomChamp1;
    nomChamp2;
    ...
    nomChampN;

    /* Définition des constructeurs */
```

[3] Pour mieux comprendre les notions de Classes et d'Objets ainsi que leur traduction dans la vie réelle, veuillez consulter la section 1.2

```
    constructeur1();
    constructeur2();
    ...
    constructeurN();

    /* Définition des méthodes */
    nomMethode1();
    nomMethode2();
    ...
    nomMethodeN();
}
```

La définition d'une classe commence d'abord par la déclaration d'un nom de package. Le package représente en fait un sous-répertoire ou une succession de sous-répertoires situé dans le dossier parent *src* qui sert à stocker le fichier contenant la définition de la classe. Lorsque le package est constitué d'une succession de sous-répertoires, leurs noms sont séparés par des « . » au lieu des « / ». Ex : *com.tuto* au lieu de *com/tuto*. Nous détaillerons plus tard la notion de package. Pour info, le nom du package doit entièrement être écrit en minuscule.

Après la déclaration du package nous passons à l'import des librairies. Les librairies sont des dépendances dont nous aurons besoin dans l'écriture de notre classe. Une librairie peut être une classe de base fournit nativement par le langage Java. Elle peut être une classe provenant d'une librairie externe déjà disponible dans votre entreprise (librairie interne partagée), ou une librairie externe disponible en open source. La librairie peut être aussi vos propres classes que vous avez écrit dans le même projet Java. Rappelons qu'un projet Java est généralement un assemblage de plusieurs classes. Les classes étant parfois définies dans des fichiers différents ou dans des packages différents, il faut passer par des imports sous formes de librairies pour pouvoir utiliser une classe déjà prête dans la définition d'une autre classe.

Après la définition du package et l'import des librairies nécessaires, on passe à la déclaration du nom de classe. La déclaration est toujours précédée du mot-clé « class ». On pourra aussi ajouter d'autres mots clés appelés *qualificateurs* qui permettent de mieux caractériser la classe. Par exemple, le qualificateur *public*. Permet de rendre la classe publique c'est-à-dire utilisable par des classes définies dans d'autres packages. Nous reviendrons plus tard sur les qualificateurs qui peuvent accompagner la déclaration d'une classe. Rappelons ici que la première lettre du nom d'une classe doit toujours être en lettre majuscule.

Après la déclaration du nom de la classe, nous passons à la définition du corps de la classe. En effet, le corps de la classe comporte trois rubriques : une rubrique consacrée à la déclaration des champs, une consacrée à la définition du/des constructeur(s) et une rubrique consacrée à la définition des méthodes. Pour rappel, les champs représentent les attributs de l'entité représentée par la classe. Par exemple, pour une classe *Employe*, les champs peuvent être l'âge, le sexe, l'ancienneté, le poste, etc.. La première rubrique dans la définition d'une classe consiste donc à déclarer ces champs. Chaque champ doit être déclaré

sous forme de variable en indiquant obligatoirement son type qui peut être un type primitif ou un type référence.

La deuxième rubrique de définition du corps de la classe est la définition du constructeur de la classe. Pour rappel, le constructeur de la classe est une fonction spéciale permettant d'initialiser les valeurs des champs et de créer un objet concret de la classe (voir section 1.2.2 pour plus de détails conceptuels sur le rapport Classe et objet). Dans la définition d'une classe on peut définir un ou plusieurs constructeurs, chacun pouvant avoir un nombre différent de paramètres. Nous reviendrons plus tard sur la définition des constructeurs d'une classe.

La troisième rubrique de la définition du corps d'une classe est la définition des méthodes. Les méthodes sont aussi des fonctions tout comme les constructeurs. Mais à la différence des constructeurs qui initialisent les champs, les méthodes ont pour rôle de modifier les valeurs existantes ou d'y accéder simplement. On distingue deux types de méthodes, celles qui permettent de modifier les valeurs des champs (on les appelle les *setters*) et qui permettent d'accéder aux valeurs et de les renvoyer (on les appelle les *getters*). Les *setters* et les *getters* sont parfois aussi appelés *accessors*.

En somme, définir une classe revient à spécifier son package, son nom, ses qualificateurs, ses champs, ses constructeurs et ses méthodes. Les champs et les méthodes d'une classe sont appelés « *membres de classes* ».

Par ailleurs, la classe, une fois définie, les codes sources de cette définition sont habituellement stockés dans un fichier portant le même nom que la classe. Par exemple lorsque la classe est nommée *MaClasse* alors le fichier contenant ses codes sources est nommé *MaClasse.java*. Mais dans le cas où une classe est définie à l'intérieur d'une autre classe, le fichier garde le nom de la classe parente.

L'étude des classes dans le langage Java dans cette section vise à essentiellement détailler chacun des éléments discuté ci-dessus. Pour entamer cette étude nous commençons d'abord par un exemple introductif de classe représentant un employé d'une entreprise.

Code source : CS01

```
package com.tuto.company.entite;

import java.time.Year;

public class Employe{

    /* Déclaration des champs */
    String nom;
    String sexe;
    int anneeNaissance;
    int age;
    int anneeEmbauche;
    int anciennete;
    double salaire ;

    /* Définition du constructeur */
```

```
    public Employe( String nom, String sexe, int anneeNaissance, int
anneeEmbauche, double salaire ){
        this.nom =nom;
        this.sexe=sexe;
        this.anneeNaissance=anneeNaissance;
        this.anneeEmbauche=anneeEmbauche;
        this.salaire=salaire ;
    }
/* Constructeur par Défaut */
public Employe(  ){
}


/* Définition des méthodes */

    public String getNom(){return this.nom;}
    public void setNom(String nom){this.nom=nom;}
    public String getSexe(){return this.sexe;}
    public void setSexe(String sexe){this.sexe=sexe;}
    public int getAnneeNaissance(){return this.anneeNaissance;}
    public void setAnneeNaissance(int
anneeNaissance){this.anneeNaissance=anneeNaissance;}
    public int getAnneeEmbauche(){return this.anneeEmbauche;}
    public void setAnneeEmbauche(int
anneeEmbauche){this.anneeEmbauche=anneeEmbauche;}
    public int getAge(){return this.age;}
    public void setAge(){
        this.age= Year.now().getValue()-this.anneeNaissance;
    }
    public int getAnciennete(){return this.anciennete;}
    public void setAnciennete(){
        this.anciennete= Year.now().getValue()-this.anneeEmbauche;
    }
    public double getSalaire(){return this.salaire;}
    public void setSalaire(double salaire){this.salaire=salaire;}
    public void augmentSalaire(double taux){
        if(taux<-1.0 || taux>1.0){
            System.out.println("Vous devez indiquer une valeur correcte du
taux\n La valeur doit être compris entre -1.0 et 1.0");
            System.exit(1);
        }
        this.salaire=this.salaire*(1+taux);}
}
```

4.1.1.1 Définition du package de classe, déclaration de la classe et import des librairies

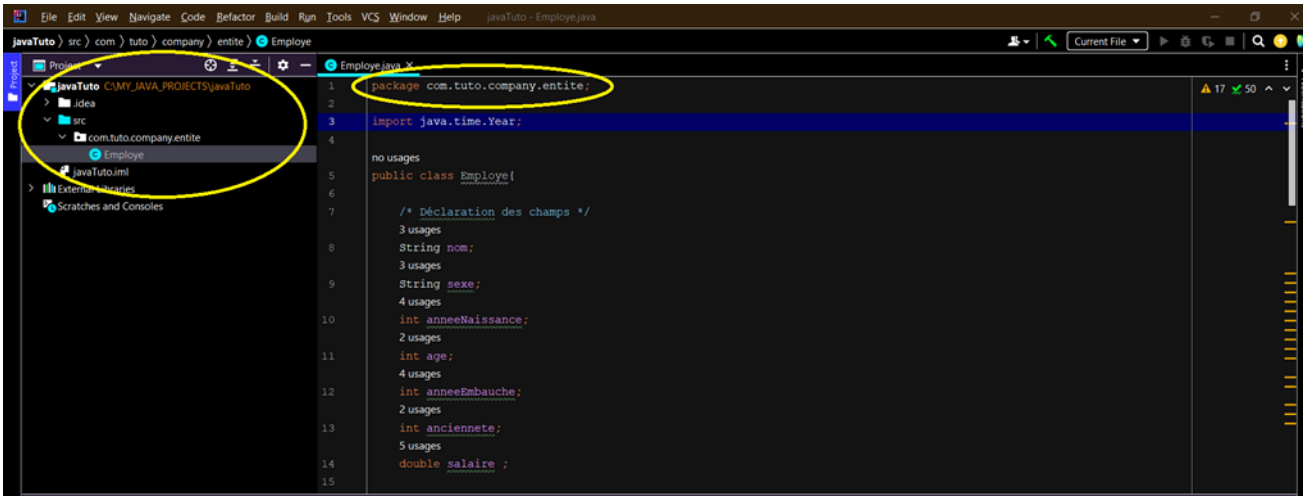
Définition du package de classe

Pour commencer l'écriture de notre classe, nous avons besoin d'abord de définir le package de classe. Le package de classe est un sous-répertoire ou une succession de sous-répertoires imbriqués au bout duquel sera stocké le fichier qui va contenir le code source de notre classe. Par exemple pour écrire la classe *Employe*, nous avons choisi le nom de package *com.tuto.compay.entite*. Cela représente une arborescence à trois sous-répertoires située

dans le dossier *src* à la racine de notre projet Java nommé *JavaTuto*. L'arborescence physique sur le disque est :

C:\MY_JAVA_PROJECTS\javaTuto\src\com\tuto\company\entite\Employe.java.

A noter que dans tout projet Java, le dossier *src* est conventionnellement le dossier racine des codes sources qu'il s'agit d'un projet d'un seul module ou d'un projet multimodules. A côté du dossier *src*, il doit également exister un dossier nommé test qui servira à accueillir les codes et les fichiers ressources servant à dérouler les tests unitaires. Nous reviendrons plus tard sur les tests unitaires Java notamment Junit tests.



L'utilisation de packages dans un projet Java vise d'une part à mieux organiser l'architecture du code. Le nombre de package retenu dans un projet n'est pas très important. L'essentiel est d'organiser les codes selon une structure qui réponde à une certaine logique choisie par le(s) développeurs. Par exemple, on peut regrouper dans le même package les classes de même nature : les classes d'entités dans un package, les classes de traitement dans un package dédié, la classe Main dans un package à part. On peut même créer un package dédié pour les classes utilitaires, qui sont en fait des classes fournissant des fonctions transverses utilisables par tous les autres types de classes. L'organisation des classes du projet Java en des packages vise, d'autre part, à mieux isoler les codes et à contrôler leur accessibilité. En effet, comme nous allons le voir plus tard, grâce à l'organisation des codes sous formes de packages, on peut contrôler le niveau d'accès à un champ ou à une méthode dans une classe. Par exemple, on peut limiter l'accès à un champ uniquement aux classes d'un même package en déclarant ce champ avec le qualificateur *protected* (nous reviendrons plus tard sur la visibilité des membres de classe).

Déclaration de la classe

Le premier acte significatif de définition d'une classe est la déclaration de son nom. Dans notre exemple, nous matérialisons l'entité des employés par la classe nommée *Employe*.

Cependant la déclaration du nom de la classe symbolisant son acte de création peut aussi être précédée par la spécification d'un certain nombre de qualificateurs qui permettront par la suite non seulement de mieux la caractériser mais aussi de contrôler ses comportements.

Par exemple on peut utiliser les mots clés public ou private pour agir sur son accessibilité par les autres classes, le mot-clé *static* pour pouvoir l'utiliser sans instanciation ou le mot-clé final pour la rendre non héritable par d'autres classes. Nous reviendrons en détail sur chacun de ces qualificateurs plus tard dans le reste du document. Dans le présent exemple, nous avons créé la classe *Employe* avec le qualificateur *public*. Cela permettant à des classes définies dans d'autres packages de pouvoir l'appeler et l'instancier. Nous reviendrons plus bas sur la procédure pour appeler et instancier une classe afin de créer un objet de la classe.

Import des librairies

L'import des librairies est aussi un point important dans la procédure de définition. Il devient surtout nécessaire lors de l'écriture du code de la classe. L'import consiste à ramener et à rendre disponibles toutes les dépendances nécessaires pour écrire la classe. Les librairies sont notamment les classes et les utilitaires qui forment nativement le socle du langage Java et qui ne sont pas disponibles par défaut pour notre classe. Les imports concernent aussi les classes du même projet Java mais qui ont été développés dans d'autres packages. En effet, pour faire communiquer les classes d'un même projet, il faut utiliser les imports. Par exemple, lorsqu'une classe *B* d'un package *y* a besoin d'une classe *A* d'un package *x*, la classe B doit nécessairement importer la classe *B* en lançant l'instruction import *x.B*.

Dans le cas de la classe *Employe*, nous avons importé la classe *Year* depuis la librairie nommée *java.time* qui est une librairie native Java permettant la manipulation des dates en Java. Notons tout de même qu'il n'est pas obligatoire d'importer les librairies au moment de déclarer la classe. On peut les importer au fur et à mesure de l'avancement du développement et les importer à l'instant où le besoin se pose.

4.1.1.2 Définition des champs

Dans la classe *Employe*, sept champs sont déclarés : *nom, sexe, anneeNaissance, age, anneeEmbauche, anciennete* et *salaire*. Parmi ces sept champs, cinq sont prédéterminées, c'est-à-dire connues d'avance. Il s'agit de *nom, sexe, anneeNaissance, anneeEmbauche* et *salaire*. Par contre deux champs sont variables car leurs valeurs changent dynamiquement en fonction du temps. Il s'agit de *age* et *anciennete*. L'âge dépend directement de l'année de naissance tandis que l'ancienneté dépend de l'année d'embauche de l'employé dans l'entreprise. Il est important d'avoir à l'esprit ce genre de relation entre les champs afin de mieux identifier ceux qu'on peut considérer comme des informations primaires (premier niveau) et les informations de second niveau. Ici l'âge et l'ancienneté sont clairement des champs de second niveau, car on peut les calculer respectivement à partir de l'année de naissance et l'année d'embauche.

4.1.1.3 Définition du (des) constructeur(s)

Par principe, le constructeur d'une classe prend le même nom que la classe. Ici la classe étant nommée *Employe*, son constructeur prend le même nom. Aussi, la classe étant

déclarée avec le qualificateur *public*, son constructeur prend également *public* ceci afin de pouvoir l'appeler dans des classes positionnées dans d'autres packages.

Une classe Java peut avoir un ou plusieurs constructeurs. Le nombre de constructeur à définir pour une classe dépend des besoins de l'utilisateur. Mais d'une manière générale, on distingue trois type de constructeurs définis selon nos propres terminologies : le constructeur standard, le constructeur par défaut et les constructeurs partiels. Le constructeur standard est le constructeur qui permet d'initialiser les valeurs de tous les champs lors de son appel. Pour pouvoir appeler ce constructeur, il faut connaître à l'avance les valeurs que vous souhaitez assigner à chacun des champs. Le constructeur par défaut est le constructeur qui permet d'instancier la classe sans initialiser les valeurs des champs ou plus précisément en assignant des valeurs par défaut aux champs. Par exemple les champs en *double* ou en *int* prennent la valeur 0 tandis que les champs de type String ou de type objet prennent la valeur nulle. Le constructeur par défaut est appelé en ne spécifiant aucun argument en paramètres. Enfin, nous avons les constructeurs partiels (il peut y en avoir autant qu'on souhaite). Les constructeurs partiels sont des constructeurs prévus pour initialiser une partie des champs définis dans la classe. Contrairement au constructeur standard (qui initialise tous les champs) et le constructeur par défaut (qui attribue les valeurs par défaut aux champs, 0 ou nulle selon le cas), le constructeur partiel initialise seulement quelques champs préalablement spécifiés. Par exemple, on peut définir un constructeur juste pour initialiser le nom, l'âge et le sexe de l'employé. Ensuite, on utilisera des méthodes pour setter les autres champs.

L'utilisation de différents types de constructeurs vise à s'adapter à de nombreuses situations. On utilise le constructeur standard lorsque l'on dispose de toutes les informations sur les champs pour créer l'objet. On utilise le constructeur par défaut lorsque la suite du fonctionnement du programme dépend de la présence de l'objet mais qu'à ce stade l'on ne dispose pas encore des valeurs des champs. Et enfin, l'on utilise le constructeur partiel lorsque la suite du programme nécessite que l'objet soit disponible mais qu'à ce stade nous ne disposons pas encore de la totalité des valeurs des champs. Bien entendu l'utilisation du constructeur par défaut ou du constructeur partiel implique que les valeurs des autres champs seront connues plus tard dans le programme et que les valeurs des champs seront mises à jour grâce à l'utilisation des méthodes dédiées appelées *accessors* (voir plus bas les méthodes). Dans l'exemple de la classe *Employe*, nous avons utilisé un constructeur partiel à la place du constructeur standard, cela pour des raisons qui sont détaillés ci-dessous. Nous avons aussi utilisé le constructeur par défaut à titre illustratif.

Constructeur partiel

Par principe, le constructeur est défini en spécifiant les valeurs de tous les champs afin de pouvoir les initialiser. Mais compte tenu de la remarque que nous avons précédemment faite sur la nature des champs, le constructeur partiel de la classe *Employe* a été défini avec cinq champs primaires obligatoires : *nom, sexe, anneeNaissance, anneeEmbauche* et *salaire*. Les deux autres champs restants (*age* et *anciennete*) seront calculés par des méthodes spécifiques dans le code.

Constructeur par défaut

Par ailleurs, il est toujours de bonne pratique de prévoir dans la classe un constructeur qui ne prend pas de d'arguments. Ce constructeur est appelé *constructeur par défaut*. Le constructeur par défaut vise à donner la possibilité d'instancier une classe, c'est-à-dire créer un objet sans renseigner les champs (mettre tous les champs à valeur nulle). L'avantage du constructeur par défaut c'est qu'il permet de créer des objets dans une partie du programme là où c'est nécessaire et d'alimenter plus tard les valeurs des champs dans le reste du traitement. Dans l'exemple ci-dessus, nous avons aussi positionné un constructeur par défaut juste à la suite du constructeur standard.

A noter qu'en plus du constructeur standard et du constructeur par défaut, on peut définir autant de constructeurs qu'on veut dans une classe. L'action de définir plusieurs constructeurs est appelée « surcharge » du constructeur. En Java la surcharge consiste à définir plusieurs fois la même fonction chacun ayant une structure d'arguments différents. L'appel de chaque fonction dépend donc du contexte tel que nous venons de l'évoquer dans le cas du constructeur partiel. Nous reviendrons plus tard sur la notion de surcharge qu'il ne faut pas aussi confondre avec la notion de redéfinition. La redéfinition consiste à réécrire dans une classe héritante une fonction qui était déjà définie dans une classe héritée. A la différence de la surcharge, la redéfinition garde la même signature de la fonction d'origine (c'est-à-dire les arguments et leur type). C'est seulement le contenu de la fonction qui change.

4.1.1.4 Définition des méthodes

Les méthodes de base d'une classe sont les *getters* et les *setters*. Un *getter* renvoie la valeur d'un champ alors qu'un *setter* modifie la valeur existante ou l'initialise si la valeur n'a pas été initialisée lors de l'appel du constructeur. La définition d'un getter est toujours précédée du type de la valeur renvoyée par l'instruction *return* : int, String, Objet, etc. La déclaration d'un setter, est quant à elle, toujours précédée du mot-clé *void,* qui signifie que cette méthode ne renvoie aucune valeur (absence de l'instruction *return*).

Dans la classe *Employe*, nous avons défini un *getter* et un *setter* pour chaque champ. Par exemple, pour le champ *nom* les méthodes définies sont *getNom()* et *setNom()*.

Pour le cas des deux champs non initialisés par le constructeur, nous avons défini deux *setters* spécifiques à savoir *setAge()* et *setAnciennete()* qui permettent respectivement de calculer l'âge et l'ancienneté et d'initialiser les champs correspondant (voir Code source CS01).

En plus des *setters* classiques, on peut aussi définir des setters spécifiques qui permettent de faire des calculs plus complexes dans le but de modifier la valeur d'un champ. C'est le cas par exemple de la méthode *augmenteSalaire()* qui permet d'appliquer un taux d'augmentation au salaire de l'employé et de mettre à jour la valeur du champs salaire. De ce point de vue, la méthode *augmenteSalaire()* est un setter complémentaire au setter standard qui est la méthode *setSalaire()*.

A noter que toutes les méthodes d'une classe ne sont pas nécessairement des *getters* et des *setters*. Il peut également exister des méthodes plus avancées dont le but n'est pas de modifier, ni de renvoyer la valeur d'un champ, mais plutôt de réaliser des actions plus complexes. Par exemples, réaliser toute une séquence de traitements combinant différents champs de la classe et de passer ces résultats à d'autres classes du programme ou de les pousser vers des systèmes externes : retour écran, base de données, page web, etc… Ainsi, on peut même dire que la puissance d'un programme écrit en langage Java réside d'abord dans les possibilités de traitement d'actions offertes par les méthodes qu'on implémente dans les classes.

Enfin, faisons remarquer que toutes les méthodes définies dans la classe *Employe* ont été déclarées avec le qualificateur public. Cela permet, comme nous allons le voir plus tard, d'appeler ces méthodes dans d'autres classes qui sont situées hors du package *com.tuto.company.entite*.

4.1.1.5 Référencement des champs définis dans la classe: le mot-clé this

Les constructeurs et les méthodes utilisent le mot-clé ***this*** pour accéder aux champs définis dans la classe. Par exemple, dans la classe *Employe* définie dans le code source **CS01**, le constructeur utilise cette expression : `this.nom =nom;`

De même, le setter comme *setNom()* utilise l'expression `this.nom=nom;` tandis que le getter getNom() utilise l'expression `this.nom;`. En effet, chaque fois qu'un constructeur ou une méthode quelconque référence un champ de classe, il utilise le mot-clé *this*. Ce mot-clé est utilisé pour confirmer que la variable qu'elle référence est bien un champ de la classe actuelle. C'est une manière d'éviter des confusions, car il arrive que des variables locales ou globales ou les champs d'autres classes portent le même nom que le champ de la classe actuelle. Le mot-clé this offre ainsi une sécurité dans le référencement des variables. Il est donc de bonne pratique de toujours utiliser le mot-clé *this* pour mieux référencer les champs de la classe actuelle.

4.1.2 Concevoir une classe de traitement

Comme nous l'avons déjà indiqué en début de cette section, nous entendons par classe de traitement une classe prévue non pas pour représenter une entité système mais plutôt prévue pour réaliser des traitements impliquant un ensemble d'entité. Dans cette sous-section nous allons présenter un cas concret de classe de traitement.

La syntaxe générale de déclaration d'une classe de traitement reste le même que la syntaxe de déclaration d'une classe d'entité (cf Syntaxe : S01). La seule particularité des classes de traitement est que les champs qui y figurent sont généralement des champs de nature techno-fonctionnels très différents des attributs primaires des entités. De plus les méthodes définies dans les classes de traitement sont pour la plupart des méthodes complexes destinées à des calculs plus poussées. En règle générale les méthodes représentent les cœurs

des programmes Java puisque c'est à leur niveau que se déroule tout le processus de traitement.

Pour illustrer l'utilisation des classes de traitement, nous allons prendre l'exemple d'une classe dont le but est de mettre à jour les informations relatives à un employé défini suivant la classe *Employe* que nous avons précédemment conçue (voir Code source CS01 plus haut).

Cette classe de traitement peut être définie comme suit :

Code source : CS02

```
package com.tuto.company.process;

// Import de la classe Employe depuis le package com.tuto.company.entite
import com.tuto.company.entite.Employe;
import static java.lang.Math.round; // Import d'une librairie utilitaire Java

public  class EntiteProcessor {

    public EntiteProcessor (){};

    public void processEmploye () {
        // Appelle du constructeur :on crée un objet de la Classe Employée
        Employe employe= new Employe( "Karim Batnini", "Masculin", 1995, 2020,
2500);
        // Calcul de l'âge et l'ancienneté de l'employé
        employe.setAge();
        employe.setAnciennete();
        // Changer le salaire de l'employé par une autre valeur
        employe.setSalaire(3000);
        // Augmenter le salaire modifié de 10% (taux=0.1)
        employe.augmentSalaire(0.10);
        // Afficher le rapport sur l'employé.
        System.out.println("Nom employé: "+employe.getNom()+", Sexe:
"+employe.getSexe()+", Age: "+employe.getAge()+
                ", Ancienneté: "+employe.getAnciennete()+ ", Salaire: "+
round(employe.getSalaire()));

    }
    }
```

Dans cet exemple, nous avons déclaré une classe nommée *EntiteProcessor*. Cette classe est définie dans le package *com.tuto.company.process*. Comme indiqué plus haut le rôle de cette classe est de traiter les données d'un employé et les mettre à jour et afficher un rapport. Pour réaliser cet objectif, la classe aura besoin juste des méthodes. Elle n'aura pas besoin de champs supplémentaires. C'est pourquoi, nous avons laissé la classe sans déclarer de champs. Et puisqu'il n'y a aucun champ à alimenter dans cette classe, le constructeur ne prend donc pas de paramètres. On se focalise alors plus sur les méthodes. En effet, nous avons défini une méthode nommée processEmploye() qui sert à réaliser les traitements souhaités sur les objets de classe *Employe*.

Instancier la classe Employe (créer un objet de la classe)

Pour pouvoir appliquer des traitements sur un employé, il faut d'abord instancier la classe *Employe*. Instancier une classe signifie créer un objet de cette classe, c'est-à-dire matérialiser la classe avec des données concrètes.

Ici pour instancier la classe *Employe*, nous devons d'abord importer la classe depuis le package *com.tuto.company.entite*. En effet, cette étape d'import est nécessaire car la classe *Processor* et la classe *Employe* ne se trouvent pas dans le même package. L'import de la classe *Employe* est fait en début de définition de la classe *EntiteProcessor* à travers la ligne de code *import com.tuto.company.entite.Employe*.

L'import étant effectué, nous pouvons maintenant instancier la classe *Employe*. L'instanciation se fait avec l'opérateur new qui est une instruction permettant d'appeler le constructeur de la classe et d'indiquer, si nécessaire, les valeurs des paramètres. Dans notre cas, ici l'instruction complète qui permet d'instancier la classe *Employe* est la suivante.

```
Employe employe= new Employe( "Karim Batnini", "Masculin", 1995, 2020, 2500);
```

Ainsi, on crée un objet nommé *employe* qui est une instance de la classe *Employe* avec des données concrètes. La création de l'objet employe via l'appel du constructeur de la classe *Employe* permet d'initialiser dans l'ordre les attributs *nom="Karim Batnini", sexe="Masculin", anneeNaissance=1995, anneeEmbauche=2020 et salaire=2500*.

On dispose désormais des données concrètes d'un employé qu'on peut afficher ou modifier juste en utilisant les *setters* et les *getters* initialement prévus dans la classe *Employe*.

Le premier traitement qu'on réalise est le calcul de l'âge et de l'ancienneté. Pour cela, nous appelons les méthodes *setAge()* et *setAnciennete()* prévues pour calculer l'âge et l'ancienneté de l'employé à partir respectivement des attributs *anneeNaissance* et *anneeEmbauche*.

Après le calcul de l'âge et de l'ancienneté, nous procédons à une première modification du salaire de l'employé. Nous remplaçons la valeur initiale 2500 par une nouvelle valeur 3000. Cette opération est réalisée grâce à l'appel de la méthode *setSalaire()* qui est le setter standard prévu pour le champs salaire. Toutefois, nous avons prévu un second setter pour le salaire qui permet non pas de remplacer directement le montant initial par un autre montant mais de faire varier le montant initiale dans un certain pourcentage. Nous avons nommé la méthode *augmenteSalaire()*. Elle prend en paramètre une valeur comprise entre -1.0 et 1.0. Toute valeur non comprise dans cet intervalle sera rejetée par le programme. Une valeur inférieure à 0 correspond à une baisse du salaire tandis qu'une valeur supérieure à 0 correspond à une augmentation. Dans l'exemple ci-dessus, nous avons appliqué un taux d'augmentation de 10% (taux=0.1).

A la suite des différents traitements sur l'objet *employe* crée, nous voulons maintenant produire un rapport complet sur l'employé. Pour cela, nous allons utiliser les getters pour afficher les informations. Mais étant donné qu'il s'agit d'un simple cas illustratif nous avons juste fait println à l'écran des infos de l'employé en appelant la méthode native Java *System.out.println()*.

A ce stade, nous avons tout ce qu'il faut pour dérouler tous les traitements nécessaires à notre programme : classe de traitement spécifiée, classe d'entité instanciée et objet créé. A présent, ce qu'il nous manque, c'est une classe qui nous permet d'exécuter notre traitement. Ce rôle étant dédié à la classe Main, la sous-section suivante permet de détailler la spécification et l'utilisation d'une classe *Main*.

4.1.3 Définir la classe Main et la méthode main()

La classe Main, comme son nom l'indique est la classe principale de votre module Java. Elle permet de lancer l'exécution du programme. La classe Main a pour rôle de fournir une méthode spéciale nommée *main()* qui est le véritable déclencheur du programme Java. En effet, lorsque le programme est soumis pour exécution, la JVM cherche cette méthode et l'utilise comme point d'entrée pour exécuter vos traitements. Il est donc important que cette méthode existe et qu'elle soit visible par la JVM. En réalité, parler de classe Main est un abus de langage. On devrait uniquement parler de classe contenant la méthode *main()*. En effet, il n'est pas obligatoire de créer une classe dédiée pour héberger la méthode *main()*. La méthode *main()* peut être positionnée dans n'importe qu'elle classe de traitement, à condition que cette classe ne soit pas appelée par une autre classe dans le programme. C'est pourquoi, l'architecture du code doit respecter un schéma hiérarchique Top-Down définie en fonction de la dépendance entre les classes. En théorie la méthode *main()* doit figurer dans une classe située dans le haut de la hiérarchie à partir de laquelle l'exécution du programme peut se dérouler sans conflit. C'est pourquoi, il est d'usage courant de définir la méthode main() dans une classe dédiée indépendante de toutes les autres classes et qui permet d'appeler et d'exécuter n'importe quelle classe du programme. Dans l'exemple ci-dessous, nous choisissons une classe Main appelée *Main* et spécifiée comme suit :

Code source : CS03

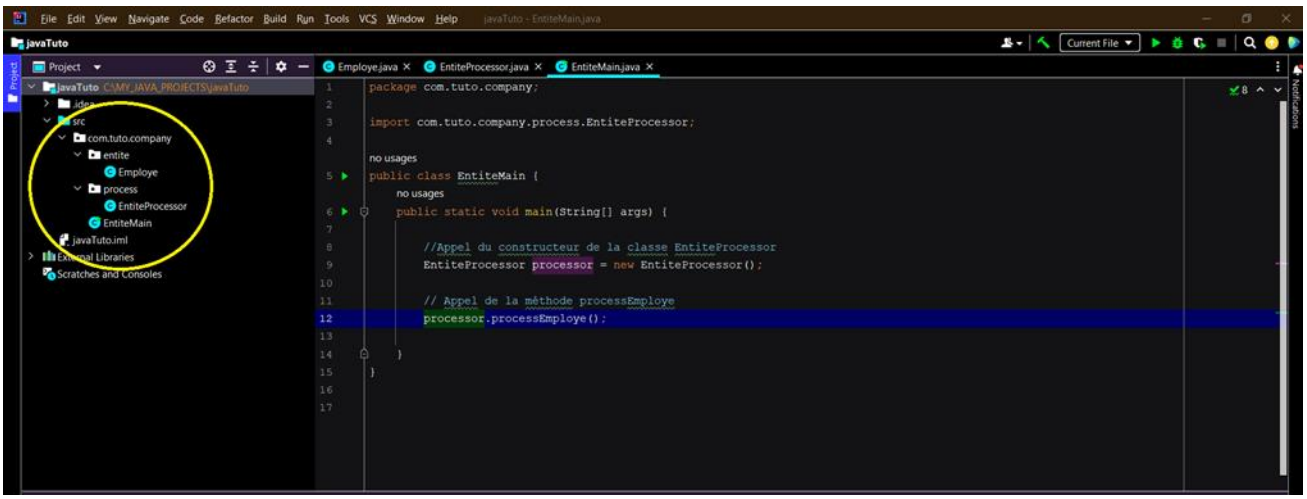
```
package com.tuto.company;

import com.tuto.company.process.EntiteProcessor;

public class Main {

    public static void main(String[] args) {

        //Appel du constructeur de la classe EntiteProcessor
        EntiteProcessor processor = new EntiteProcessor();

        // Appel de la méthode processEmploye
        processor.processEmploye();

    }
}
```

Nous avons positionné la classe *Main* dans le package *com.tuto.company* indépendamment des classes *Employe* et *EntiteProcessor*. Ensuite, nous déclarons la méthode *main()*. A noter que la méthode *main()* est une méthode générique qui se définit toujours selon la même syntaxe :

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

}
```

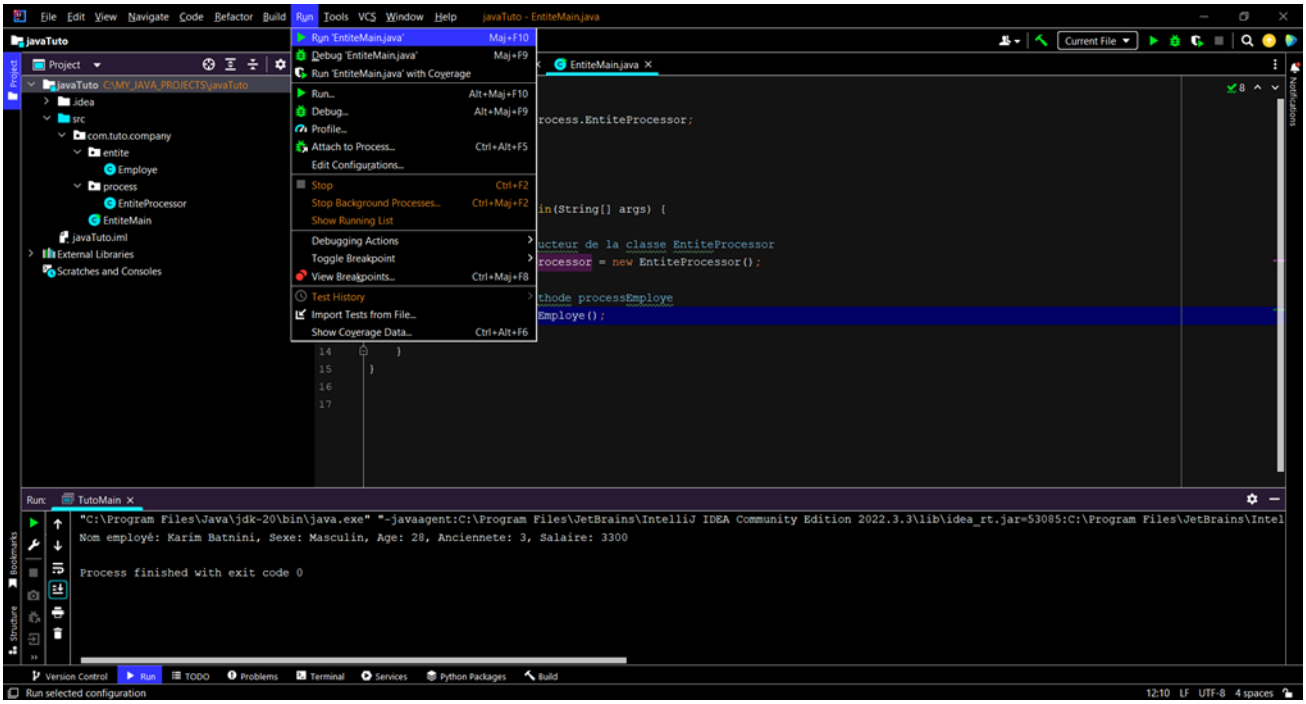
C'est à l'intérieur de cette spécification qu'on instancie les classes de traitement afin de pouvoir les exécuter. La classe de traitement dans notre exemple est la classe *EntiteProcessor*. Nous l'avons d'abord importé depuis le package *com.tuto.company.process* afin de la rendre disponible dans la méthode *main()*. Ensuite, nous instancions la classe en appelant son constructeur. A noter que ce constructeur est sans arguments car on ne vise à définir aucun champ dans la classe *EntiteProcessor*. Nous cherchons simplement à exécuter la méthode *processEmploye*() qui spécifie un certain nombre de traitements sur les données de l'employé considéré. A noter qu'il n'est pas nécessaire d'importer ici la classe *Employe* car cette classe est déjà embarquée dans la classe *EntiteProcessor* (voir Code source : CS02).

Après la définition de la classe Main, l'architecture du notre code se présente comme ci-dessous.



Pour exécuter le traitement, il suffit simplement de compiler ces trois classes en bytecode et de soumettre la classe Main à la JVM pour exécuter. Nous reviendrons plus tard sur les procédures de compilation de projet Java. Pour l'instant, nous nous limitons aux fonctionnalités offertes par l'IDE pour exécuter le code. Ici, nous utilisons Intellij IDEA (voir la section 2.2 pour plus détails sur le choix d'IDE, sa configuration et son mode d'utilisation).

Pour exécuter ce code sur Intellij, cliquer dans le menu Run> Run Main.java.



A la fin d'exécution, le rapport renvoyé est :

```
Nom employé: Karim Batnini, Sexe: Masculin, Age: 28, Ancienne: 3, Salaire: 3300
```

4.1.4 Interdire l'héritage et l'extension d'une classe : le mot-clé final

Nous avons déjà montré qu'on pouvait interdire la modification d'une variable lorsque sa valeur est assignée pour la première fois. Pour cela, il suffit de déclarer la variable avec le qualificateur final. Dans le cas des classes, l'utilisation du mot-clé final a un sens plus large. En effet, elle signifie que la classe ne peut plus être étendue c'est-à-dire qu'aucune autre classe ne peut hériter de cette classe en ajoutant des champs et des méthodes supplémentaires. L'usage du qualificateur final vise souvent à sécuriser une classe et interdire toute modification pour éviter l'accès et le retraitement des données sensibles portées souvent dans les classes. L'exemple ci-dessous montre la déclaration et la définition d'une classe de type final.

```
import java.security.NoSuchAlgorithmException;
import java.math.*;

public final class MykeyGenerator {
  private String clearKey;
  public MykeyGenerator (String clearKey){
      this.clearKey=clearKey;
  }
public String getSecretKey () throws NoSuchAlgorithmException {
    MessageDigest m= MessageDigest.getInstance("MD5");
    m.update(this.clearKey.getBytes(),0,this.clearKey.length());
    return new BigInteger(1,m.digest()).toString(16);
}
```

```
}
```

Dans cet exemple, nous définissons une classe nommée MykeyGenerator dans le but de générer un haskey à partir d'un code ou d'un mot de passe fournit en clair : clearKey. Nous utilisons l'algorithme MD5 pour encrypter cette clé. Nous avons déclaré la classe MykeyGenerator avec le qualificateur final pour éviter qu'un utilisateur mal intentionné puisse étendre cette classe en ajoutant une autre fonction hashage autre que MD5. Dès lors, on ne pourra plus faire la correspondance entre les codes en clair et le code encrypté. Cela représente l'une des situations où il est possible d'utiliser le mot clé final pour une classe.

L'exemple ci-dessous montre un exemple d'appel de la classe spécifiée.

```
import com.tuto.company.other.*;

import java.security.NoSuchAlgorithmException;

public class Main {
    public static void main(String[] args) throws NoSuchAlgorithmException {

        MykeyGenerator keygen= new MykeyGenerator("00235467KMK"); // Instance
de la classe final
         String secretKey=keygen.getSecretKey();
         System.out.println("Votre code crypté est: "+secretKey);
    }
}
```

En exécutant ce code, on obtient :

```
Votre code crypté est : c0c5afde646c8bb67ba1c79946e46fcf
```

4.2 Créer un objet (instancier une classe)

Un objet est une matérialisation de la classe avec des données concrètes. Une classe est une structure générique qui se comporte comme une moule à briques avec lequel on peut produire des briques de même format mais pas nécessairement de même constitution. La même moule peut être utilisée pour produire une brique de ciment ou une brique d'argile. Même si les deux briques ont la même spécification (forme proposée par la moule), elles n'ont pas les mêmes informations (même constituant). Chaque brique est donc un objet de la classe moule.

En Java, pour obtenir un objet à partir d'une classe, on dit qu'on instancie la classe ou qu'on crée une instance de la classe. L'objet est donc une instance de la classe.

4.2.1 Instancier une classe

Un objet est créé en spécifiant le mot-clé ***new*** suivi du constructeur et éventuellement renseigner les arguments. On peut instancier une classe soit avec le constructeur standard

en renseignant les arguments, soit en appelant le constructeur par défaut, celui n'ayant aucun argument. L'exemple ci-dessous montre les deux façons d'instancier la classe *Employe* que nous avons définie dans le code source CS01.

```
// Instancier avec le constructeur standard
Employe employe= new Employe( "Louise", "Féminin", 1998, 2021, 1500);
// Instancier avec le constructeur par défaut
Employe employe= new Employe();
```

Le constructeur standard instancie la classe *Employe* en renseignant les informations qui constituent les valeurs de champs. Le constructeur par défaut, lui, instancie la classe en fixant les valeurs de tous les champs à nulle. C'est par la suite qu'on pourra appeler les méthodes setters pour définir ces valeurs. A noter que lorsqu'un objet est créé par le constructeur par défaut (constructeur sans argument), les valeurs des champs sont définies par défaut. Les champs de type int et double sont initialisés à 0 et les champs de type String et les autres champs objets de type classe sont initialisés à null.

Dans cet exemple, l'identificateur *employe* est un objet de la classe *Employe* car elle encapsule désormais des valeurs concrètes. Un objet, une fois défini, se comporte comme une variable. Son type est non pas un type primitif mais plutôt un type référence représenté par la classe dont il constitue une instance. Dans l'exemple ci-dessus, le type de la variable *employe* est bien la classe *Employe*.

4.2.2 Règle de nommage d'un objet

Par ailleurs, un objet étant un identificateur tout comme les variables, il obéît aux mêmes règles de nommage que ces dernières. Leur nom commence toujours par une lettre minuscule ; leur nom ne doit pas comporter de caractères spéciaux ; lorsque le nom est composé de plusieurs mots, (à l'exception du premier mot), la première lettre de chaque autre mot est écrite en majuscule ; enfin le nom d'un objet ne doit pas être un mot réservé du langage Java (voir la section 3.3.2 pour plus de détails sur les règles et convention de nommage des variables).

4.2.3 Interdire la modification de la référence d'un objet : le mot-clé final

Lorsqu'un objet a été déclaré avec le mot-clé final, dès que cet objet a été initialisé en instanciant une classe une première fois, il n'est plus possible créer un autre objet portant le même nom en instanciant une nouvelle fois la même classe ou tout autre classe. Le mot-clé final verrouille la référence à un objet et la rend non modifiable dans le reste du programme. L'exemple ci-dessous montre l'utilisation du mot-clé final pour rendre non modifiable la référence à un objet.

```
// Première instanciation avec le constructeur par défaut
final Employe employe= new Employe();
```

```
// Instanciation avec la même référence
employe= new Employe( "Louise", "Féminin", 1998, 2021, 1500);// définition
rejetée
```

Cette modification de la référence *employe* sera rejetée car l'objet *employe* a été déclarée pour la première fois avec le mot-clé final.

Attention à ne pas confondre la référence à un objet et le contenu de cet objet. En effet, même si le qualificateur final rejette une nouvelle instanciation de la classe *Employe* sous le même nom *employe*, il est tout à fait possible de modifier le contenu de l'objet *employe*. Par exemple, on peut appeler les setters de l'objet pour redéfinir les champs comme le nom, le sexe, etc.. En effet, le qualificateur final verrouille la référence à l'objet (via son nom) mais pas son contenu. Pour verrouiller le contenu d'un objet, par exemple la valeur d'un champ, ce champ doit être d'abord déclaré comme final dans la classe source.

4.3 Encapsulation et visibilité des membres de classe

4.3.1 Rappel du principe d'encapsulation

Comme nous l'avons déjà évoquée dans le chapitre introductif, l'encapsulation est le principe de base qui gouverne la Programmation Orientée-Objet. L'instanciation d'une classe sous forme d'objet permet d'envelopper les données dans une structure dont on peut contrôler le comportement. Pour rappel, dans une classe, les champs servent à stocker les informations (les attributs) tandis que les méthodes servent à accéder à ces données et à les modifier éventuellement. En principe, pour agir sur les données stockées dans un objet, l'utilisateur doit passer par les méthodes prévues à cet effet. Par exemple, pour récupérer la valeur d'un champ, il appelle un *getter* et pour modifier la valeur d'un champ il appelle un *setter*. Sans l'utilisation de ces méthodes, les champs restent totalement invisibles et inaccessibles aux autres classes et objets du programme. Ainsi, l'encapsulation est un principe qui aboutit à restreindre le mode d'accès aux données. Toutefois, Java offre aussi la possibilité, à travers des qualificateurs dédiés, d'élargir le mode d'accès aux champs et de mieux contrôler le scope de visibilité aussi bien d'un champ, d'une méthode ou même d'une classe. La section ci-dessous passe en revue ces qualificateurs.

4.3.2 Visibilité des champs et des méthodes : public, private, protected

Les mots-clés *public*, *private* et *protected* sont les trois qualificateurs prévus pour contrôler le niveau de visibilité des membres : champs et méthodes. Le qualificateur est spécifié de la déclaration du champ ou de la méthode. Il est placé avant le type et le nom du membre déclaré. Ci-dessous quelques exemples de déclaration de champs et de méthodes.

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
// Cas de déclaration de champs
public enum TailleChemise { S, M, L, XL, XXL };
private String couleur= "BLEU";
protected double poids=85.0;

// Cas de déclaration de méthodes
public String dateCourante(){
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    Date date = new Date();
    return dateFormat.format(date);
}
private double getSalaire(){
    return this.salaire;
}
```

Cet exemple montre plusieurs cas d'utilisation des qualificateurs *public, private* et *protected* lors de la définition de champs pou de méthode. Le choix d'un type de qualificateur dépend du niveau de visibilité que vous souhaitez fixer pour le membre. Ci-dessous les détails sur le rôle de chaque qualificateur.

4.3.2.1 public

Lorsqu'un membre de classe est déclaré public, tous les autres objets du programme peuvent y accéder sans passer par les méthodes s'il s'agit d'un champ. Et lorsqu'il s'agit d'une méthode, cette méthode peut être appelée dans n'importe partie du programme. Par exemples, en reprenant la classe *Employe* définie dans le code source CS01, si le champ *nom* avait été déclaré public[4], alors il aurait été possible d'accéder et de modifier la valeur de *nom* partout dans le programme et cela sans passer par les méthodes *getNom()* et *setNom()*. Par exemple dans la classe *EntiteProcessor* (voir code source CS02), après avoir créé l'objet *employe*, il aurait été possible de lancer les actions suivantes :

```
String n=employe.nom; // A la place de: String n=employe.getNom()
employe.nom="Victor"; // A la place de: employe.setNom("Victor")
```

Cet exemple montre donc qu'il faut utiliser le qualificateur public avec beaucoup de précaution lors de la définition d'un champ.

De même, lorsque vous ne souhaitez pas qu'une méthode puisse être appelée n'importe où dans le programme, vous ne devez pas utiliser le qualificateur public. Encore une fois, le choix d'un type de qualificateur dépend du degré de visibilité que vous souhaitez accorder aux autres objets du programme sur le membre de classe. Dans le cas des méthodes, on souhaite avoir la possibilité d'appeler partout dans le programme alors que pour d'autres non. Par exemples, on peut vouloir appeler les méthodes *getNom()* et *setNom()* partout dans le programme. Alors qu'à l'inverse l'on ne voudrait pas autant pouvoir accéder aux méthodes *getSalaire()* ou setSalaire() simplement parce qu'on estime que les informations sur le salaire sont plus sensibles.

[4] Dans la classe Employe, les champs ont été déclarés sans qualificateurs. Dans ce cas le qualificateur par défaut est protected.

4.3.2.2 private

Lorsqu'un membre d'une classe est déclaré avec le qualificateur *private*, ce membre n'est visible que par les autres membres de la classe. Les membres des autres classes du programme ne peuvent pas directement accéder, ni modifier ces membres. Lorsque le membre déclaré en private est un champ, pour accéder à ces champs dans une autre partie du programme, il faut passer par des méthodes prévues à cet effet dans la classe parente du membre appelées *accessors* (qui sont en fait les *getters* et les *setters*). Mais ces *accessors* doivent être néanmoins déclarés *public* pour pouvoir les appeler en dehors de la classe parente. Par exemple, dans la classe *Employe* définie dans le code source CS01, tous les champs étant en private[5], pour pouvoir les utiliser dans le reste du programme, nous avons prévu des *accessors* qui sont eux tous déclarés public. Ex : *getNom(), setNom(), getAge(), setAge(), getSexe(), setSexe()*. Voir code source CS01.

Dans une classe, lorsqu'une méthode est déclarée *private*, la visibilité de cette méthode est restreinte aux autres membres de la classe. La méthode n'est ni directement accessible, ni appelable à l'extérieur de lasse. Revenons au code source CS01. Toutes les méthodes définies dans cette classe sont de type public car nous avons souhaité que ces champs soient accessibles hors de la classe *Employe*. Par exemple, ces méthodes ont été appelées dans la classe *EntiteProcessor* pour réaliser quelques retraitements sur l'objet *employe* et afficher un rapport sur l'employé retraité (voir code source CS02). Mais dans la classe *Employe*, il aurait été bien possible de définir des classes de type *private* et qu'on n'aura pas besoin d'appeler à l'extérieur et qui sont appelée uniquement par d'autres méthodes de la classe. Pour illustrer ce cas, prenons un exemple concret.

Dans la classe *Employe*, l'âge et l'ancienneté de l'employé ont été calculés avec les méthodes suivantes en calculant d'abord l'année courante avec la fonction `Year.now().getValue()` :

```
public void setAge(){
    this.age= Year.now().getValue()-this.anneeNaissance;
}
public void setAnciennete(){
    this.anciennete= Year.now().getValue()-this.anneeEmbauche;
}
```

Au lieu d'appeler deux fois la même fonction pour calculer l'année courante, on pouvait définir une méthode servant à récupérer l'année nommée *getAnneeCourante()* et appeler cette méthode dans les méthodes *setAge()* et *setAnciennete()*. Ainsi deviendrait comme ci-dessous :

```
private int getAnneeCourante(){
        return Year.now().getValue();
}
public void setAge(){
    this.age= getAnneeCourante()-this.anneeNaissance;
}
public void setAnciennete(){
    this.anciennete= getAnneeCourante()-this.anneeEmbauche;
}
```

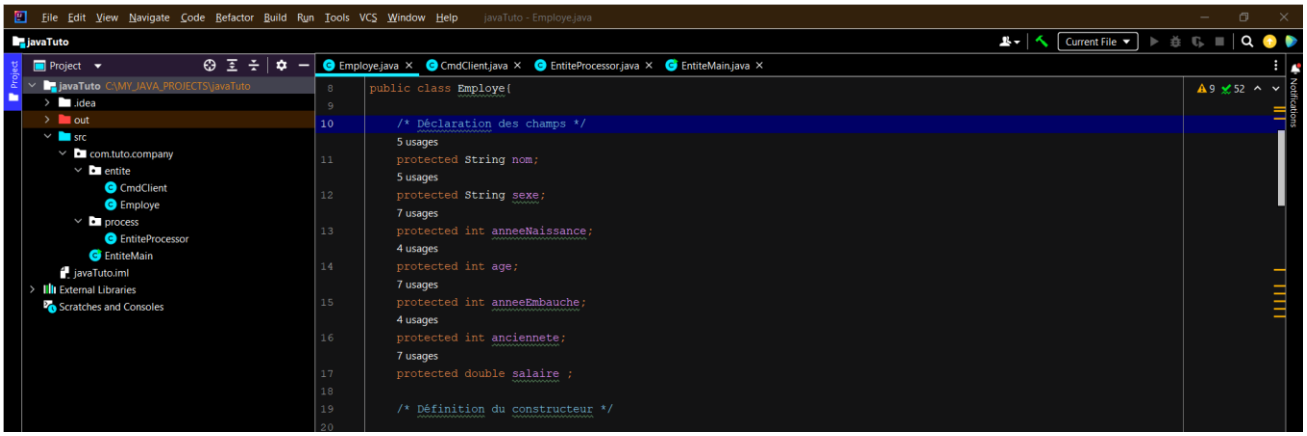
[5] Par défaut les champs et les méthodes déclarés sans qualificateur sont implicitement mis en visibilité protected. Les champs d'une interface sont implicitement public static final et les méthodes d'une interface sont par défaut publiques.

Dans cet exemple, puisque la méthode *getAnneeCourante()* est utilisée uniquement par les autres méthodes de la classe, on la déclare avec le qualificateur *private*. Cette méthode ne sera donc pas visible hors de la classe. Ceci-dit, rien ne nous empêche aussi de déclarer la méthode en type public.

4.3.2.3 protected

Le qualificateur *protected* offre à un membre de classe un niveau de visibilité intermédiaire entre *public* et *private*. Il étend la visibilité au niveau du package. En effet, contrairement au mot-clé public qui ouvre la visibilité du membre à tous les objets provenant de tous les packages, et contrairement au mot-clé private qui limite la visibilité aux membres de la même classe, le mot-clé *protected* fixe la visibilité au niveau du package. Ainsi, lorsqu'un membre est déclaré *protected* dans une classe, ce membre sera visible pour tous les objets des classes partageant le même package que la classe parente. Ainsi, il est possible d'accéder et éventuellement de le modifier.

Pour illustrer comment le qualificateur *protected* affecte le niveau de visibilité d'un champ, nous avons défini une nouvelle classe nommée *CmdClient* qui fournit les informations sur une commande passée par un client. Nous positionnons cette classe dans le package *com.tuto.company.entite*. Il s'agit du même package que la classe *Employe* que nous avons déjà définie (voir code source CS01). En revanche, nous avons légèrement retouché la classe *Employe* en déclarant tous les champs avec le qualificateur *protected*. Ensuite, pour des raisons de convenance, nous avons déclaré la méthode getAnneeCourante() en mode public. Les modifications sur les déclarations des champs dans la classe *Employe* sont visibles sur la capture d'écran ci-dessous [6].



Le code source CS04 ci-dessous représente la définition de la nouvelle classe *CmdClient*.

Code source : CS04

[6] Nous ne présentons pas le code source modifié de la classe *Employe*. Si vous souhaitez obtenir le code modifié, il vous suffira de mettre à jour la classe directement dans votre IDE.

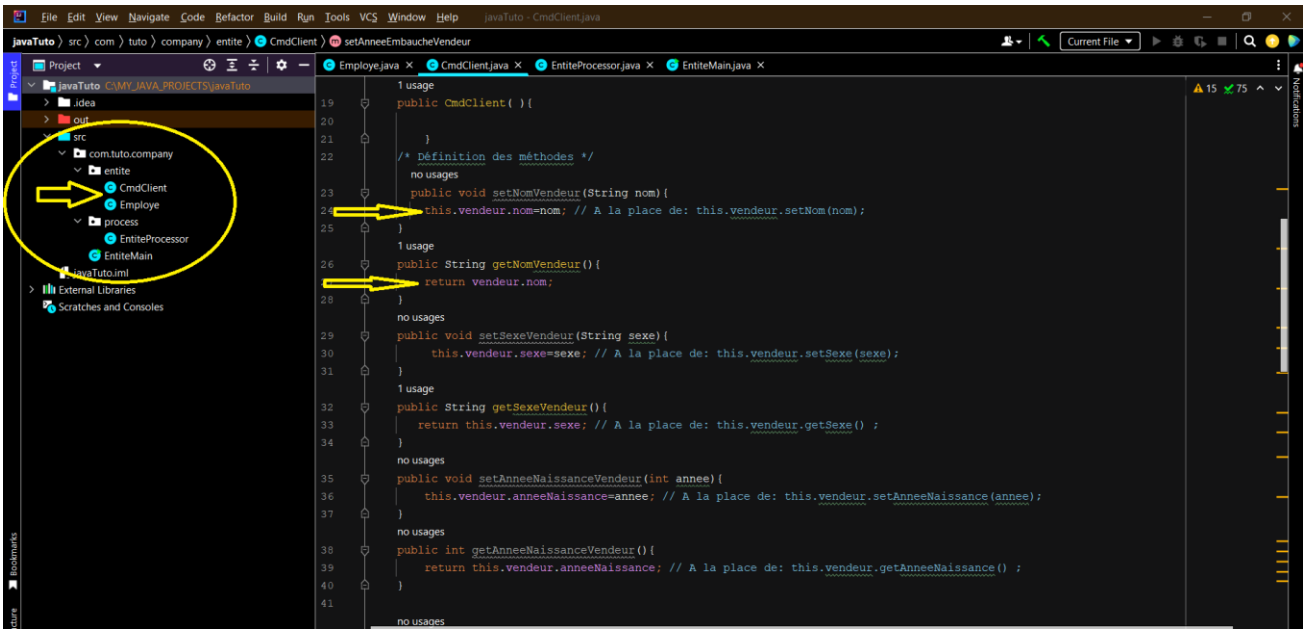
```
package com.tuto.company.entite;

public class CmdClient {


    /* Déclaration des champs */
    String idCmd;
    String idClient;
    double montantCmd ;
    Employe vendeur;

    /* Définition du constructeur */
    public CmdClient(String idCmd, double montantCmd, String idClient ){
        this.idCmd =idCmd;
        this.montantCmd=montantCmd;
        this.idClient =idClient;
        this.vendeur=new Employe();
    }
    public CmdClient( ){

        }
    /* Définition des méthodes */
      public void setNomVendeur(String nom){
        this.vendeur.nom=nom; // A la place de: this.vendeur.setNom(nom);
    }
    public String getNomVendeur(){
        return vendeur.nom; // A la place de: this.vendeur.getNom(nom);
    }
    public void setSexeVendeur(String sexe){
         this.vendeur.sexe=sexe; // A la place de: this.vendeur.setSexe(sexe);
    }
    public String getSexeVendeur(){
       return this.vendeur.sexe; // A la place de: this.vendeur.getSexe() ;
    }
    public void setAnneeNaissanceVendeur(int annee){
        this.vendeur.anneeNaissance=annee; // A la place de:
this.vendeur.setAnneeNaissance(annee);
    }
    public int getAnneeNaissanceVendeur(){
        return this.vendeur.anneeNaissance; // A la place de:
this.vendeur.getAnneeNaissance() ;
    }

    public void setAgeVendeur(){
        this.vendeur.age= vendeur.getAnneeCourante()-
this.vendeur.anneeNaissance; // A la place de : this.vendeur.setAge();
    }
    public int getAgeVendeur(){
        return this.vendeur.age ; // A la place de : this.vendeur.getAge();
    }
    public void setAnneeEmbaucheVendeur(int annee){
        this.vendeur.anneeEmbauche=annee; // A la place de:
this.vendeur.setAnneeEmbauche(annee);
    }
    public int getAnneeEmbaucheVendeur(){
        return this.vendeur.anneeEmbauche; // A la place de:
this.vendeur.getAnneeEmbauche() ;
    }

    public void setAncienneteVendeur(){
        this.vendeur.anciennete= vendeur.getAnneeCourante()-
```

```
this.vendeur.anneeEmbauche; // A la place de : this.vendeur.setAnciennete();
    }
    public int getAncienneteVendeur(){
        return this.vendeur.anciennete ; // A la place de :
this.vendeur.getAnciennete();
    }
    public void setSalaireVendeur(double salaire){
        this.vendeur.salaire=salaire; // A la place de:
this.vendeur.setSalaire(salaire);
    }
    public double getSalaireVendeur(){
        if(this.vendeur==null){this.vendeur=new Employe();}
        return this.vendeur.salaire; // A la place de:
this.vendeur.getSalaire() ;
    }
    public String getIdClient(){return this.idClient;}
    public void setIdClient(String idClient){this.idClient=idClient;}
    public double getMontantCmd(){return this.montantCmd;}
    public void setMontantCmd(double montantCmd){this.montantCmd=montantCmd;}
    public Employe getVendeur(){return this.vendeur;}
}
```

Dans cet exemple, nous avons considéré qu'une commande client est une entité pouvant être représentée de façon minimaliste par les informations suivantes : l'identifiant de la commande (champ *idCmd*), l'identifiant du client qui a passé la commande (*idClient*), le montant de la commande (*montantCmd*) et le *vendeur*, c'est-à-dire l'employé qui a assuré le suivi de la commande. Le vendeur est donc une instance de la classe *Employe*.

Il y a donc une dépendance entre la classe *CdmClient* et la classe *Employe* car toute instanciation de la classe *CmdClient* nécessite un objet *employe*, donc une instanciation de la classe *Employe*. Mais dans la vie réelle, il se peut que lorsqu'un client passe une commande (par exemple sur un site de commerce en ligne, etc..), l'application de réception des commandes peut matérialiser la commande en renseignant automatiquement les informations de bases. Ex : *idCdm*, *idClient*, *montantCmd*. Mais il peut s'écouler un certain temps entre le moment où la commande est matérialisée dans l'application et le moment où elle est prise en charge par un employé pour être traitée. Dans une telle situation, le champ vendeur pourra rester non renseigné dans un premier temps (valeur nulle). Ensuite lors du traitement de la commande, la valeur du champ vendeur sera assignée. C'est-à-dire une instance de la classe *Employe* sera créée pour servir de valeur au champ vendeur. Mais la création d'un objet vendeur ne signifie pas également que tous les champs de cet objet sont renseignés. Cela dépend du constructeur appelé pour instancier la classe *Employe*. Par exemple si c'est le constructeur par défaut qui est appelé, tous les champs *String* auront une valeur nulle et tous les champs de type *int* et *double* seront fixés à 0. Il faudrait alors laisser la possibilité de changer ces valeurs afin de renseigner complètement les informations du vendeur (employé). Dans la classe *Employe*, lorsque les champs sont déclarés en *private*, on ne pourra modifier ces champs qu'en appelant les méthodes de la classe *Employe* elle-même. Mais lorsque les champs sont déclarés en *protected*, on peut directement modifier les valeurs sans passer par les méthodes de la classe *Employe*. La

classe qui a créé un objet *employe* peut directement modifier ses attributs si cette classe se trouve dans le package que la classe *Employe*. C'est le cas ici pour la classe *CmdClient* qui, en effet, se trouve dans le même package que la classe *Employe* qui est ici *com.tuto.company.entite* (voir sur la gauche de la capture d'écran ci-dessous).



La classe *Cmdclient* et la classe *Employe* étant situées dans le même package et les champs de la classe *Employe* étant déclarés en *protected*, alors la classe *CmdClient* peut accéder à ces champs sans passer par les *accessors* de la classe *Employe* (c'est-à-dire sans passer par les getters et les setters de la classe *Employe*). En effet, dans la classe *Cdmclient*, après avoir créé un objet de la classe *Employe*, il suffit simplement de référencer un champ quelconque de cet objet en faisant *nomObjet.nomChamp*. Il s'agit alors d'un accès direct. Par exemple pour accéder aux attributs du vendeur qui est un objet de la classe *Employe*, les méthodes définies dans la classe *CmdClient* font des accès directs (voir Code source CS04 ci-dessus). Par exemple pour accéder au nom du vendeur, les méthodes *getNomVendeur()* et *setNomVendeur()* de la classe *CmdClient* sont définies comme suit :

```
public String getNomVendeur(){
    return vendeur.nom; // A la place de: this.vendeur.getNom(nom);
}
public void setNomVendeur(String nom){
    this.vendeur.nom=nom; // A la place de: this.vendeur.setNom(nom);
}
// vendeur étant une instance de la classe Employe
```

En somme le principe d'encapsulation est réellement respecté lorsque les champs sont déclarés en mode private. Mais lorsqu'un champ est déclaré en mode public ou en mode *protected*, cela laisse la possibilité à d'autres objets du programme d'accéder aux valeurs des champs en court-circuitant les méthodes prévues à cet effet.

4.3.3 Interdire la modification d'un membre de classe : le mot-clé final

4.3.3.1 Cas des champs

Pour interdire la modification de la valeur d'un champ dès que la valeur est assignée une première fois, on utilise le mot-clé final. Cependant l'effet de ce qualificateur diffère selon qu'il s'agit d'un champ de type primitif ou d'un champ de type objet. Lorsque le mot-clé final est utilisé pour un champ de type primitif (*int, double, float, long*, etc..), la valeur et la référence du champ sont verrouillés. C'est-à-dire dès qu'on définit la valeur du champ pour la première fois, on ne plus la modifier. Ex :

```
final int age= 25;
age=26; // Modification rejetée
```

En revanche, lorsque le champ est de type objet c'est-à-dire instance d'une classe, alors le qualificateur final verrouille seulement la référence de l'objet. Mais il reste possible d'appeler les méthodes de l'objet pour modifier les champs qui le caractérisent. Ex :

```
final Employe vendeur= new Employe( "Karim Batnini", "Masculin", 1995, 2020, 2500);
vendeur=new Employe( "Claudine", "Féminin", 1997, 2021, 2000); // Modification rejetée
 // Modification du salaire du vendeur
 vendeur.setSalaire(3000); // Action autorisée
```

Dans cet exemple, le champ vendeur est déclaré avec le qualificateur final et initialisé en instanciant la classe *Employe*. Le champs vendeur est donc de type classe. Ensuite, nous essayons d'assigner une nouvelle valeur au champ. Cette modification est rejetée. Par contre, nous pouvons modifier les champs de l'objet vendeur lui-même. C'est pourquoi nous appelons la méthode *setSalaire()* avec le paramètre 3000. Cela change la valeur initiale qui était de 2500 à 3000.

4.3.3.2 Cas des méthodes

Dans une classe, lorsqu'une méthode est déclarée avec le qualificateur final, cela signifie qu'il ne sera pas possible de redéfinir cette méthode dans les classes qui hériterons de la classe actuelle. Plus concrètement, si une méthode *m()* a été déclarée final dans une classe A. Si on crée une classe B par héritage du classe A[7], il ne sera pas possible dans la classe B de redéfinir la méthode *m()* c'est-à-dire réécrire le code la méthode *m()*. Nous reviendrons plus tard sur la notion de redéfinition de méthode. Néanmoins nous présentons un exemple ci-dessous pour illustrer l'utilisation d'une méthode qualifiée *final*.

```
// Définition  de la classe Parent
package com.tuto.company.other;
import java.time.Year;
public class Pere {
    String nom;
```

[7] Nous reviendrons plus tard sur la notion d'héritage de classe.

```
    int age ;
    public final void setNom() {
        this.nom="Dupont";
    }
    public void setAge() {
        this.age=Year.now().getValue()-1980;
    }
}


// Définition  de la classe Enfant
package com.tuto.company.other;
import java.time.Year;
public class Fils extends Pere {
    public void setAge() {
        this.age=Year.now().getValue()-2000;
    }
    public int getAge(){
        return this.age;
    }
    public String getNom(){
        return this.nom;
    }
}
```

Dans cet exemple, nous avons défini deux classes : la classe *Pere* et classe *Fils* toutes les deux caractérisées par les attributs *nom* et *age*. La classe *Pere* déclare les deux attributs et définit deux méthodes *setNom()* et *setAge()*. La méthode *setNom()* fixe la nom à « *Dupont* » et la méthode est déclarée *final*. La classe *Pere* définit également la méthode *setAge()* qui calcule l'âge du père en faisant la différence entre la date courante et 1980 (année de naissance du père).

La classe *Fils* « étend » la classe *Pere* en utilisant le mot-clé *extends*. On dit alors que la classe *Fils* hérite de la classe *Pere* (nous reviendrons plus tard sur la notion d'héritage de classe). Comme on peut le remarquer, les attributs *nom* et *age* ne sont pas re-déclarés dans la classe Fils. Car il n'est pas nécessaire de re-déclarer dans la classe héritante les champs et les méthodes qui ont été déjà déclarés dans la classe parente si ces champs et méthodes n'ont pas vocation à être redéfinis. Ensuite la méthode *setNom()* n'a pas été reportée dans la classe Fils car cette méthode a été déclarée final dans la classe Père. Cela signifie qu'elle ne peut pas être modifiée par toute classe qui hériterait de la classe Pere. En revanche la méthode *setAge()* a été modifiée dans la classe Fils. En effet, la formule de calcul de l'âge du fils n'est plus la même que celle du père, l'année de naissance de l'enfant est 2000. De plus, la classe Fils a aussi ajouté deux méthodes en plus : *getNom()* et *getAge()* qui n'étaient pas disponibles dans la classe *Pere*.

En somme, lorsqu'une méthode est déclarée final dans une classe parente, cette méthode reste non modifiable dans toutes les classes qui étendrons par héritage de cette classe d'origine.

4.4 Définition des champs et méthodes statiques : le mot-clé static

Les champs et les méthodes statiques sont des membres de classe auxquels on peut accéder et modifier sans à avoir instancier la classe, c'est-à-dire sans avoir à créer un objet. Habituellement pour accéder à un champ d'une classe ou pour appeler une méthode donnée dans une classe, on instancie d'abord la classe en créant un objet en utilisant l'opérateur *new*. L'objet étant créé on peut maintenant accéder à chacun de ces membres (champs et méthodes). Mais Java offre aussi la possibilité d'accéder à un champ ou à une méthode d'une classe sans créer un objet. Cela est possible en utilisant le qualificateur *static*. Les sous-sections ci-dessous présente l'utilisation des champs et méthodes statiques.

4.4.1 Champs statiques (variables de classe)

Les champs statiques plus connus sous le nom de *variables de classes* (à la différence des *variables d'instance*) sont des champs dont les valeurs sont communes à toutes les instances d'une classe. Il est possible d'accéder à ces champs sans nécessairement passer par les objets. Pour illustrer la création et l'utilisation de champ *static*, nous définitions d'abord deux classes A et B telles que présentées ci-dessous.

```
// Définition d'une classe A
package com.tuto.company.other;
public class A {
   /* Définition d'un champ statique */
   static int myStaticVar= 20;

}

// Définition d'une classe B
package com.tuto.company.other;
public class B {

    int myValue= A.myStaticVar; /* Recup du champ static depuis la classe*/

}
```

Dans la classe *A*, nous déclarons un champ *static* nommé *myStaticVar*. Dans la classe *B*, nous définissons un champ appelé *myValue* qui prend la valeur du champ *myStaticVar*. Comme on peut le remarquer, on récupère directement la valeur de *myStaticVar* sans avoir à instancier la classe *A*. Tel est le résultat lorsqu'un champ est défini avec le qualificateur *static*.

Il faut noter que l'usage du mot-clé *static* n'interdit pas d'instancier la classe A pour accéder à la valeur du champ. En effet, la classe B pouvait bien aussi être définie comme suit :

```
package com.tuto.company.other;
public class B {
   /* Recup du champ depuis la classe*/
    A a=new A();
    int value= a.myStaticVar;
```

```
}
```

Dans ce cas, on instancie d'abord la classe *A* pour créer l'objet a. Ensuite on appelle le champ *myStatitic* en faisant *a.myStaticVar*.

4.4.2 Méthodes statiques (méthodes de classe)

Les méthodes statiques plus connues sous le nom de *méthodes de classes* sont des méthodes auxquelles on peut accéder sans avoir à instancier la classe c'est-à-dire sans passer par les objets de cette classe. Tout comme les champs statiques, on peut aussi définir des méthodes statiques en précédent leur déclaration avec le qualificateur *static*. Lorsqu'une méthode est déclarée *static* dans une classe, il n'est pas nécessaire d'instancier la classe pour pouvoir appeler la méthode. Pour illustrer ces propos, partons de deux classes A et B telles que définies ci-dessous.

```
// Définition  de la classe A
package com.tuto.company.other;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
public class A {
   /* Définition d'une méthode statique */
    static String getDateCourante(){
       DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
       Date date = new Date();
       return dateFormat.format(date);
    }

}
// Définition  de la classe B
package com.tuto.company.other;
public class B {
   /* Appel de la méthode statique depuis la classe A*/
    String dateJ= A.getDateCourante();

}
```

Dans la classe *A*, nous avons défini la méthode *getDateCourante()*. Cette méthode calcule et renvoie la date courante. La méthode étant déclarée *static* nous pouvons directement l'appeler dans la classe B sans avoir à instancier la classe *A* .

Pour rappel le fait que la méthode `getDateCourante()` ait été déclarée en *static*, cela n'empêche pas qu'on puisse l'appeler après avoir instancié la classe *A*. Ci-dessous une illustration.

```
package com.tuto.company.other;
public class B {

    /*On instancie d'abord la classe A */
    A a=new A();
   /* Appel de la méthode statique depuis l'objet a*/
        String dateJ= a.getDateCourante();
```

```
}
```

4.5 Surcharge du constructeur et des méthodes d'une classe

Dans le jargon de la programmation, la surcharge est une opération qui consiste à proposer plusieurs variantes d'une même méthode en faisant varier sa signature. La signature d'une méthode est définie par la composition de ses arguments, c'est-à-dire les paramètres et leur type. La surcharge vise à permettre à chacune des variantes de s'exécuter selon un contexte donné. En Java, le cas le plus typique pour illustrer la surcharge est l'opérateur « + ». Lorsque cet opérateur est appliqué sur deux opérandes numériques, il fait la somme arithmétique. Ex : 2+4=5. En revanche, lorsque l'opérateur est appliqué sur deux opérandes en chaîne de caractères, il fait de la concaténation. Ex : "Hello"+"World"="HelloWorld". La surcharge consiste donc à définir une fonction et ses paramètres en fonction du contexte d'exécution. Dans une classe Java, la surcharge peut concerner aussi bien le constructeur que les méthodes. En effet, comme nous l'avons déjà évoqué dans les sections précédentes, on peut définir autant de constructeurs qu'on souhaite pour une classe. Il est également possible de définir plusieurs fois la même méthode, chacune avec ses propres arguments. Dans cette section, nous allons illustrer la notion de surcharge aussi bien pour le constructeur que pour les méthodes d'une classe.

4.5.1 Surcharge du constructeur

Le bout de code ci-dessous montre quelques exemples de surcharge du constructeur d'une classe.

```
package com.tuto.company.other;
public class A {
   /* Déclaration des champs */
    int x;
    double y;
    String z;

    /* Constructeur 1: constructeur par défaut */
    public A (){
    }
    /* Constructeur 2: constructeur standard: x,y et z connus */
    public A (int x, double y, String z){
        this.x=x;
        this.y=y;
        this.z=z;
    }
     /* Autres constructeurs: surcharges */

    /* Constructeur 3: deux arguments */
    public A (double y, String z){
        this.x=Integer.MAX_VALUE;
        this.y=y;
        this.z=z;
    }
```

```
    /* Constructeur 4: 1 argument */
    public A (Double y){
    }
    /* Constructeur 5: 1 argument */
    public A (String z){
        this();
        this.z=z;
    }

}
```

Dans l'exemple ci-dessus, nous définissons une classe à trois champs : x, y et z. Nous avons également défini quatre constructeurs en plus du constructeur par défaut ; chacun correspondant à une surcharge particulière.

Constructeur 1 : constructeur par défaut

D'abord concernant le constructeur 1 (constructeur par défaut), il est défini sans argument. Pour instancier la classe A avec ce constructeur, il suffisant de spécifier une ligne comme par exemple :

```
A a = new A() ;
```

A noter que lorsqu'on instancie un objet avec le constructeur par défaut, l'objet est créé en initialisant les champs aux valeurs par défaut. Les champs *int* et *double* sont initialisés à 0 tandis que les champs String et les champs de type objet de classe sont initialisés à null.

Constructeur 2 : le constructeur standard

On appelle le constructeur standard pour instancier la classe lorsque la valeur de tous les champs sont connus. Ex :

```
A a =new A (15, 80.0, "Alfred");
```

Constructeur 3

Le constructeur 3 est une surcharge du constructeur 2 dans laquelle on fixe par défaut la valeur du champ x à la valeur maximum des nombres entiers (Integer.***MAX_VALUE***). Seuls les champs y et z restent à spécifier en argument lors de l'instanciation de la classe. Ex :

```
A a =new A (80.0, "Alfred");
```

Constructeur 4,

Le constructeur 4 est une surcharge qui prend un seul argument : le champ z. En appelant ce constructeur les champs x et y seront définis avec les valeurs par défaut comme pour le constructeur par défaut. Seule la valeur du z sera égale à valeur spécifiée en argument du constructeur. Ex :

```
A a = new A(15.0);
```

Le constructeur 5

Le constructeur 5 est un constructeur à un seul argument, mais il a la particularité d'appeler d'abord le constructeur par défaut avant de définir la valeur de z. En effet, l'instruction **this**(); est un raccourci l'instruction new A ();. Autrement dit le constructeur 5 fait d'abord appel au constructeur par défaut pour initialiser tous les champs aux valeurs par défaut. Il modifie la valeur de z pour lui assigner celle spécifiée en paramètre. Ex :

```
A a = new A("Alfred");
```

A noter que le constructeur 4 et le constructeur 5 prennent tous les deux un seul argument. La différence est pour le constructeur 4 l'argument est de type double alors que pour le constructeur 5, l'argument est de type *String*. Mais grâce aux propriétés de la surcharge, l'exécuteur Java sait automatiquement quel constructeur il doit appeler en fonction du type du paramètre indiqué.

4.5.2 Surcharge de méthodes

La surcharge de méthodes fonctionne sur le même principe que la surcharge de constructeurs. Elle consiste à définir plusieurs fonctions portant le même nom mais ayant des signatures différentes c'est-à-dire ayant des arguments différents. Pour illustrer la surcharge de méthodes partons de l'exemple d'une classe A qui définit une méthode permettant d'additionner des nombres. Voir exemple ci-dessous.

```
package com.tuto.company.other;

public class A {

    /* Méthode 1: première définition */
    public int addNumbers (int x, int y ){
        return x+y;
    }

    /* Méthode 2:  */
    public int addNumbers (int x, int y, int z ){
        return x+y+z;
    }

    /* Méthode 3 */
    public double addNumbers (double  x, double y ){
        return x+y;
    }

}
```

Dans cette classe nous avons défini trois fois la méthode *addNumber()*. La première définition prend en paramètre deux nombres de type *int* et renvoie la somme (qui est également de type *int*). La deuxième méthode prend en paramètre trois nombres de type

int et renvoie également la somme (type int). La troisième prend deux nombres en paramètres. Mais cette fois, à la différence de la première méthode les nombres sont de type double et le résultat renvoyé est aussi de type double.

Ces trois cas sont des exemples simples pour illustrer la surcharge de méthodes. Et le principe reste le même quelle que soit la complexité de la méthode. En somme, l'exécuteur Java choisi la méthode en fonction du nombre et du type des paramètres spécifiés.

4.6 Classes imbriquées

En Java la règle habituelle est de créer une classe par fichier source. Ainsi le fichier dans lequel est défini la classe porte le même nom que la classe. Et cette classe est définie avec le qualificateur *public* ou *protected* afin de la rendre visible par toutes les autres classes ou par les classes du même package. Cependant, définir une classe par fichier n'est pas toujours judicieux. Java offre la possibilité de définir des classes à l'intérieur d'autres classes. On les appelle classes imbriquées (nested *classes*). La classe dans laquelle sont imbriquées les autres classes est appelée *outer class* (classe englobante) et les classes qui sont portées sont appelées *inner classes* (classes englobées). La vocation d'une inner class est d'être directement utilisée dans la classe outer pour effectuer un traitement. Mais il arrive que la inner class soit utilisée à l'extérieur de l'outer classe notamment pour effectuer des traitements dans d'autres classes.

L'utilisation des classes imbriquées offre plusieurs avantages. D'abord, elle permet de rationaliser la dépendance entre les classes en procédant à un regroupement logique des classes utilisées pour réaliser le même traitement. Ensuite, elle permet de renforcer le principe d'encapsulation en positionnant au même endroit les classes qui sont autorisées à accéder à tel ou tel membre de classe. Aussi, l'imbrication des classes permet dans certaines situations de faciliter la lecture du code dans la mesure où elle permet de définir les classes et les rapprocher de l'endroit où elles sont utilisées sans avoir à parcourir à chaque fois l'arborescence du projet pour retrouver une classe. Néanmoins, il faut légèrement nuancer cette dernière remarque à cause du fait qu'une trop grande imbrication des classes peut produire l'effet inverse. C'est-à-dire une complexification de la structure des classes. Par conséquent, l'imbrication des classes doit être utilisée à bon escient.

De façon pratique, on distingue plusieurs types d'inner classes : les inner classes standards (non statiques), les inner classes statiques, les inner classes locales et les inner classes anonymes. Dans cette section, nous allons présenter chacun de ces types de classes imbriquées.

4.6.1 Inner class standard (non static)

Une inner class standard est une classe imbriquée qui est instanciée soit à l'intérieur de la classe englobante pour effectuer des traitements internes, soit utilisée à l'extérieur de la classe externe mais seulement après avoir instanciée la classe englobante. Les exemples ci-dessous illustrent les deux cas d'utilisation d'une inner class standard.

4.6.1.1 Cas où l'inner class est utilisée directement à l'intérieur de l'outer class

```
package com.tuto.company.other;
public class MyOuterClass {
    int x;
    String y;
    MyInnerClass myInnerObject;

    // inner class standard
    private class MyInnerClass {
        int a=5;
        int b=3;
        public void printTotal() {
            System.out.println("Total "+this.a+this.b);
        }
    }
    // Instancier la classe Inner pour définir l'attribut myInnerObject de la
classe Outer.
    void setObjet() {
        this.myInnerObject=new MyInnerClass();
    }
}
```

Dans cet exemple, la classe MyInnerClass est imbriquée dans la classe MyOuterClass. Et elle est directement instanciée dans cette classe pour pouvoir définir le champ myInnerObject. Par ailleurs, étant donné que la classe MyInnerClass est utilisée uniquement dans la classe MyOuterClass, elle est déclarée avec le qualificateur private.

4.6.1.2 Cas où l'inner class est utilisée hors de l'outer class

Il arrive qu'on définisse une inner class mais que cette inner class soit utilisée à l'extérieur de la classe englobante. Cette situation arrive par exemple lorsque l'inner class utilise certaines informations de la classe outer mais que ces informations sont déclarées private c'est-à-dire inaccessibles aux objets non-membres de la classe. L'exemple ci-dessous illustre l'utilisation de l'inner class utilisée hors de l'outer class.

```
package com.tuto.company.other;
public class MyOuterClass {
    private int x;
    private String y;

    // Constructeur de l'outter class
    public MyOuterClass(int x, String y){
        this.x=x;
        this.y=y;
    }
    // Définition de l'inner class
    public class MyInnerClass {
        int a=MyOuterClass.this.x;
        String b=MyOuterClass.this.y;
```

```
        public int getA(){return this.a;}
        public String getB(){return this.b;}

    }
}
```

Dans cet exemple l'outer class définit deux attributs private x et y. Ces attributs sont accessibles uniquement à l'intérieur de la classe. La classe inner définit deux attributs a et b qui prennent respectivement la valeur de x et y de la class outer. De plus, la classe inner fournit deux méthodes getA() et getB() qui permettent de renvoyer les valeurs de a et b par ricochet les valeurs de x et y. L'inner class MyInnerClass étant définie public, on peut l'appeler à l'extérieur de la classe MyOuterClass. Mais cet appel se fait avec une certaine subtilité. L'exemple ci-dessous montre comment instancier la classe MyInnerClass.

```
package com.tuto.company;

import com.tuto.company.other.*;

public class MyCaller {

        MyOuterClass outer=new MyOuterClass(15,"Hello"); // on instancie
l'outer class
        MyOuterClass.MyInnerClass inner = outer.new MyInnerClass(); // On
instance l'inner class à partir l'objet outer
        System.out.println(inner.getA()+" "+inner.getB()); // On appelle les
méthode de l'objet inner

}
```

Ici, on définit une classe appelante nommé MyCaller qui va appeler l'inner class MyInnerClass. Pour cela, on instancie d'abord la classe MyOuterClass pour créer l'objet outer. Ensuite, on instancie l'inner class MyInnerClass à partir de l'objet créé (notez la présence de l'opération new). On crée ainsi l'objet inner dont le type déclaré est MyOuterClass.MyInnerClass.

L'objet inner étant créé, on peut maintenant appeler ses méthodes notamment getA() et getB(). Telle est l'illustration de l'usage d'une classe inner à l'extérieur de la classe englobante.

4.6.2 Inner class statique

A l'inverse d'une inner class standard, une inner class statique est une inner classequi ne nécessite pas d'être instanciée pour pouvoir être utilisée. Une inner class est déclarée avec le mot-clé static. Très généralement, une inner class a pour but d'exposer à l'extérieur les membres (champs et méthodes) de l'outer class lorsque ces membres sont déclarés en mode static et private. L'exemple ci-dessous illustre l'utilisation d'une inner class statique.

```
package com.tuto.company.other;
public class MyOuterClass {
    static private int x=15;
    static private int y=20;
```

```
    static private int sum(){
        return x+y;
    }
    // Définition de l'inner class
    public static class MyInnerClass {
        public int getX(){
            return x;
        };
        public int getY(){
            return y;
        };
        public int getSum(){
            return sum();
        };
    }
}
```

Dans cet exemple, l'outer class MyOuterClasse déclare trois membres de type private. Il s'agit des champs x et y et de la méthode sum(). Tous ces membres sont static c'est-à-dire qu'il ne sont pas liés à une instance précise de l'outer clase. Mais étant donné que ces membres sont déclarés en private, ils ne sont pas accessibles par les objets extérieurs. L'inner class MyInnerClass a été définie ici pour pouvoir rendre ces membres de classe visibles depuis l'extérieur. Etant donné que la classe MyInnerClass est déclarée en public static, on l'appelle à l'extérieur de la classe MyOuterClass sans pour autant l'instancier. Seulement la classe MyOuterClass doit être rendue disponible avec l'opérateur new. L'exemple ci-dessous montre l'appel de la classe MyInnerClass à l'extérieur de la classe MyOuterClass.

```
package com.tuto.company;

import com.tuto.company.other.*;

public class MyCaller {

        MyOuterClass.MyInnerClass inner = new MyOuterClass.MyInnerClass(); //
On instance l'inner class
        // On appelle ses méthodes pour exposer les champs de la classe Outer
        System.out.println(inner.getX());
        System.out.println(inner.getY());
        System.out.println(inner.getSum());
}
```

NB : lorsqu'une inner class est déclarée static, elle n'accepte que les membres static de l'outer class. Tous les membres non static de l'outer class sont rejetés car ce sont des membres d'instance, c'est-à-dire que leurs valeurs peuvent varier d'une instance à une autre. Alors que pour les membres static, les valeurs des champs restent figées.

4.6.3 Inner class locale

Une inner class locale est une classe imbriquée définie à l'intérieur d'une méthode de la classe englobante. L'utilisation de l'inner class locale se limite exclusivement à la méthode

dans laquelle elle est définie que cette méthode soit private, protected ou public. L'exemple ci-dessous illustre l'utilisation d'une inner class locale.

```
package com.tuto.company.other;
public class MyOuterClass {

    String greeting="Hello, votre message est :";

     public void customPrint(String message) {

         // Définition l'inner classe
         class MyInnerClass {
            private String message;

                public MyInnerClass(String message) {
                    this.message = message;                }

                public void print() {

System.out.println(MyOuterClass.this.greeting+this.message);
                }
             }
         MyInnerClass customPrinter = new MyInnerClass(message);
         customPrinter.print();
    }

}
```

Dans cet exemple, on définit l'inner class dans une méthode de la classe MyOuterClass appelée *customPrint()*. A l'intérieur de la méthode, on définit la classe MyInnerClass qui elle-même définit une méthode print(). La méthode print() concatène le champ greeting de l'outer class avec un message spécifié par l'utilisateur pour renvoyer un message customisé. Pour envoyer ce message customisé, on instancie d'abord l'inner class dans la méthode *customPrint()* et on appelle la méthode print() de l'inner class. Ainsi, partout où l'outer class sera instanciée on peut appeler sa méthode *customPrint()* pour afficher. L'exemple ci-dessous montre l'appel de la méthode *customPrint()*.

```
package com.tuto.company;

import com.tuto.company.other.*;

public class Main {
    public static void main(String[] args) {

        MyOuterClass outer = new MyOuterClass(); // On instance l'inner class
        // On appelle la méthode contenant l'inner class
        outer.customPrint("Ceci est une inner classe locale");


    }
}
```

En exécutant ce code, on reçoit le message suivant :

```
Hello, votre message est :Ceci est une inner classe locale
```

4.6.4 Inner class anonyme

Comme son nom l'indique, une classe anonyme est une classe dont la référence n'est pas matérialisée par un nom. Généralement, la classe anonyme instancie à la volée une classe interface ou une classe abstraite et redéfinit certaines de ses méthodes, le tout à l'intérieur d'un bloc d'instructions. Et à la fin de ce bloc d'instructions il ne sera plus possible d'accéder à l'objet précédemment instancié. Les classes anonymes sont souvent utilisées sous formes de inner classes notamment dans des traitements à la volée. L'exemple ci-dessous montre un cas d'utilisation d'inner class anonyme.

Admettons qu'on dispose d'une classe abstraite nommée *Greeting* stockée dans un package à part et définie comme suit[8] :

```
package com.tuto.company.abstracts;
public abstract class Greeting {
    public abstract void greeting();
}
```

Supposons que cette classe abstraite soit dédiée à envoyer des salutations dans toutes les langues. On prévoit pour cela une méthode nommée greeting(). Cette méthode n'est pas encore implémentée. Et on souhaite laisser la possibilité à l'utilisateur d'implémenter cette méthode à sa guise.

Admettons maintenant que l'utilisateur veuille implémenter à la volée sans avoir d'abord à créer une classe distincte qui implémente d'abord la classe *Greeting* et ensuite qui définit la méthode greeting(), et enfin instancier la classe implémentée pour pouvoir utiliser la méthode implémentée. Pour raccourcir ce processus, on peut utiliser une classe anonyme et l'utiliser directement à l'intérieur de notre classe de travail. Telle est l'une des utilités des inner classes anonymes. L'exemple ci-dessous illustre l'utilisation d'une inner classe anonyme utilisant de la classe abstraite Greeting.

```
package com.tuto.company.other;
import com.tuto.company.abstracts.*;
public class MyOuterClass {

    String greeting="Bonjour, comment allez-vous ?";

    public void customGreeting() {

        Greeting innerAnonymous = new Greeting() {
            @Override
```

[8] Une classe abstraite est une classe dont les méthodes sont prévues mais ne sont pas encore implémentées. La déclaration d'une classe abstraite se fait avec le mot-clé abstract. Nous reviendrons plus tard sur les classes abstraites.

```
            public void greeting() {
                System.out.println(MyOuterClass.this.greeting);
            }
        };
        innerAnonymous.greeting();

    }

}
```

Ainsi on peut maintenant appeler la méthode *customGreeting()* de l'outer class comme suit :

```
package com.tuto.company;

import com.tuto.company.other.*;

public class Main {
    public static void main(String[] args) {

        MyOuterClass outer = new MyOuterClass(); // On instance l'inner class
        // On appelle la méthode contenant l'inner class
        outer.customGreeting();
    }
}
```

4.7 Héritage de classe

4.7.1 Le concept d'héritage de classe

L'héritage de classe est un autre concept central de la Programmation Orientée-Objet. Son principe consiste à dériver des nouvelles classes à partir des classes existantes afin d'utiliser les fonctionnalités de ces dernières sans avoir à les développer une nouvelle fois. Il arrive que dans un projet, plusieurs classes partagent les mêmes informations réutilisables. Grâce à l'héritage de classe, on peut réorganiser les définitions des classes de telle sorte que les informations réutilisables soient centralisées dans une ou plusieurs classes de base qui serviront ainsi de socle pour définir des classes dérivées. Ainsi la classe principale est appelée super-classe. Tandis que la classe qui est dérivée est appelée sous-classe.

En Java, on peut dériver autant sous-classes qu'on souhaite à partir d'une super-classe. En revanche, une sous-classe ne peut dériver que d'une et seule super-classe, contrairement à d'autres langages comme C++ qui permettent qu'une sous-classes puisse hériter de plusieurs super-classes.

Notons que l'héritage de classe s'applique aussi bien sur les classes initialement développées dans le projet mais aussi sur des classes provenant des librairies Java (natives ou externes).

4.7.2 Définir une sous-classe : le mot-clé extends

Pour hériter une classe B d'une classe A, on utilise le mot-clé *extends*. La syntaxe ci-dessous montre la structure d'une déclaration d'héritage de classe.

```
package nomPackage;

public class A {

 public A (){}

}
```

```
package nomPackage;

public class B extends A {

    public B (){
        super();
    }

}
```

Dans cette syntaxe, la classe B est déclarée avec le mot clé *extends* pour signifier qu'elle hérite de la classe A, c'est-à-dire qu'elle hérite de l'ensemble des champs et méthodes de la classe A. Ensuite, le constructeur de la classe B fait appel au constructeur de la classe A afin d'instancier un objet de la classe A et initialiser les valeurs de champs définis au niveau de A.

En pratique, l'héritage de classe vise soit à étendre les capacités de la super-classe en ajoutant de nouvelles fonctionnalités, soit à surcharger les fonctionnalités déjà existantes soit à redéfinir ces fonctionnalités. Pour illustrer chacun de ces usages à travers un exemple concret, prenons le cas de la classe *Employe* telle que définie ci-dessous.

```
package com.tuto.company.entite;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.Year;
import java.util.Date;

public class Employe{

    protected String idEmploye;
    protected int anneeNaissance;
    protected  int anneeEmbauche;
    protected double salaire;
```

```
    public Employe(String idEmploye, int anneeNaissance, int anneeEmbauche,
double salaire ){
        this.idEmploye=idEmploye;
        this.anneeNaissance=anneeNaissance;
        this.anneeEmbauche=anneeEmbauche;
        this.salaire=salaire;
    }
    public Employe( ){
    }
    public String getIdEmploye(){return this.idEmploye;}

    // Méthode à surcharger dans la sous-classe Vendeur
    public void setAnneeNaissance(int anneeNaissance){
        this.anneeNaissance=anneeNaissance;}

    // Méthode à redéfinir dans la sous-classe Vendeur
    public double calculPrime(){
        double prime=0.01*this.salaire;
        return prime;
    }


}
```

La classe *Employe* définit quatre champs (*idEmploye, anneeNaissance, anneeEmbauche* et *salaire*) et trois méthodes : *getIdEmploye(), setAnneeNaissance()* et *calculPrime()*.

Nous voulons hériter une classe nommée Vendeur à partir de la classe *Employe* en appliquant les trois visées de l'héritage de classe à savoir : ajouter de nouveaux champs ou méthodes, surcharger les méthodes existantes ou les redéfinir. Chacun de ces points sera détaillé dans une sous-section dédiée. Mais commençons d'abord par présenter la manière par laquelle la classe Vendeur hérite de la classe *Employe*.

```
package com.tuto.company.entite;

import com.tuto.company.entite.Employe;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.Year;
import java.util.Date;

public class Vendeur extends Employe {

    // Ajou d'un nouveau champ
    private  int age;

    public Vendeur(String idEmploye, int anneeNaissance, int anneeEmbauche,
double salaire ){
        super(idEmploye, anneeNaissance, anneeEmbauche, salaire );
    }
    public String getIdEmploye(){return this.idEmploye;}

    // Surcharge de la méthode  setAnneeNaissance()
    public void setAnneeNaissance(String dateNaissance) throws ParseException {
        Date dateNaiss=new SimpleDateFormat("dd/MM/yyyy").parse(dateNaissance);
```

```
        this.anneeNaissance=dateNaiss.getYear();;
    }

    // Redéfinition de la méthode calculPrime()
    @Override
    public double calculPrime(){
        double prime=0.01*this.salaire+ 30*(Year.now().getValue()-
this.anneeEmbauche);
        return prime;
    }

    // Ajout d'une nouvelle méthode ( extention)
    public void setAge(){
        this.age= Year.now().getValue()-this.anneeNaissance;
    }


}
```

D'abord la classe Vendeur définit un attribut supplémentaire qui est *age* et définit une méthode nommée *setAge()*. L'ajout du champ age et de la méthode *setAge()* constitue une première extension de la super-classe *Employe*.

Ensuite la classe Vendeur *setAnneeNaissance()* en proposant un nouvel paramétrage qui est non pas la valeur directe de l'année (qui est de type *int*), mais plutôt un paramètre date de naissance fournit avec le type String. Dès lors, la classe Vendeur propose deux versions de la méthode *setAnneeNaissance()*, une qui vient de la super-classe *Employe* et une qui est définit dans la sous-classe elle-même. La surcharge de la méthode *setAnneeNaissance()* constitue également une extension des fonctionnalité de la classe *Employe*.

Enfin, la classe Vendeur redéfinit la méthode *calculPrime()* en modifiant la formule de calcul. En effet dans la classe *Employe*, le montant de la prime est calculé comme 1% du montant du salaire. Mais dans la classe Vendeur, la prime est calculée comme 1% du salaire additionnée à 30 fois le nombre d'années d'ancienneté ; l'ancienneté étant définie comme la différence entre l'année courante et l'année d'embauche. La modification de la méthode *calculPrime()* sans changer ni la signature, ni le type de la valeur renvoyée est appelée redéfinition. Une méthode redéfinie est identifiée par l'annotation *@override*.

Dans les sous-sections qui suivent, nous ferons une distinction détaillée entre l'extension de la super-classe (à travers l'ajout de nouveaux membres), la surcharge des méthodes héritées (à travers le changement de signature des méthodes héritées) et la redéfinition des méthodes héritées (à travers la modification des corps des instructions).

4.7.3 Etendre la classe en ajoutant des nouveaux champs ou des nouvelles méthodes

La première possibilité offerte par l'héritage de classe est de pouvoir enrichir la classe principale (super-classe) en ajoutant des nouveaux membres (champs ou méthodes).

L'ajout de nouveaux membres à une classe héritée est appelée extension ou enrichissement car il s'agit d'ajouter des fonctionnalités qui n'existait pas avant. Dans l'exemple présentée ci-dessous, la classe Vendeur qui hérite de la classe *Employe* ajoute un nouveau champ (age) et une nouvelle méthode (*setAge()*).

L'ajout de nouveaux champs ou de nouvelles méthodes constituent la première forme d'extension ou d'enrichissement de la classe héritée. Mais il existe une autre forme d'enrichissement de la classe principale qui consiste à surcharger les méthodes existantes (voir ci-dessous).

4.7.4 Surcharger les méthodes de la super classe

La surcharge d'une méthode consiste à proposer une nouvelle méthode ayant le même nom mais pas la même signature (c'est-à-dire n'ayant pas le même nombre de paramètres et/ou les paramètres n'ont pas les mêmes types) ou un changement du type de la valeur retournée.

Dans l'exemple précédemment présenté, la classe Vendeur surcharge la méthode *setAnneeNaissance()* en changeant la signature. Dans la super-classe cette méthode prend un paramètre de type *int* et renvoie une valeur de type *int*. Dans la classe Vendeur, la nouvelle méthode prend cette fois un paramètre de type String qui correspond à la date de naissance. Mais la valeur de retour reste toujours l'année qui est toujours de type *int*.

A noter que lorsqu'une méthode est surchargée, la méthode intiale et la nouvelle méthode sont toutes disponibles dans la sous-classe sans que l'une masque l'autre.

4.7.5 Redéfinir des méthodes existantes dans la classe de base

La troisième possibilité offerte par le mécanisme d'héritage de classe est la redéfinition des méthodes déjà existantes dans la classe de base sans créer une autre version des mêmes méthodes comme c'est le cas de la surcharge. On parle alors de substitution. A la différence d'une méthode surchargée qui apporte une autre version de la méthode, une méthode redéfinie remplace complètement la méthode initiale.

Dans l'exemple présenté plus haut, la méthode calc*ulPrime()* définie dans la classe Vendeur remplace celle qui avait été déjà définie dans la classe *Employe*.

Pour indiquer qu'une méthode de la sous-classe constitue une redéfinition de la classe de la même méthode dans la super-classe, on utilise l'annotation *@Override*. Cette annotation sert simplement à indiquer au compilateur que cette méthode est une redéfinition d'une méthode portant le même nom dans la classe principale. A noter que cette annotation n'est pas obligatoire, elle est simplement recommandée afin d'une part fournir des informations complémentaires au compilateur et d'autre part faciliter la lecture du code source.

Noter par ailleurs que la redéfinition de méthodes n'est pas toujours possible dans l'héritage de classe. En effet, lorsqu'une méthode est déclarée avec le qualificateur final dans la classe principale, il n'est plus possible de redéfinir cette méthode dans la sous-classe. D'ailleurs

de manière générale, lorsqu'une classe est déclarée final, il est plus possible d'hériter cette classe. En fait, le mot-clé final constitue un verrou dans l'héritage de classe.

4.7.6 Accès aux membres de la classe de base

Par défaut, une classe dérivée a accès à tous les membres (champs et méthodes) déclarés public dans la classe parente. C'est pour cette raison qu'on peut appeler une méthode de la super-classe directement sur un objet créé en instanciant la sous-classe. Car par principe, l'héritage ramène tous les membres de la classe principale dans la classe dérivée sans avoir à les re-spécifier. Par exemple dans l'exemple présentée en début de section, un objet de la classe Vendeur peut appeler la méthode *getIdEmploye()* qui n'est pourtant définie que dans la classe *Employe*. Par exemple on peut instancier la classe Vendeur et appeler ces méthodes comme suit :

```
package com.tuto.company;

import com.tuto.company.entite.*;

import java.text.ParseException;

public class Main {
    public static void main(String[] args) throws ParseException {

     Vendeur vendeur =new Vendeur("v0045A", 2000, 2022, 1500 );
        vendeur.getIdEmploye();
        vendeur.setAnneeNaissance("23/05/1998");
        vendeur.calculPrime();
        vendeur.setAge();
    }
}
```

Dans cet exemple, nous appelons bien la méthode *getIdEmploye()* sur l'objet vendeur alors que cette méthode est définie dans la classe *Employe*. Ce qui illustre bien le principe d'héritage.

En revanche, il n'est pas possible d'accéder directement aux membres déclarés private dans la classe principale. En effet lorsqu'un champ est déclaré private dans la super-classe, pour accéder à ces champs dans la sous-classe, il faut passer par des méthodes déclarées public dans la super-classe. C'est le cas par exemple de la méthode *getIdEmploye()* déclarée en public et qui permet de renvoyer la valeur du champ *IdEmploye* définit en private. Par contre lorsqu'une méthode est définie en private dans la classe principale, il ne sera pas possible d'appeler cette méthode sur un objet de la classe dérivée.

Enfin lorsqu'un membre de classe est déclaré *protected* dans la classe principale, seule les sous-classes se trouvant dans le package de la classe principale auront accès. Les sous-classes définies se situant dans d'autres packages pourront accéder aux champs en passant par des méthodes déclarées public dans la super-classe. Mais elles ne pourront pas accéder aux méthodes déclarées en *protected*.

4.7.7 Appel du constructeur de la classe principale : le mot-clé super

Il faut remarquer d'entrée de jeu que le mécanisme d'héritage de classe ne concerne pas directement le constructeur de la classe principale dans la mesure où le constructeur de la classe principale n'est ni surchargeable, ni redéfinissable dans la classe dérivée. En revanche, il est impossible d'instancier la classe dérivée sans appeler au préalable le constructeur de la classe principale. Car c'est à la suite de l'appel du constructeur de la classe principale que les champs et les méthodes hérités seront disponibles dans la classe héritante. L'appel du constructeur de la classe principale est donc obligatoire dans l'héritage de classe.

L'appel du constructeur de la classe principale se fait à l'intérieur du constructeur de la classe dérivée en utilisant le mot-clé *super()* qui fait référence à la super classe.

Il existe deux modes de spécification de l'instruction *super(…)*. Lorsqu'elle est spécifiée sans argument, cela signifie que c'est le constructeur par défaut de la classe principale qui sera appelée. En revanche, lorsqu'elle est spécifiée avec des arguments, c'est le constructeur qui dont la signature correspond à ces arguments qui sera appelé. A noter que le mot-clé *super(…)* avec ou sans argument doit être doit être la première instruction définie dans le bloc d'instructions du constructeur de la classe dérivée. Dans l'exemple de la classe Vendeur présentée en début de section, le constructeur a été défini comme suit :

```
public Vendeur(String idEmploye, int anneeNaissance, int anneeEmbauche, double salaire
){
        super(idEmploye, anneeNaissance, anneeEmbauche, salaire );
    }
```

Dans cette spécification, le constructeur de la classe Vendeur passe ses arguments au constructeur de la classe *Empoyee* à travers l'instruction super (…). Dans ce présent exemple, l'instruction *super()* étant spécifiée avec des arguments, c'est le constructeur de la classe *Employe* qui a la même signature que cette spécification qui sera appelée pour instancier la classe *Employe*.

4.8 Polymorphisme d'objet de classe

4.8.1 Notion de polymorphisme

Le polymorphisme est la faculté d'un objet à pouvoir être considéré comme l'instance de plusieurs classes liées entre elles par des relations d'héritage. Le polymorphisme est une conséquence directe de l'héritage de classe, en particulier de la redéfinition de méthodes dans les sous-classes (voir section 4.7). Grâce au polymorphisme un objet change automatiquement de type et adopte le comportement adapté en fonction de la méthode appelée au moment de l'exécution. Par exemple, soit un objet *o* pouvant être à la fois d'un type *A* et d'un *B* ; A et B étant deux classes liées par des relations d'héritage. Lorsqu'on appelle *o* avec la méthode *m1* définie dans la classe A, l'objet o prend implicitement le type

A. Et lorsqu'on appelle la méthode *m2* définie dans la classe B, l'objet prend implicitement le type B.

Pour chaque objet polymorphe, on distingue un type « réel » et un type « référence ». Le type réel est le type dans lequel l'objet a été instancié avec l'opérateur *new*. Tandis que le type référence est le type dans lequel l'objet a été déclaré. Supposons par exemple que l'objet *o* soit de type réel B et d'un type référence A, selon le principe de polymorphisme, on peut écrire :

```
A o =new B();
```

Ici A est le type référence et B est le type réel. C'est-à-dire lorsqu'on lance l'instruction o.getClass().getName(), on retrouvera « B » et non « A ». B reste donc la classe réelle de o.

Notons que pour que l'objet o puisse être créé comme tel, c'est-à-dire une instance de la classe B mais référencé avec la classe A, l'une des deux conditions doivent être satisfaites :

- ✓ A est une super-classe et B est une sous-classe de A.
- ✓ A est un interface et B (ou un des parents de B) implémente A.

Lorsque l'une de ces conditions est respectée, alors o est considéré comme un objet polymorphe capable d'adapter ses comportements en fonction des méthodes appelées, que ces méthodes soient définies uniquement dans A, uniquement dans B ou simultanément dans les deux classes.

Toutefois, le comportement de l'objet o de type référence A et de type réel B diffère significativement selon les cas :

- Lorsqu'une méthode m() est définie dans B et non définie A, alors il n'est pas possible d'appeler cette méthode sur l'objet o. Car à la compilation (compile time), le compilateur ne considère que le type référence c'est-à-dire A et n'autorise donc pas l'accès à B.
- Lorsque la méthode m() de B est une redéfinition de la même méthode de A, alors c'est la méthode redéfinie qui sera appelée lorsqu'on fait o.m().
- Si A est une classe interface[9] prévoyant la méthode m(), et lorsque la méthode m() n'est pas implémentée dans la classe B, le compilateur regarde d'abord s'il existe une autre classe qui implémentent la méthode m() et qui est une super-classe pour B. Le cas échéant la méthode m() trouvée dans cette classe parente de B sera appelée sur l'objet o.

Ces trois cas illustrent toute la subtilité du polymorphisme.

Deux principes rendent effectif le polymorphisme d'objet. Le premier principe est celui de la compatibilité entre le type réel et le type référence. En effet, les type B et A doivent être compatibles, c'est-à-dire qu'on doit pouvoir caster le type B en type A. Cette compatibilité est surtout garantie lorsque B est une sous-classe de A, c'est-à-dire que B a été définie en

[9] Nous reviendrons plus tard sur la notion de classe interface.

utilisant le mot-clé extends A ou d'une autre classe héritant de A. Le deuxième principe est celui de l'« association dynamique » des méthodes plus connue sous le terme anglais de *dynamic method binding ou encore late binding*. Le binding permet d'associer les méthodes aux objets. Cette association peut se faire soit à la compilation (compile time) ou à l'exécution (runtime). Lorsque l'association se fait en runtime, on parle de dynamic binding ou late binding. Contrairement à d'autres langages comme C++ qui font du earling binding, Java adopte le dynamic binding. Ainsi ce n'est qu'à l'exécution qu'est connue la méthode à appeler sur l'objet o.

En définitive, le polymorphisme d'un objet décrit la faculté de cet objet à choisir la méthode à exécuter en présence d'un type réel et d'un type référence.

4.8.2 Exemple d'illustration du polymorphisme

Soient deux classes Employe et Vendeur définies telles que la classe Vendeur est une sous-classe de la classe Employe. La classe Vendeur étend non seulement la classe Employe en ajoutant un nouveau champ et une nouvelle méthode qui n'existaient pas dans la classe Employe, mais aussi elle surcharge une méthode et redéfinie une autre. Nous ajoutons aussi une troisième classe nommée Main qui permet d'instancier les objets à partir des deux classes et d'appeler les méthodes définies. L'objectif étant de mieux illustrer les principes liés au polymorphisme d'objets.

Définition de la classe Employe

```
package com.tuto.company.entite;
public class Employe{

    protected String idEmploye;
    protected int anneeNaissance;
    protected  int anneeEmbauche;
    protected double salaire;

    public Employe(String idEmploye, int anneeNaissance, int anneeEmbauche,
double salaire ){
        this.idEmploye=idEmploye;
        this.anneeNaissance=anneeNaissance;
        this.anneeEmbauche=anneeEmbauche;
        this.salaire=salaire;
    }
    public Employe( ){
    }
    public String getIdEmploye(){return this.idEmploye;}

    // Méthode à surcharger dans la sous-classe
    public void setAnneeNaissance(int anneeNaissance){
        this.anneeNaissance=anneeNaissance;}

    // Méthode à redéfinir dans la sous-classe Vendeur
    public double calculPrime(){
        double prime=0.01*this.salaire;
        return prime;
    }
```

```
}
```

Définition de la classe Vendeur

```
package com.tuto.company.entite;

import com.tuto.company.entite.Employe;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.Year;
import java.util.Date;

public class Vendeur extends Employe {

    // Ajou d'un nouveau champ
    private  int age;

    public Vendeur(String idEmploye, int anneeNaissance, int anneeEmbauche,
double salaire ){
        super(idEmploye, anneeNaissance, anneeEmbauche, salaire );
    }
    public String getIdEmploye(){return this.idEmploye;}

    // Surcharge de la méthode  setAnneeNaissance()
    public void setAnneeNaissance(String dateNaissance) throws ParseException {
        Date dateNaiss=new SimpleDateFormat("dd/MM/yyyy").parse(dateNaissance);
        this.anneeNaissance=dateNaiss.getYear();;
    }

    // Redéfinition de la méthode calculPrime()
    @Override
    public double calculPrime(){
        double prime=0.01*this.salaire+ (Year.now().getValue()-
this.anneeEmbauche);
        return prime;
    }

    // Ajout d'une nouvelle méthode ( extention)
    public void setAge(){
        this.age= Year.now().getValue()-this.anneeNaissance;
    }


}
```

Définition de la classe Main

```
package com.tuto.company;
import com.tuto.company.entite.*;
import java.text.ParseException;

public class Main {
    public static void main(String[] args) throws ParseException {

        // Cas 1: type réel Vendeur, type référence Employe
        Employe o1 =new Vendeur("v0045A", 2000, 2022, 1500 );


        // Appels de méthodes sur l'objet o1
```

```
        o1.getIdEmploye();
        o1.setAnneeNaissance(1998);
        o1.calculPrime();


        // Convertir le type référence de o1 en type Vendeur
        Vendeur o2 = (Vendeur) o1;
        // Appels de méthodes sur l'objet o2
        o2.getIdEmploye();
        o2.setAnneeNaissance("23/05/1998");
        o2.calculPrime();
        o2.setAge();
    }

}
```

Tout d'abord, nous avons créé un objet o1 comme une instance de la classe Vendeur. Mais le type déclaré pour o1 est Employe. Cette spécification est légale car il y a une compatibilité entre la classe Vendeur et la classe Employe pour la simple raison que les deux classes sont liées par une relation d'héritage. En effet, la classe Vendeur étend la classe Employe de trois façons. D'abord, elle ajoute un nouvel champ (age) et une nouvelle méthode setAge(). Ces membres n'existaient pas dans la classe Employe. Ensuite, elle surcharge la méthode setAnneeNaissance(). Et enfin, elle redéfinit la méthode calculPrime(). Un tel cadre permet de mieux illustrer le polymorphisme.

Premièrement, on peut constater que l'objet o1 a un type réel et un type référence. Le type réel est Vendeur, car l'objet o1 est obtenu en utilisant l'opérateur new sur la classe Vendeur. En revanche l'objet o1 a été déclaré en type Employe. Ce qui représente son type référence.

Deuxièmement, après avoir créé l'objet o1, nous faisons un certain nombre d'appels de méthodes dont les détails sont fournis ci-dessous :

- *o1.getIdEmploye()* : dans cet appel c'est la méthode définie dans la classe Employe qui est appelée car cette méthode n'est définie que dans cette classe.
- *o1.setAnneeNaissance()* : Même si la classe Vendeur propose une surcharge de la méthode setAnneeNaissance, c'est la méthode définie dans la classe Employe qui est appelée. Car c'est la méthode dont la signature correspond à l'argument spécifié en paramètre. Dans la classe Vendeur, la méthode setAnneeNaissance() prend un paramètre de type String, alors que dans la classe Employe, elle prend un argument de type int. Etant donné que l'appel de la méthode sur l'objet o1 est fait en spécifiant un paramètre de type int (1998), alors c'est la méthode de la classe Employe qui est appelée.
- *o1.calculPrime()* : La méthode calculPrime() est définie dans la classe Employe. Mais la classe Vendeur propose une redéfinition de la méthode. En effet la formule de calcul pour un vendeur diffère pour un employé standard. C'est pourquoi dans la classe Vendeur une redéfinition de la méthode a été proposée. Cette redéfinition est marquée par l'annotation @Override. Et lorsqu'on appelle cette méthode sur l'objet o1, c'est la redéfinition disponible dans la classe Vendeur qui sera appelée.

- Remarquons ici qu'il est impossible d'appeler la méthode setAge() sur l'objet o1, car cette méthode n'est pas définie dans la classe Employe qui est le type référence de l'objet.

Dans l'exemple ci-dessous, nous avons changé le type référence de l'objet o1 en castant le type référence Employe en type référence Vendeur. Désormais, pour l'objet o2, le type référence est le même que le type réel (Vendeur). Dans cette nouvelle configuration, il y a moins de contrainte d'appel de méthodes. En effet, quand on appelle une méthode sur l'objet o2, cette méthode est d'abord recherchée dans la classe Vendeur. Si elle n'est pas disponible dans la classe vendeur, elle sera recherchée dans les classes parentes de la Vendeur en suivant séquentiellement la hiérarchie de parenté. Ici la seule classe parente étant Employe, alors la méthode sera recherchée dans cette classe. C'est le cas par exemple de la méthode getIdEmploye(). En dehors de cette méthode, toutes les autres méthodes appelées sur l'objet o2 sont déjà disponibles dans le type de référence Vendeur.

En résumé, nous pouvons dire que le polymorphisme consiste à choisir les méthodes à exécuter en fonction de l'organisation des méthodes entre le type de référence et le type réel d'un objet. En fonction de différentes situations, nous pouvons tirer plusieurs principes résumés ci-dessous.

Principe 1 : Lorsqu'une méthode est définie dans la classe référence et héritée dans la classe réelle, cette méthode est appelée à partir de la classe référence. On entend par classe référence la classe qui représente le type référence, et par classe réelle, la classe qui représente le type réel.

Principe 2 : Lorsqu'une méthode est définie dans la classe référence et « surchargée » dans la classe réelle, c'est la méthode dont la signature correspond aux arguments spécifiés lors de l'appel qui sera exécutée, que cette méthode soit située dans la classe référence ou dans la classe réelle.

Principe 3 : Lorsqu'une méthode est définie dans la classe référence et « redéfinie » dans la classe réelle, alors c'est la méthode redéfinie qui est choisie lors de l'appel sur un objet.

Principe 4 : Lorsque la méthode appelée sur l'objet est disponible dans la classe référence et non disponible dans la classe réelle, alors cette méthode sera recherchée dans les classes parentes de la classe réelle en suivant séquentiellement la hiérarchie de parenté.

4.9 Les classes abstraites

4.9.1 Le concept de classe abstraite

Une classe abstraite est une classe semi-finie dans laquelle certaines méthodes sont déclarées mais non implémentées, c'est-à-dire ne contenant aucun bloc d'instructions permettant de les utiliser dans la suite du traitement.

Le concept de classe abstraite s'inscrit dans le prolongement du concept d'héritage de classe. Nous avons déjà montré qu'on peut définir plusieurs sous-classes à partir d'une même classe principale appelée super-classe. Dans beaucoup d'applications, la mise en place de super-classes vise à centraliser certaines fonctionnalités génériques pouvant être utilisées, modulées et personnalisées par chacune de classes dérivées à leur façon. La classe principale contient alors un ensemble de méthodes implémentées et prêtes à l'emploi. Et les sous-classes utilisent directement ces méthodes soit en les surchargeant, soit en les redéfinissant, soit en les utilisant telle quelle. Les classes abstraites poussent le principe de l'héritage encore plus loin en mettant à la disposition de l'utilisateur une structure dans laquelle toutes les méthodes définies dans les sous-classes ne sont pas implémentées. Ces méthodes sont simplement déclarées. Et c'est à l'utilisateur de les implémenter en choisissant lui-même les blocs d'instructions à définir.

Ainsi, à la différence des super-classes, les classes abstraites n'implémentent pas toutes les méthodes. Certaines restent au stade déclaratif en spécifiant seulement la signature de la méthode.

Du fait que toutes ses méthodes ne sont pas implémentées, les classes abstraites ne sont pas instanciables. C'est-à-dire qu'on ne peut pas créer d'objet à partir de ces classes. Seules les classes qui les implémentent sont instanciables. Dans cette section, nous allons présenter les classes abstraites ainsi que leur mode d'utilisation dans un projet Java.

4.9.2 Définir une classe abstraite : l'usage du mot-clé abstract

Une classe abstraite est définie avec le mot-clé *abstract* qui précède la déclaration. L'exemple ci-dessous montre la création d'une classe abstraite nommée Figure(). C'est une classe abstraite représentant une figure géométrique quelconque : carré, rectangle, cercle, etc. L'objectif est de définir une classe abstraite qui va servir de base pour implémenter chacune de ces figures géométriques. La classe Figure peut être définie comme suit :

```
package com.tuto.figure;

public abstract class Figure {

 public static String versionId= "001";

 public Figure(){}

 abstract double perimete ();
 abstract double superficie ();

 public void info (){
  System.out.println("Ceci est une classe abstraite mais cette méthode n'est
pas abstraite");
 };

}
```

La classe Figure() est une classe abstraite formée de trois méthodes dont deux sont abstraites. Il s'agit de la méthode périmetre() qui vise à calculer le pourtour de la figure et

la méthode superficie() qui vise à calculer la surface de la figure. Rappelons qu'une méthode abstraite est une méthode déclarée mais non implémentée. Ce sont les méthodes abstraites qui rendent une classe abstraite. Car, dès qu'une méthode est déclarée abstraite dans une classe toute la classe est systématiquement considérée comme une classe abstraite même si elle contient déjà des méthodes implémentées. Dans l'exemple ci-dessous la classe Figure est déclarée abstraite même si la méthode info() est déjà définie et implémentée. A noter aussi que la classe Figure déclare un champ nommé versionId qui est une constante de type égale à « 001 ».

Remarquons ici que les méthodes abstraites perimetre() et superficie() ont été déclarées sans arguments. Nous avons fait ce choix pour pouvoir garder ces méthodes les plus génériques possibles. Mais habituellement les méthodes abstraites sont déclarées avec des signatures, c'est-à-dire les arguments et leur type. Par exemple, on pouvait imaginer une méthode abstract nommée position prenant les paramètres x et y et définie comme *abstract void position (double x, double y)*. Ceci permet d'indiquer que la méthode abstract est déclarée comme l'aurait été une méthode ordinaire.

Voici ci-dessous résumées les principales caractéristiques d'une classe abstraite.

- C'est une classe non instanciable. C'est-à-dire qu'on ne peut pas créer d'objet à partir de cette classe comme on aurait pu le faire avec les classes ordinaires.
- Les classes abstraites autorisent des constructeurs. Ceux-ci seront dans les constructeur des sous-classes qui implémentent les méthodes abstraites de la classe.
- Une classe abstraite ne peut pas être déclarée final sinon elle ne pourra pas être étendue et implémentée par des sous classes (principe de l'héritage de classe). De même, une méthode abstraite ne peut pas être déclarée final sinon elle ne pourra pas être implémentée par les classes qui étendront la classe parente. En somme les mots-clés final et abstract sont antinomiques.
- Comme dans une classe ordinaire, on peut définir des champs et des méthodes static dans une classe abstraite.
- Lorsqu'une classe contient au moins une méthode abstraite, alors toute la classe doit être déclarée comme abstraite.
- Lorsqu'une sous-classe étend une classe abstraite, celle-ci doit implémenter toutes les méthodes abstraites de la classe. Si tel n'est pas le cas, alors cette sous-classe doit à son tour être déclarée comme classe abstraite pour que d'autres classes puissent en hériter pour implémenter les méthodes restantes.

4.9.3 Implémenter une classe abstraite : usage du mot-clé extends

Dans la sous-section précédente, nous avons défini la classe abstraite Figure() et avons présenté ses principales caractéristiques. Dans la présente sous-section, nous allons proposer quelques implémentations de cette classe abstraite en prenant le cas de trois

figures : le carré, le rectangle et le cercle. Voici ci-dessous les différentes implémentations de la classe abstraite.

Définition de la classe Carre

```
package com.tuto.figure;
public class Carre extends Figure {
 private double cote;
 public Carre(double cote){
  super();
  this.cote=cote;
 }
 double perimetre (){
  return this.cote*4;
 };
 double superficie () {
  return Math.pow(this.cote, 2);
 };

 }
```

Définition de la classe Rectangle

```
package com.tuto.figure;
public class Rectangle extends Figure {
 private double longeur;
 private double largeur;
 public Rectangle(double longeur, double largeur){
  super();
  this.longeur=longeur;
  this.largeur=largeur;
 }
 double perimetre (){
  return 2*this.longeur+2*this.largeur;
 };
 double superficie () {
  return this.longeur*this.largeur;
 };

 }
```

Définition de la classe Cercle

```
public class Cercle extends Figure {
 private double rayon;
 public Cercle(double rayon){
  super();
  this.rayon=rayon;
 }
 double perimetre (){
  return Math.PI * (2 * this.rayon);
 };
 double superficie () {
  return Math.PI * Math.pow(this.rayon, 2);
```

```
};

}
```

Les trois classes définies ci-dessus proposent chacun une implémentation spécifique.

En réalité, utiliser le terme implémenter la classe est un abus de langage, car en effet, on n'implémente pas la classe abstraite, on implémente plutôt ses méthodes abstraites. Pour ce qui concerne la classe elle-même, on devrait plutôt dire qu'on hérite la classe. Car chaque classe est définie en utilisant le mot-clé *extends*. Ce qui correspond à une extension classique dans le cadre de l'héritage de classe où la classe abstraite représente la super-classe et les classes implémentant les méthodes représentent les sous-classes. De ce point de vue, l'abstraction de classe et l'implémentation de ses méthodes représentent simplement un cas spéciale d'héritage de classe. Par exemple, l'implémentation des méthodes abstraites dans les classes se confond pratiquement avec la redéfinition de méthodes dans une classe dérivée avec la seule particularité que dans la classe abstraite, les blocs d'instructions constituant les méthodes à redéfinir sont entièrement vides.

4.10 Les interfaces

4.10.1 Le concept de d'interface

Simplement définie, une interface est une structure de classe dans laquelle aucune méthode n'est encore implémentée. A la différence d'une classe ordinaire où toutes les méthodes sont implémentées et à l'inverse d'une classe abstraite où certaines méthodes sont implémentées, dans une interface aucune méthode n'est implémentée. L'interface ajoute encore une couche d'abstraction supplémentaire par rapport aux classes abstraites.

D'une manière générale, une interface peut être vue comme un contrat entre un fournisseur de service et ses clients. Pour mieux illustrer ces propos, prenons par exemple le cas d'une base de données telle que Oracle. Pour accéder au contenu d'une base, Oracle a défini un certain nombre de protocoles à respecter soit pour lire les données, soit pour écrire les données. Ces protocoles représentent le contrat entre le fournisseur de service, qui est ici Oracle et ses différents utilisateurs de services (clients). Admettons par exemple que ce protocole soit le JDBC (Java database connectivity). Et chaque client, compte tenue de la technologie qu'il utilise, doit développer un connecteur propre à lui et qui respecte le protocole JDBC. Le protocole JDBC représente alors l'interface, car il définit de manière générique le mode d'accès aux données sur la base Oracle. Cet exemple, bien qu'imparfait, illustre au mieux le concept d'interface. Au-delà de l'exemple du JDBC, de nombreuses API fonctionnent sur le principe d'interface offrant un protocole standard que chaque utilisateur doit implémenter à sa façon pour d'accéder aux services proposés par le fournisseur.

Prenant en compte l'ensemble des considérations évoquées, une interface peut alors être définie comme une classe proposant un ensemble de spécifications standards qui doivent

être implémentées dans des classes dérivées. En langage Java, ces spécifications standards sont représentées par les méthodes déclarées mais non encore implémentées.

Dans cette section, nous allons présenter les interfaces en langage Java, leurs caractéristiques et leur mode d'utilisation.

4.10.2 Définir une interface : usage du mot-clé interface

Comme nous l'avons déjà évoqué, une interface est une classe dans laquelle aucune méthode n'est implémentée, en d'autres termes toutes les méthodes sont abstraites, seules les signatures sont spécifiées.

Une interface est déclarée avec le mot-clé interface. L'exemple ci-dessous illustre la création d'une interface nommée Figure() qui représente une figure géométrique quelconque : carré, rectangle, cercle, etc.

```
package com.tuto.figure;

public interface Figure {
  public static String versionId= "001";
  double perimetre ();
  double superficie ();
  public void info ();

}
```

Ici, nous avons défini l'interface Figure en utilisant le mot-clé interface. Ensuite, nous avons déclaré trois méthodes abstraites que sont *perimetre(), superficie()* et *info()* qui sont prévues respectivement pour calculer le pourtour de la figure, la surface et fournir un descriptif sur la classe. Ces trois méthodes sont restées abstraites car les modes d'implémentation diffèrent d'une figure à l'autre (voir plus bas pour les implémentations).

A noter ici que les trois méthodes abstraites ont été déclarées sans signature (c'est-à-dire sans arguments) même si les types de retour ont été spécifiées. Ceci représente simplement un cas particulier, car les méthodes abstraites peuvent aussi être déclarées avec des signatures. Par exemple, on pouvait imaginer une méthode abstract nommée position prenant les paramètres x et y et définie comme *abstract void position (double x, double y)*.

Notons aussi que dans une interface, même si toutes les méthodes sont abstraites, on peut retrouver des champs dont la valeur est bien définie. Dans l'exemple de l'interface Figure définie ci-dessus, le champs nommé *idVersion* est bien défini et prend la valeur « 001 ». Ainsi, tout comme les classes ordinaires et les classes abstraites, les interfaces peuvent aussi contenir des données concrètes même si celles-ci ne sont accessibles qu'après implémentation de la classe.

Voici ci-dessous quelques propriétés caractérisant les classes interfaces :

- Une interface ne peut être instanciée.

- Une interface contient les déclarations et les signatures de méthodes ;
- Une interface peut hériter d'une autre interface en utilisant le mot-clé extends
- Une classe (abstraite ou non) peut implémenter plusieurs interfaces en même temps.

4.10.3 Implémentation d'une interface : usage du mot-clé implements

Dans cette section, nous présentons les modes d'implémentation d'une interface. Nous montrons surtout qu'une interface peut être implémentée soit sous forme d'une classe ordinaire, soit sous forme d'une classe abstraite. Les deux exemples de classes ci-dessous illustrent l'implémentation de l'interface Figure () précédemment présentée. La classe Carre implémente l'interface sous forme de classe ordinaire tandis que la classe Rectangle implémente l'interface sous forme de classe abstraite.

Définition de la classe Carre

```
package com.tuto.figure;
public class Carre implements Figure {
 private double cote;
 public Carre(double cote){
  this.cote=cote;
 }
 public double perimetre (){
  return this.cote*4;
 };
 public double superficie() {
  return Math.pow(this.cote, 2);
 };
 public void info (){
  System.out.println("La classe actuelle est "+this.getClass().getName());
 }

}
```

Définition de la classe Rectangle

```
package com.tuto.figure;
public abstract class Rectangle implements Figure {
 private double longeur;
 private double largeur;
 public Rectangle(double longeur, double largeur){
  this.longeur=longeur;
  this.largeur=largeur;
 }
 public double perimetre (){
  return 2*this.longeur+2*this.largeur;
 };
 public double superficie () {
  return this.longeur*this.largeur;
 };

 }
```

Quelques remarques sont à faire concernant les définitions de la classe Carre et de la classe Rectangle. D'abord, la classe Carre est une classe ordinaire car elle implémente toutes les méthodes abstraites de l'interface Figure. Cela veut donc dire qu'on peut instancier la classe Carre et ainsi créer un objet réel. L'exemple ci-dessous illustre une instance de la classe Carre :

```
package com.tuto.company;
import com.tuto.figure.Carre;
public class Main {
    public static void main(String[] args) {

        Carre carre =new Carre(20 );
        System.out.println(carre.perimetre());
        System.out.println(carre.superficie());
        carre.info();
    }

}
```

L'exécution de cette commande renvoie :

```
80.0
400.0
La classe actuelle est com.tuto.figure.Carre
```

Concernant la classe Rectangle, elle implémente les deux méthodes perimetre() et superficie() de l'interface Figure, mais elle n'implémente pas la méthode info(). Ce qui signifie que la classe Rectangle reste au stade de classe abstraite car au moins une méthode parmi celles qu'elle a héritées de l'interface Figure n'est pas implémentée. De ce fait, la classe Rectangle n'est pas encore instanciable (voir 4.9 pour les caractéristiques des classes abstraites). Pour pouvoir l'instancier, il faut d'abord définir une classe dérivée qui implémente la méthode info().

Au final, dans cette section, nous avons montré qu'on peut implémenter totalement ou partiellement les méthodes d'une interface. Lorsque toutes les méthodes sont implémentées, on aboutit alors à une classe ordinaire. En revanche lorsque toutes les méthodes ne sont pas implémentées, on obtient une classe abstraite, qui mérite à son tour d'être étendue pour implémenter les méthodes restantes.

4.11 Classes anonymes

Comme son nom l'indique, une classe anonyme est une classe sans nom explicite. En effet, d'habitude chaque classe, qu'il s'agisse d'une classe ordinaire, d'une classe abstraite ou d'une interface, est déclarée sous un nom spécifique et parfois définie dans un fichier dédié portant le même nom que la classe (à l'exception des classes internes). Une classe anonyme n'a pas une définition propre. Il s'agit d'une classe définie et instanciée à la volée au moment de l'exécution du traitement.

Une classe anonyme est souvent utilisée comme une inner classe locale définie à l'intérieur d'une méthode dans une autre classe. Elle est définit soit en implémentant une interface, soit en étendant une classe ordinaire comme sous-classe. Lors de la définition et de l'instanciation de la classe anonyme, c'est le nom de la classe qu'elle implémente ou étend qui est spécifiée. C'est d'ailleurs pourquoi, on l'appelle classe anonyme. Pour illustrer concrètement la notion de classe anonyme, supposons une classe interface nommée *Greeting* définie ci-dessous prévue pour envoyer des salutations personnalisées.

```
package com.tuto.interfaces;
public interface Greeting {
    public void sendGreeting();
}
```

Définissons maintenant une classe de traitement nommée *EnvoiSalutation* qui contient une méthode nommée *envoi()* servant à envoyer des messages personnalisés.

```
package com.tuto.others;
import com.tuto.interfaces.*;
public class EnvoiSalutation {

  public void envoi() {
       // Définition classe anonyme à partir de l'interface Greeting
      Greeting anonymeSalut = new Greeting() {
       @Override
       public void sendGreeting() {
        System.out.println("Bonjour, comment allez vous ?");
       }
      };
   anonymeSalut.sendGreeting();
 }

}
```

Nous souhaitons que la méthode *envoi()* de la classe *EnvoiSalutation* utilise la méthode *sendGreeting* déjà prévue dans la classe *Greeting*. Mais étant donné que la méthode *sendGreeting()* n'est pas implémentée (car la classe *Greeting* est une interface), alors pour pouvoir utiliser la méthode *sendGreeting()*, nous devons d'abord l'implémenter. Pour cela, nous instancions à la volée la classe Greeting et nous implémentons la méthode *sendGreeting()* pour construire l'objet *anonymeSalut*. Cet objet est l'instance non pas de la classe *Greeting* mais plutôt d'une classe anonyme qui implémente la classe *Greeting*[10].

[10] A noter que la classe Greeting pouvait aussi être une classe abstraite définie comme suit :

```
package com.tuto.interfaces;
public abstract class Greeting {
    public void sendGreeting(){};
}
```

Avec cette classe abstraite, le principe d'instanciation de la classe anonyme aurait été le même.
De même, la classe Greeting pouvait également être une classe ordinaire non final. La classe anonyme aurait simplement instancié cette classe et redéfinie la méthode sendGreeting() comme elle l'a fait pour la classe interface ou la classe abstraite.

Remarquons ici que la classe anonyme a été définie et instanciée à l'intérieur d'une méthode de la classe Salutation. La classe n'est donc pas visible à l'extérieur de cette méthode. C'est pourquoi les classes anonymes sont généralement des classes internes locales (voir la section 4.6 sur les classes imbriquées).

La classe *EnvoieSalutation* étant définie, nous pouvons maintenant concevoir une classe Main permettant d'instancier cette classe et d'appeler sa méthode *envoi()*. L'exemple ci-dessous constitue l'illustration.

```
package com.tuto.company;
import com.tuto.others.*;
public class Main {
    public static void main(String[] args) {

        EnvoiSalutation salutation=new EnvoiSalutation();
        salutation.envoi();
    }

}
```

L'exécution de ce bout de code renvoie :

```
Bonjour, comment allez vous ?
```

Voici ci-dessous résumées les principales caractéristiques d'une classe anonyme :

Les classes anonymes sont des classes internes locales. Elles sont définies à l'intérieur des méthodes d'autres classes. Elles ont donc une portée locale. Les classes anonymes implémentent les méthodes des interfaces, des classes abstraites ou redéfinissent des méthodes d'une classe ordinaire.

4.12 La classe Object : classe mère en langage Java

4.12.1 Présentation de la classe Object

Toutes les classes que nous créons héritent d'une seule et même classe : la classe Object[11]. En effet, l'organisation des classes Java forme une structure d'héritage hiérarchique unique dont la racine est la classe Object. Elle représente donc la classe mère en Java. Lorsque l'on crée une classe en déclarant sa définition sans préciser une relation d'héritage, tout se passe comme si Java ajoutait implicitement l'instruction *extends Object* pour indiquer que cette classe hérite de la classe Object. Il y a donc un héritage par défaut pour toutes les classes que nous créons. Par exemple, en définissant une classe A telle que :

```
public class A {
 public A (){}

}
```

[11] Les interfaces et les classes abstraites n'héritent pas directement de la classe Object mais ses méthodes sont disponibles dans ces classes par un mécanisme interne propre au langage Java.

Cette définition équivaut à écrire :

```
public class A extends Object {
 public A (){}

}
```

La classe A étant une sous-classe de la classe Object, toutes les méthodes de la classe Object sont donc disponibles dans la classe A qui peut alors les étendre, les surcharger ou les redéfinir selon les règles de l'héritage de classe (voir la section 4.7 concernant l'héritage de classe Java). La page suivante présente la structure et les caractéristiques de la classe Object https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html#equals(java.lang.Object)

4.12.2 Polymorphisme avec la classe Object

Comme nous l'avons déjà montré, le polymorphisme est une conséquence directe de l'héritage de classe. Pour rappel, le polymorphisme est la faculté d'un objet d'avoir plusieurs types et d'adapter l'appel des méthodes en fonction de chaque type suivant le contexte. Un objet dispose d'un type réel et d'un type référence. Le type réel correspond à la classe qui a été instanciée pour créer l'objet. Et le type référence est la classe qui a été utilisée pour référencer l'objet sachant qu'il doit y avoir une compatibilité entre les deux types. Prenons par exemple, le cas d'un objet o dont la classe réelle est A et la classe référence est Object, l'objet o peut être créé comme suit :

```
Object o = new A() ;
```

Cette instruction est bien légale car la classe A hérite directement de la classe Object.

Ici l'objet o est bien polymorphe car, en fonction du contexte, elle peut soit prendre la forme de classe Object (son type référence), soit prendre la forme de la classe A (son type réel). Par exemple, lorsqu'on appelle une méthode qui existe à la fois dans la classe Object et dans la classe A, l'objet o prend automatiquement son type réel A et exécute la méthode se trouvant dans A. Et lorsque la méthode existe uniquement dans la classe Object et non dans la classe A, l'objet o garde son type référence Object et exécute la méthode référencée. Enfin, lorsque la méthode existe uniquement dans la classe A (cas d'une extension), alors pour pouvoir exécuteur cette méthode, il faut d'abord caster l'objet o en son type réel A. Voici ci-dessous quelques exemples d'illustration (pour plus de détails sur le polymorphisme d'objet, voir la section 4.8).

Définition de la classe A

```
package com.tuto.company.other;
public class A  {
 public A (){}
```

```
 // Redéfinition
 @Override
 public String toString(){
  return this.getClass().getName();
 }
 //Extension
 public void affiche(){
  System.out.println("Cette méthode n'existe pas dans la classe object");
 }
}
```

Création de l'objet o et illustration des principes de polymorphismes

```
package com.tuto.company;
import com.tuto.company.other.*;
public class Main {
    public static void main(String[] args) {

        // Instancier l'objet o
        Object o = new A();
        // Appel de la méthode existant dans la classe Object et dans la classe
A
        System.out.println("La classe actuelle est :"+o.toString());
        // Appel de la méthode existant uniquement dans la classe Object
        System.out.println("Le Hashcode de la classe est :"+o.hashCode());
        // Appel de la méthode existant uniquement dans la classe A
        ((A)o).affiche();
    }
 }
```

En exécutant, ce code, nous obtenons les résultats suivants :

```
La classe actuelle est :com.tuto.company.other.A
Le Hashcode de la classe est :1595428806
Cette méthode n'existe pas dans la classe object
```

Dans cet exemple, l'objet o a été créé avec le type réel A mais référencé avec le type Object.

Nous avons d'abord appelé la méthode toString() sur l'objet o. Cette méthode existe à la fois dans la classe Object et dans la classe A. Dans ce cas c'est la méthode existante dans la classe A qui est exécutée.

Ensuite, nous avons appelé la méthode hashCode(). Mais étant donné que cette méthode existe seulement dans la classe Object, l'objet o garde son type réel et exécute la méthode hashCode().

Enfin, nous avons appelé la méthode affiche() qui est définie uniquement dans la classe A. Dans ce scénario, il faut d'abord caster le type référence de l'objet en A pour pouvoir appeler la méthode affiche(). D'où l'usage de l'opérateur de cast (A) sur l'objet o et ensuite l'appel de la méthode affiche() sur l'objet casté.

En résumé, le fait que toute classe Java hérite de la classe Object, les principes de polymorphisme s'appliquent à tout objet créé en instanciant une classe et référencé sous le type Object.

4.12.3 Quelques usages des méthodes de la classe Object

Grâce aux principes d'héritage de classe et de polymorphisme d'objet, il est possible d'appliquer à tout objet les méthodes disponibles dans la classe Object. Cette classe fournit un certain nombre de méthodes qui s'avérent utiles dans de nombreuses situations. Parmi ces méthodes, on dénote entre autres : getClass() et equals(). La méthode getClass() est utilisée pour récupérer la classe d'origine d'un objet. La méthode equals() est utilisée pour comparer deux objets. Ce qui peut s'avérer utile dans de nombreux cas de traitement. L'exemple ci-dessous illustre l'utilisation de la méthode getClass() et la méthode equals(). Pour plus de détail sur les méthodes de la classe Object, consulter la page : https://docs.oracle.com/javase/10/docs/api/java/lang/Class.html

Définition d'une classe A

```
package com.tuto.company.other;
public class A  {
 int x;
 int y;
 public A (int x, int y){
  this.x=x;
  this.y=y;
 }
 public void setX(int x){this.x=x;}
 public void setY(int y){this.y=y;}

}
```

Utilisation des méthodes getClass() et equals() de la classe Object

```
package com.tuto.company;
import com.tuto.company.other.*;
public class Main {
    public static void main(String[] args) {

        // Instancier l'objet o1
        Object o1 = new A(2,3);
        // Instancier l'objet o1
        Object o2 = new A(2,3);
        // Copie de l'objet o1
        Object o3=o1;

        // Utilisation de la méthode getClass()
        System.out.println(o1.getClass()); // Renvoie class A
        System.out.println(o1.getClass()); // Renvoie class A
        // Utilisation de la méthode equals
        System.out.println(o1.equals(o2)); // renvoie false
        System.out.println(o1.equals(o3)); // renvoie true

    }
 }
```

4.13 Gestion dynamique des objets : usage de la classe Class

4.13.1 Généralités

La gestion dynamique des objets (ou introspection de classe) est un mécanisme par lequel on peut accéder dynamiquement au contenu d'une classe et d'identifier les différentes méthodes et champs qui la composent sans passer par le code source. Traditionnellement, pour obtenir un objet on instancie d'abord une classe en utilisant l'opérateur new suivi du nom de la classe (ou plus exactement du nom du constructeur). Cette approche de création des objets n'est pas la seule dans le langage Java. En effet, Java offre la possibilité d'exploiter dynamiquement des classes en utilisant une classe spéciale appelée Class. La classe Class offre un ensemble de méthodes permettant de réaliser des opérations de manipulation de classes : création de classe à partir d'un nom spécifié en String, créer un objet de la classe sans utiliser l'opérateur new, lister et utiliser l'ensemble des méthodes, des champs et des qualificateurs, etc.

Le but de cette section est de présenter l'usage de l'usage de classe Class pour gérer dynamiquement les objets Java.

4.13.2 Présentation de la classe Class

La classe Class est une classe spéciale Java qui permet de représenter sous forme d'objet toutes les classes et interfaces lors de l'exécution du code.

De la même manière que toutes les classes Java héritent de la même classe Object[12], toutes les classes et interfaces peuvent aussi être représentées sous la forme d'un type générique représentée par la classe Class. A noter que la classe Class hérite elle-même de la classe Object. Toutefois la classe Class a plusieurs particularités par rapport à la classe Object. D'abord, la classe Class ne dispose pas de constructeur public comme pour la classe Object. De ce fait, on ne peut pas utiliser l'opérateur new pour créer une instance de la classe Class. Les instances de la classe Class sont construites uniquement par la JVM lors de l'exécution du code. Aussi, la classe Class est une classe générique prenant en paramètre un type T qui peut être de n'importe quel type Java. Ci-après quelques spécifications de paramètres de la classe Class : Class<Object>, Class<String>, Class<Integer>, Class<MyClass>, etc… Pour obtenir plus de détails sur la classe Class, consulter la page suivante : https://docs.oracle.com/javase/10/docs/api/java/lang/Class.html

4.13.3 Code source d'illustration : Code source CS06

Pour illustrer les usages de la classe Class ainsi que les différentes fonctionnalités d'introspection, nous allons utiliser un exemple code source représenté par la classe Employe définie ci-dessous.

Code source : CS06

[12] Voit section précédente pour plus de détails sur la classe Object.

```java
package com.tuto.introspection;

import java.time.Year;
public class Employe{
    private String nom;
    private String sexe;
    private int anneeNaissance;
    private int anneeEmbauche;
    private int anciennete;
    private int age;
    private double salaire ;

    /* Constructeur par Défaut */
    public Employe(  ){
    }
    /* Second constructeur constructeur */
    public Employe( String nom, String sexe, int anneeNaissance, int
anneeEmbauche, double salaire ){
        this.nom =nom;
        this.sexe=sexe;
        this.anneeNaissance=anneeNaissance;
        this.anneeEmbauche=anneeEmbauche;
        this.salaire=salaire ;
    }

    /* Méthodes */
    public String getNom(){return this.nom;}
    public void setNom(String nom){this.nom=nom;}
    public String getSexe(){return this.sexe;}
    public void setSexe(String sexe){this.sexe=sexe;}
    public int getAnneeNaissance(){return this.anneeNaissance;}
    public void setAnneeNaissance(int
anneeNaissance){this.anneeNaissance=anneeNaissance;}
    public int getAnneeEmbauche(){return this.anneeEmbauche;}
    public void setAnneeEmbauche(int
anneeEmbauche){this.anneeEmbauche=anneeEmbauche;}
    public int getAge(){return this.age;}
    private int getAnneeCourante(){
        return Year.now().getValue();
    }
    public void setAge(){
        this.age= getAnneeCourante()-this.anneeNaissance;
    }
    public int getAnciennete(){return this.anciennete;}
    public void setAnciennete(){
        this.anciennete= getAnneeCourante()-this.anneeEmbauche;
    }
    public double getSalaire(){return this.salaire;}
    public void setSalaire(double salaire){this.salaire=salaire;}
    public void augmentSalaire(double taux){
        if(taux<-1.0 || taux>1.0){
            System.out.println("Vous devez indiquer une valeur correcte du
taux\n La valeur doit être compris entre -1.0 et 1.0");
            System.exit(1);
        }
        this.salaire=this.salaire*(1+taux);}
}
```

Dans la classe *Employe*, sept champs sont déclarés : *nom, sexe, anneeNaissance, age, anneeEmbauche, anciennete* et *salaire*. Parmi ces sept champs, deux champs sont variables car leurs valeurs changent dynamiquement en fonction du temps. Il s'agit des champs *age* et *anciennete*. L'âge dépend de l'année de naissance et l'ancienneté dépend de l'année d'embauche de l'employé dans l'entreprise.

La classe Employe dispose de deux constructeurs : un constructeur par défaut et un constructeur spécifique dont la signature est formée par cinq champs parmi les sept déclarés. Les deux champs age et anciennete étant des champs variables, ils seront calculés en appelant des méthodes spécifiques.

S'agissant des méthodes disponibles dans la classe Employe, un *getter* et un *setter* a été défini pour chaque champ. Par exemple, pour le champ *nom* les méthodes définies sont *getNom()* et *setNom()*. Et pour les deux champs age et anciennete non initiliasés par le constructeur, nous avons défini deux *setters* spécifiques à savoir *setAge()* et *setAnciennete()* qui permettent respectivement de calculer l'âge et l'ancienneté et d'initialiser les champs correspondant. En plus des *setters* classiques, nous avons défini un setter spécifique *augmenteSalaire()* qui permet d'appliquer un taux d'augmentation sur le salaire de l'employé et de mettre à jour la valeur du champs salaire.

S'agissant des qualificateurs, tous les champs de la classe sont déclarés en private. Ce qui signifie que ces champs ne sont accessibles et modifiables qu'en passant par les méthodes de la classe elle-même. Ces méthodes peuvent être de type private, protected ou public[13]. Ici toutes les méthodes sont de type public à l'exception de getAnneeCourante() qui est de type private.

4.13.4 Créer un objet de type Class

Il existe plusieurs façons de créer un objet de type Class. Nous présentons ci-après quelques méthodes de création d'objet de type Class.

4.13.4.1 Créer un objet de type Class à partir d'un nom de classe spécifié en valeur String : usage de la méthode Class.forName()

Pour créer un objet de type Class à partir d'une valeur String, on utilise l'instruction Class.forName(nomClasseString) où nomClasseString est le nom pleinement qualifié de la classe (fully-qualified name). Le nom pleinement qualifié est le nom de la classe accompagné de la spécification du package. Par exemple com.tuto.introspection.Employe est le nom pleinement qualifié de la classe Employe.

L'exemple ci-dessous crée un objet de type Class à partir de la classe Employe dont le nom est spécifié en valeur String.

```
package com.tuto.introspection;
```

[13] Pour plus de détails sur les qualificateurs private, protected et public, voir la section Encapsulation et visibilité des membres de classe.

```
import java.lang.reflect.Type;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException {

        // Valeur String à partir de laquelle, l'objet Class sera créé
        String nomClasseString="com.tuto.introspection.Employe";
        // Création de l'objet Class
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        System.out.println("L'objet Class a été bien créé à partir de la valeur
String "+nomClasseString);
        // Utilisation de l'objet Class chargé
        System.out.println("L'objet de type Class est: "+
ClassEmploye.toString());

    }
}
```

Output :

```
L'objet Class a été bien créé à partir de la valeur String :
com.tuto.introspection.Employe
L'objet de type Class est: class introspection.Employe com.tuto.
```

Dans cet exemple, nous avons créé un objet de type Class nommé ClassEmploye dont le nom pleinement qualifié est le même que celui de la classe d'origine : com.tuto.introspection.Employe.

Rappelons que la classe Employe existe déjà dans le package com.tuto.introspection. Elle est définie dans le code source CS06. A noter que si la classe Employe n'existait pas ou si le package spécifié n'était pas correct, nous allions recevoir une exception de type : ClassNotFoundException. Par exemple, essayons de créer un objet de type Class à partir de la classe Client qui n'existe pas à priori dans le package com.tuto.introspection. Ce code se présente comme suit :

```
package com.tuto.introspection;

import java.lang.reflect.Type;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException {

        // Valeur String à partir de laquelle, l'objet Class sera créé
        String nomClasseString="com.tuto.introspection.Client";
        // Création de l'objet Class
        Class<Type> ClassClient = (Class<Type>) Class.forName(nomClasseString);
        System.out.println("L'objet Class a été bien créé à partir de la valeur
String "+nomClasseString);
        // Utilisation de l'objet Class chargé
        System.out.println("L'objet de type Class est: "+
ClassClient.toString());

    }
}
```

Output :

```
Exception in thread "main" java.lang.ClassNotFoundException:
com.tuto.introspection.Client
at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.j
ava:641)
at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoader
s.java:188)
at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
at java.base/java.lang.Class.forName0(Native Method)
at java.base/java.lang.Class.forName(Class.java:391)
at java.base/java.lang.Class.forName(Class.java:382)
at com.tuto.introspection.Main.main(Main.java:11)
```

4.13.4.2 Créer un objet de type Class à partir d'un objet concret : usage de la méthode getClass()

On peut créer un objet de type Class à partir d'un objet concret, c'est-à-dire un objet obtenu en appelant le constructeur d'une classe concrète avec l'opérateur new. Pour créer un objet de type Class à partir un objet concret, il suffit d'appeler la méthode getClass(). La méthode getClass() est une méthode de la classe Object qui est la classe mère de toutes les classes concrètes Java. L'exemple ci-dessous montre la création d'un objet de type Class à partir d'un objet concret obtenu par instanciation de la classe Employe.

```
package com.tuto.introspection;

import java.lang.reflect.Type;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException {

        // Création de l'objet concret
        Object employe= new Employe( "Karine", "Feminin", 1995, 2020, 2500);
        // Création de l'objet Class
        Class<Type> ClassEmploye=  (Class<Type>)employe.getClass();
        System.out.println("L'objet Class a été bien créé à partir de l'objet
concret employe ");
        // Utilisation de l'objet Class chargé
        System.out.println("L'objet de type Class est: "+
ClassEmploye.toString());

    }
}
```

Output :

```
L'objet Class a été bien créé à partir de l'objet concret employe
L'objet de type Class est: class introspection.Employe com.tuto.
```

Dans cet exemple, nous créons d'abord l'objet employe en instanciant la classe Employe définie dans le code source CS06. Le type déclaré pour l'objet employe est Object au lieu de Employe. Cela ne pose aucun problème, puisque toutes les classes concrètes héritent de la

classe Object. Le fait de déclarer Object comme le type de l'objet employe permet de rendre cet objet plus générique. Ce qui permet de faciliter son usage dans différents contextes.

Pour créer, l'objet de type Class, nous appelons simplement la méthode getClass() sur l'objet employe.

4.13.5 Les méthodes couramment utilisées de la classe Class

La classe Class dispose de plusieurs méthodes qui permettent d'appliquer des opérations de traitement sur les objets de type Class. Ci-dessous quelques-unes de ces méthodes. Pour une liste complète des méthodes applicables sur un objet de type Class, consulter la page suivante : https://docs.oracle.com/javase/10/docs/api/java/lang/Class.html.

4.13.5.1 La méthode newInstance()

La méthode newInstance() permet de créer une nouvelle instance (c'est-à-dire un objet) de la classe représentée par l'objet de type Class. La méthode newInstance() est l'équivalent de l'usage de l'opérateur new pour instancier une classe dans l'approche traditionnelle de création d'objet. L'exemple ci-dessous montre l'utilisation de la méthode newInstance().

```
package com.tuto.introspection;

import java.lang.reflect.Type;
public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode newInstance() pour créer une nouvelle
instance de la classe Employe
        Employe employe =(Employe) ClassEmploye.newInstance();
        System.out.println("Un objet de la classe Employe a été créé avec
succès");

    }
}
```

Output :

```
Un objet de la classe Employe a été créé avec succès
```

La méthode newInstance() est une méthode alternative de création d'objet par rapport à l'opérateur new habituellement utilisé pour instancier les classes. Cependant, il faut noter qu'à la différence de l'opérateur new, la méthode newInstance() n'appelle que le constructeur par défaut de la classe. En effet, l'appel de la méthode newInstance() sur l'objet ClassEmploye équivaut à l'appel du constructeur par défaut de la classe Employe telle que employe = new Employe() . Cela signifie que l'appel de la méthode newInstance() initialise tous les champs de la classe à null ou 0 pour les champs de type numérique. Il faut ensuite appeler les setters de l'objet afin de spécifier les valeurs souhaitées des champs. Par

exemple, pour définir les valeurs des champs de l'objet employe obtenu suite à l'appel de la méthod newInstance(), le code précédemment présenté peut être complété comme suit :

```
package com.tuto.introspection;
import java.lang.reflect.Field;
import java.lang.reflect.Type;
public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode newInstance() pour créer une nouvelle
instance de la classe Employe
        Employe employe =(Employe) ClassEmploye.newInstance();
        System.out.println("Un objet de la classe Employe a été créé avec
succès");
        // Définition des champs de l'objet employe
        employe.setNom("Karine");
        employe.setSexe("Feminin");
        employe.setAnneeNaissance(1995);
        employe.setAnneeEmbauche(2020);
        employe.setSalaire(2500);
        employe.setAge();
        employe.setAnciennete();

    }
}
```

Cet exemple montre qu'après l'appel de la méthode newInstance() sur un objet de type Class pour créer un objet concret de la classe initiale, il est parfois nécessaire d'appeler les setters de la classe pour pouvoir définir les valeurs des champs. Dans cet exemple, nous avons appelé les différents setters définis dans la classe Employe (voir code source CS06 pour les détails concernant la définition de la classe Employe).

4.13.5.2 La méthode getName()

Cette méthode permet de récupérer et de renvoyer le nom de la classe dont l'objet de type Class représente une instance. Ci-dessous un exemple d'usage de la méthode getName()

```
package com.tuto.introspection;
import java.lang.reflect.Type;
public class Main {
    public static void main(String[] args) throws ClassNotFoundException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode getName()
        String fqn=ClassEmploye.getName();
        System.out.println("Le nom complet qualifié est : "+ fqn);

    }
}
```

Output :

```
Le nom complet qualifié est : com.tuto.introspection.Employe
```

Dans cet exemple, nous créons d'abord un objet de type Class à partir d'un nom de classe (ou d'une interface) spécifié sous forme de String. Il s'agit du nom pleinement qualifié de la classe (fully qualified name). Nous avons nommé l'objet de type de Class ClassEmploye.

A la suite de la création de l'objet ClassEmploye, nous avons ensuite appelé la méthode getName() sur cet objet. Et nous affichons le résultat renvoyé dans une variable fqn (fully qualified name).

La variable fqn renvoie la valeur com.tuto.introspection.Employe. Ce qui correspond au nom pleinement qualifié initialement utilisé pour créer l'objet de type de Class. Tout ceci signifie que la méthode getName() renvoie le nom pleinement qualifié d'une classe de type Class. Pour rappel, le nom pleinement qualifié (fully qualified name) d'une classe est le nom de la classe préfixé par le nom du package qui le porte.

4.13.5.3 La méthode getSimpleName()

A la différence de la méthode getName() qui renvoie le nom pleinement qualifié c'est-à-dire le nom de la classe (ou de l'interface) préfixé avec le nom du package, la méthode getSimpleName() renvoie uniquement le nom de la classe. L'exemple ci-dessous montre l'utilisation de la méthode getSimpleName().

```
package com.tuto.introspection;
import java.lang.reflect.Type;
public class Main {
    public static void main(String[] args) throws ClassNotFoundException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode getName()
        String sn=ClassEmploye.getSimpleName();
        System.out.println("Le nom simple de la classe est : "+ sn);

    }
}
```

Output :

```
Le nom simple de la classe est : Employe
```

La méthode getSimpleName() renvoie le nom de la classe ou de l'interface ayant servi à créer l'objet de type Class sans indiquer le package dans lequel il est situé.

4.13.5.4 La méthode getFields():

La méthode getFields() permet de récupérer sous forme d'Array l'ensemble des champs de type public d'une classe de type Class. A noter que si tous les champs de la classe sont de

type private ou protected, aucun champ ne sera renvoyé par la méthode getFields(). L'exemple ci-dessous illustre l'utilisation de la méthode getFields() sur l'objet ClassEmploye.

```
package com.tuto.introspection;
import java.lang.reflect.Field;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
NoSuchFieldException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode getField()
       Field [] fieldsArrays=ClassEmploye.getFields();
       List listChamps = new ArrayList<>();
       for (int i = 0; i < fieldsArrays.length; i++)
           // récupérer les champs et leur type
           listChamps.add(fieldsArrays[i].getType().getSimpleName()+"
"+fieldsArrays[i].getName());
       if (listChamps.size()==0) {
           System.out.println("La classe Employe n'a aucun champ public ");
       } else {
           System.out.println("La liste des champs publics de classe Employe
sont : " + listChamps);
       }

    }
}
```

Output :

```
La classe Employe n'a aucun champ public
```

Dans l'exemple ci-dessus, aucun champ n'est renvoyé car tous les champs de la classe Employe de type private (voir code source CS06).

Mais en changeant les qualificateurs des champs de type public des champs définis dans la classe Employe et en exécutant le code on obtient les résultats suivants :

```
La liste des champs publics de classe Employe sont : [String nom, String sexe,
int anneeNaissance, int anneeEmbauche, int anciennete, int age, double salaire]
```

A noter que les champs renvoyés disposent aussi de leurs propres méthodes qui permettent par exemples de savoir les types de chaque champ. C'est pourquoi nous appelons des méthodes comme getType() pour avoir des informations détaillées sur chaque champ renvoyé dans la liste.

4.13.5.5 La méthode getMethods()

La méthode getMethods() renvoie sous forme d'Array l'ensemble des méthodes de type public dans la classe représentée sous forme d'objet de type Class. L'exemple ci-dessous montre l'utilisation de la méthode getMethods() sur l'objet ClassEmploye obtenue à partir de la classe Employe défini dans le code source CS06.

```
package com.tuto.introspection;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
NoSuchFieldException {
       // Création de l'objet de type Class
        String nomClasseString="com.tuto.introspection.Employe";
        Class<Type> ClassEmploye = (Class<Type>)
Class.forName(nomClasseString);
        // Utilisation de la méthode getField()
       Method[] methodsArrays=ClassEmploye.getMethods();
       List listChamps = new ArrayList<>();
       for (int i = 0; i < methodsArrays.length; i++)
           // récupérer les champs, les paramètres et les valeurs de retour
           listChamps.add(methodsArrays[i].getReturnType().getSimpleName()+"
"+methodsArrays[i].getName());
       if (listChamps.size()==0) {
           System.out.println("La classe Employe n'a aucun champ public ");
       } else {
           System.out.println("La liste des champs publics de classe Employe
sont : " + listChamps);
       }

    }
}
```

Output :

```
La liste des champs publics de classe Employe sont : [void setNom, String
getSexe, void setSexe, String getNom, int getAnneeNaissance, double getSalaire,
int getAnneeEmbauche, void setSalaire, int getAge, int getAnciennete, void
augmentSalaire, void setAge, void setAnneeNaissance, void setAnciennete, void
setAnneeEmbauche, boolean equals, String toString, int hashCode, Class
getClass, void notify, void notifyAll, void wait, void wait, void wait]
```

Dans cet exemple, nous appelons la méthode getMethods() sur l'objet ClassEmploye pour renvoyer l'ensemble de ses méthodes de type public. A noter que les méthodes renvoyées disposent aussi de leurs propres méthodes qui permettent par exemple de savoir les types des valeurs de retour. C'est pourquoi nous appelons des méthodes comme getReturnType() et getSimpleName() pour avoir des informations détaillées sur chaque méthode renvoyée dans la liste.

5 LES EXPRESSIONS LAMBDA

5.1 Généralités sur les expressions lambda

Jusqu'à la version 8, Java est resté strictement un langage de programmation impérative. Mais depuis la version 8, le langage a étendu ses capacités de programmation fonctionnelle[14]. Et cela, grâce à l'apport des expressions lambda (encore appelées fonctions lambda).

Typiquement, une expression lambda est une fonction anonyme, c'est-à-dire une fonction définie sans qu'un nom lui soit associé, par analogie aux classes anonymes. L'origine des expressions lambda remonte des lambda-calculus, premier formalisme à avoir défini et caractérisé les fonctions récursives, proposé dans les années 1930 par Alonzo Church. Les fonctions lambda sont à la base de la programmation fonctionnelle et permettent d'écrire des programmes plus courts et concis comparativement à l'approche traditionnelle basée sur la définition de méthodes explicitement nommées

La particularité des expressions lambda c'est qu'il est possible de les stocker dans des variables, définir des fonctions qui prennent en entrée d'autres fonctions et/ou qui renvoient des fonctions comme valeur de retour.

5.2 Syntaxe générale d'une expression lambda

La syntaxe générale d'une expression lambda se présente comme suit :

```
(parametre1, parametre2,…, parametreN) -> { instructions }
```

Le formalisme d'une expression lambda est construit autour deux parties séparées par le symbole ->. A gauche, on spécifie entre parenthèses l'ensemble des paramètres de la fonction ainsi que leur type. Et à droite on spécifie les instructions qui forment le corps de la fonction.

Les paramètres de la fonction (spécifiés à gauche de l'opérateur ->) sont généralement des variables ou des objets ordinaires Java. Chaque paramètre est déclaré avec son type : primitif ou de type classe.

Les blocs d'instructions (définis à droite de l'opérateur ->) traduisent les opérations que la fonction est censée réaliser. L'exemple ci-dessous montre un exemple simple de fonction lambda :

```
(int x, int y) -> {  return x + y; }
```

[14] La programmation impérative est un paradigme de programmation dans lequel les opérations de traitement sont spécifiées comme une séquence clairement définies d'instructions. Cette approche de programmation est plus dadopté dans les langages comme Java,C, C++. La programmation fonctionnelle, quant à elle, est un paradigme dans lequel toutes les intructions sont exprimées sous forme de fonction, au sens mathématique du terme. Le cœur du programme n'est plus des objets mais plutôt des fonctions. Dans la programmation fonctionnelle, les fonctions sont passées commes des paramètres à d'autres fonctions et les valeurs de retour de certaines fonctions peuvent être des fonctions. Les fonctions sont définies comme des entités auxquelles on peut appliquer les mêmes opérations que n'importe quelle autre entité du langage. Ci-après quelques langages de programmation fonctionnelle: Scala, Lisp, ML, Closure. Etc.

Dans cet exemple, l'expression lambda prend en paramètres deux variable x et y de type int, chacune et renvoie la somme des deux variables. Pour rappel, dans la programmation impérative cette fonction aurait été spécifiée par exemple comme suit :

```
public static int addition(int x, int x) {
        return x + y;
}
```

Mais la seule différence entre cette formalisation et un formalisme sous forme d'expression lambda est qu'une expression lambda est exprimée sans spécifier un nom tandis.

Par ailleurs, notons que tout comme une méthode ordinaire, une fonction lambda peut être définie sans paramètres. Dans ce cas, les parenthèses seront laissées vides. L'exemple ci-dessous illustre une fonction lambda sans paramètres.

```
()->{System.out.println("Bonjour, comment allez vous ?");};
```

Notons aussi que la syntaxe générale des expressions lambda précédemment présentée reste assez standard et peut, en pratique, être déclinée sous différentes variantes selon les situations. L'objectif de ce chapitre est de présenter les définitions et les usages pratiques des fonctions lambda.

5.3 Expression lambda et interface fonctionnelle

La programmation fonctionnelle n'étant pas directement applicable dans un contexte orientée-objet, Java adopte une démarche indirecte pour pouvoir rendre possible la programmation fonctionnelle en se basant sur les éléments du langage déjà existant. Pour ce fait, Java utilise la notion d'interface fonctionnelle.

L'un des premiers usages des expressions lambda dans le langage Java est l'implémentation d'une interface fonctionnelle. Pour rappel, une interface fonctionnelle est une interface ne comportant qu'une seule méthode à implémenter. Comme nous l'avons déjà vu avec les classes anonymes, il est possible d'implémenter une méthode unique et instancier la classe à la volée dans une méthode de l'inner classe sans avoir à donner un nom à l'instance créée. Dans la même logique, lorsque l'on dispose d'une interface ne comportant qu'une seule méthode, on peut implémenter à la volée cette méthode, construire un objet à partir de cette implémentation, passer cet objet à une autre fonction ou méthode dans le reste du programme, et le tout dans une seule séquence d'instructions. C'est sur cette astuce technique que se base le langage Java pour matérialiser les expressions lambda pour pouvoir mettre en œuvre la programmation fonctionnelle. Pour bien illustrer l'usage des expressions lambda pour implémenter une interface fonctionnelle, partons d'une interface telle que spécifiée ci-dessous :

```
package com.tuto.lambda;

public interface Operation {
```

```
    public int addition (int x, int y);
}
```

Java prévoit une annotation spécifique permettant au JVM de reconnaître implicitement une interface fonctionnelle. C'est l'annotation @FunctionalInterface. Même si l'usage de cette annotation n'est pas obligatoire, il est judicieux de l'utiliser autant que possible lors de la définition des interfaces fonctionnelles. Ainsi, l'interface précédemment présentée pouvait aussi être présentée comme suit :

```
package com.tuto.lambda;

@FunctionalInterface
public interface Operation {

    public int addition (int x, int y);

}
```

Supposons que nous souhaitons maintenant utiliser la méthode addition() définie dans cette interface fonctionnelle. Grâce aux expressions lambda, il est possible d'implémenter à la volée cette méthode. L'exemple ci-dessous illustre une manière d'implémenter l'interface et d'utiliser la méthode addition().

```
Operation op = (int x, int y )-> {return x+y;};

int total = op.addition(5, 7);

System.out.println( total);
```

Grâce à l'utilisation des expressions lambda, nous avons pu implémenter à la volée, la seule méthode déclarée dans l'interface Operation tout en créant un objet de type Operation. Ensuite, nous avons pu faire l'appel proprement dite de la méthode addition() qui avait été initialement déclarée dans l'interface. La spécification de l'expression lambda permet directement d'implémenter la méthode abstraite addition() car il n'existe pas une autre méthode abstraite dans l'interface Operation. C'est d'ailleurs la raison pour laquelle l'interface Operation est qualifiée d'interface fonctionnelle, car elle ne contient qu'une seule méthode abstraite. La spécification d'une expression lambda pour instancier un objet de type Operation implémentera implicitement la méthode addition(). On peut remarquer au passage qu'à aucun moment, nous avons utilisé l'opérateur new pour instancier l'objet op. Ce qui marque une grande différence avec l'usage des classes anonymes. Pour ces dernières, l'instanciation est explicitement requise suite à l'implémentation de la méthode abstraite.

C'est sur la base de ces mécanismes décrits ci-dessus que l'usage des fonctions lambda est rendu possible dans le langage Java. Grace à l'interface fonctionnelle, il devient possible de passer en paramètre d'une méthode ou d'une autre fonction (lambda) un objet obtenu suite à la spécification d'une expression lambda. L'exemple ci-dessous définit une classe concrète dans laquelle une des méthodes fait appel à l'objet op obtenu via une fonction lambda telle que précédemment présentée. Cette classe est nommée Calcul et se présente comme suit :

```
package com.tuto.lambda;

public class Calcul{

    private int x;
```

```
    private int y;
    private int resultat;

    // Le constructeur de la classe
    public Calcul(int x, int y) {
        this.x=x;
        this.y=y;
    }
    // Appel de l'objet issu de la fonction lambda
    public void execute(Operation op) {
        this.resultat= op.addition(this.x,  this.y);
    }

    public int getResultat() {
        return resultat;
    }

}
```

La fonction Calcul étant définie, on peut lancer l'exécution des opérations dans une classe Main définie comme suit :

```
package com.tuto.lambda;
public class Main {
        public static void main(String[] args) {
                // Instancier la classe clacul
                Calcul calcul = new Calcul(5, 7);
                // Implémenter la méthode addition  de l'interface Operation et
instancier l'objet op via une expression lambda
                // Et Executer l'addition
                calcul.execute((int x, int y )-> {return x+y;});
                int resultat=calcul.getResultat();
                //Afficher le résultat après l'execution
                System.out.println("Le résultat est: "+resultat);
        }
}
```

Output :

```
Le résultat est: 12
```

Dans cet exemple, nous instancions un objet de la classe concrète Calcul. Cet objet nommé calcul a pour attributs x=5 et y =7. Ensuite, nous appelons la méthode execute() de cet objet. Cette méthode a pour signature un objet de type op, c'est-à-dire un objet obtenu à partir de l'expression lambda qui implémente la méthode abstraite addition() de l'interface fonctionnelle Operation. En appelant la méthode excute(), on appelle directement une implémentation de la méthode addition() fait la somme des deux attributs x et y pour assigner à l'attribut resultat (soit 12).

Enfin, nous appelons la méthode getResultat() qui renvoie la valeur de l'attribut resultat. La valeur est affichée en appelant la méthode println(). Tel est l'exemple typique de l'utilisation d'une expression lambda dans un programme Java.

5.4 De la classe anonyme à l'expression lambda

Dans de nombreuses situations, l'utilisation des expressions lambda est une alternative à l'usage des classes anonymes en particulier lorsque ces classes anonymes implémentent des interfaces fonctionnelles. Dans cette section, nous montrons comment retranscrire une classe anonyme en une expression lambda dans un programme.

Soit une interface fonctionnelle et une classe concrète nommées respectivement Greeting et EnvoiSalutation et définies comme suit :

```
package com.tuto.lambda;
@FunctionalInterface
public interface Greeting {
        public void sendGreeting();
}
```

```
package com.tuto.lambda;
public class EnvoiSalutation {
        public void envoi() {
                // Définition classe anonyme à partir de l'interface Greeting
                Greeting anonymeSalut = new Greeting() {
                        @Override
                        public void sendGreeting() {
                                System.out.println("Bonjour, comment allez vous
?");
                        }
                };
                anonymeSalut.sendGreeting();
        }
}
```

L'interface Greeting prévoit une méthode sendGreeting() dont le but sera d'envoyer un mot de salutation. Cette méthode pourra être implémentée par n'importe quelle classe du programme.

Justement la classe EnvoiSalutation est une classe concrète qui a une méthode nommée envoi(). Le mode de fonctionnement de cette méthode envoi() est la suivante. Elle instancie à la volée une classe anonyme qui est l'implémentation de l'interface Greeting et de sa méthode sendGreeting(). L'objet obtenu par l'instanciation de la classe anonyme est directement utilisé dans la méthode envoi() en appelant la méthode sendGreeting() précédemment implémentée à la volée. Tel est le principe d'utilisation des classes anonymes dans un programme.

Cependant, on constate que cette procédure génère beaucoup plus de lignes de code pour une action aussi simple. En utilisant une expression lambda, il est possible de rendre ce code beaucoup plus concis. L'exemple ci-dessous montre la réécriture de la classe EnvoiSalutation en utilisant une expression lambda à la place d'une classe anonyme.

```
package com.tuto.lambda;
public class EnvoiSalutation {
        public void envoi() {
                // Définition d'une expression lambda
```

```
                Greeting lambdaSalut = () ->{System.out.println("Bonjour,
comment allez vous ?");};
                lambdaSalut.sendGreeting();
        }
}
```

Dans cette nouvelle spécification, seulement deux lignes de code sont nécessaires pour définir la méthode envoi(). D'abord la spécification de l'expression lambda. L'interface Greeting étant une interface fonctionnelle, la création d'un objet de type Greeting à travers une expression lambda implémente automatiquement la méthode abstraite prévue dans la déclaration de l'interface. En l'occurrence, il s'agit ici de la méthode sendGreeting(). Comme on peut le constater dans le code ci-dessus, dans l'usage d'une expression lambda pour implémenter une interface fonctionnelle, il n'est pas nécessaire de spécifier le nom de la méthode implémentée. Celle-ci devient une fonction anonyme dont les paramètres (s'ils existent) sont les paramètres de la méthode abstraite initiale et les instructions, celles spécifiées pour pouvoir implémenter la méthode abstraite.

Au final, comme on peut le remarquer, l'utilisation de l'expression lambda à la place de la classe anonyme permet de rendre le code beaucoup plus concis en réduisant le nombre de lignes de code.

Attention toutefois, l'utilisation des expressions lambda, même si elle permet de rendre les codes beaucoup plus concis n'est pas toujours un gage de clarté. Par exemple, sans une documentation claire, il n'est pas possible de savoir quelle interface fonctionnelle a été utilisée et quelle méthode a été implémentée. Toutefois, il reste tout à fait possible d'utiliser les expressions lambda sans qu'elles soient adossées à une interface fonctionnelle. Dans ce cas, il s'agit des expressions lambda libres conçues pour répondre aux besoins de l'utilisateur.

5.5 Quelques cas concrets d'utilisation des expressions lambda

Dans cette section, nous allons présenter quelques cas d'utilisation des expressions lambda en particulier l'usage dans des traitements de collections, les opérations map, filter, etc…

5.5.1 Trier les éléments d'une collection. Ex : ArrayList

L'interface fonctionnelle Comparator<T> fournit une méthode abstraite compare() qui peut être implémentée par une expression lambda pour comparer deux objets Java. Rappelons que la structure du langage Java est conçue de telle sorte que la comparaison de deux objets x et y Java renvoie l'une des trois valeurs possibles : -1 si x<y, 0 si x==y et 1 si x>y. Ce critère peut ainsi être utilisé pour élaborer une méthode de comparaison et implémenter par la même occasion la méthode compare() de l'interface Comparator <T>. La méthode étant implémentée, on peut l'utiliser sur n'importe quelle collection dont les éléments sont comparables pour trier les éléments. L'exemple ci-dessous, utilise une

expression lambda pour implémenter la méthode compare() de l'interface Comparator<T> pour réaliser un tri croissant et un tri décroissant sur les éléments d'un ArrayList.

```
package com.tuto.lambda;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class Main {
        public static void main(String[] args) {
                // Initialise un ArrayList
                List numeros= new ArrayList<Integer>(Arrays.asList(24, 17, 85,
44, 52, 12, 35, 85, 3,54));
                System.out.println("Ordre avant tri: "+numeros.toString()); //
Affiche l'arrayList avant le tri
                // Tri par ordre croissant
                Collections.sort(numeros,(Integer e1, Integer e2)->{
                                if (e2 < e1)
                                        return 1;
                                else if (e2.equals(e1))
                                        return 0;
                                else // e2 > e1
                                        return -1;
                        });
                System.out.println("Ordre après tri croissant:
"+numeros.toString()); // Affiche l'arrayList après le tri croissant
                // Tri par ordre décroissant
                Collections.sort(numeros,(Integer e1, Integer e2)->{
                        if (e2 < e1)
                                return -1;
                        else if (e2.equals(e1))
                                return 0;
                        else // e2 > e1
                                return 1;
                });
                System.out.println("Ordre après tri décroissant:
"+numeros.toString()); // Affiche l'arrayList après le tri décroissant

        }
}
```

Output :

```
Ordre avant tri: [24, 17, 85, 44, 52, 12, 35, 85, 3, 54]
Ordre après tri croissant: [3, 12, 17, 24, 35, 44, 52, 54, 85, 85]
Ordre après tri décroissant: [85, 85, 54, 52, 44, 35, 24, 17, 12, 3]
```

Dans l'exemple ci-dessus, nous commençons par créer une collection de type ArrayList constitué uniquement des valeurs de type Integer. Cette collection est nommée numeros. Comme on peut le constater les éléments de numeros ne sont pas triés. L'objectif de l'exemple c'est justement de trier les éléments, d'abord dans un ordre croissant, ensuite dans un ordre décroissant.

S'agissant du tri croissant des éléments, nous appelons d'abord la méthode sort() de la classe statique Collections. La méthode sort() permet de spécifier non seulement la

collection à trier, mais aussi la fonction de comparaison qui permet de comparer les éléments pour pouvoir réaliser le tri.

Pour réaliser le tri croissant sur les éléments de l'ArrayList numeros, nous spécifions une expression lambda dont les paramètres d'entrée sont deux entiers e1 et e2 de types Integer. Le bloc d'instructions est construit suivant une structure de contrôle IF…ELSE à partir de trois conditions renvoyant trois entiers différents : 1 (si e1<e2), 0 (si e1==e2) et 1 (si e1>e2). A noter que pour la comparaison e1==e2, nous avons utilisé la méthode equal() au lieu de l'opérateur == car les valeurs à comparer sont de type Integer (type classe) au lieu de int (type primitif). En effet, l'opérateur == appliqué à des objets de type classe fait une comparaison par référence (c'est-à-dire vérifie s'il s'agit des instances d'une même classe) et non de même valeur. Alors que deux instances différentes d'une même classe contenant la même valeur doivent être considérées comme égales, par comparaison. C'est la méthode equals() qui permet de faire de telle comparaison. En définitive, la comparaison des valeurs des paramètres d'entrée et la valeur de retour de la structure de contrôle IF…ELSE représente une expression lambda implémentant la méthode compare() de l'interface Comparator<T> pour réaliser un tri croissant.

Et pour réaliser un tri décroissant sur les valeurs de la collection ArrayList numeros, nous gardons la même expression lambda du tri croissant en changeant simplement le signe des valeurs retour (revoir le code source ci-dessus).

A titre d'information, rappelons que dans une approche Orientée-Objet standard, pour réaliser le tri de la collection ArrayList numeros précédemment présentée, on pouvait instancier une classe anonyme à partir de l'interface Comparator<T> et implémenter à la volée la méthode compare() et faire les tris sur les éléments de l'ArrayList d'entrée. Ce code aurait été présenté comme suit :

```
package com.tuto.lambda;
import java.util.*;

public class Main {
        public static void main(String[] args) {
                // Initialise un ArrayList
                List numeros= new ArrayList<Integer>(Arrays.asList(24, 17, 85,
44, 52, 12, 35, 85, 3,54));
                System.out.println("Ordre avant tri: "+numeros.toString()); //
Affiche l'arrayList avant le tri
                // Procédure de tri par ordre croissant
                // Instancier une classe anonyme à partir de l'interface
Comparator
                Comparator comparatorCroissant = new Comparator<Integer>() {
                        @Override
                        public int compare( Integer e1, Integer e2) {
                                if (e2 < e1)
                                        return 1;
                                else if (e2.equals(e1))
                                        return 0;
                                else // e2 > e1
                                        return -1;
                        };
                };
```

```
                Collections.sort(numeros,comparatorCroissant);
                System.out.println("Ordre après tri croissant:
"+numeros.toString()); // Affiche l'arrayList après le tri croissant
                // Procédure de tri par ordre décroissant
                // Instancier une classe anonyme à partir de l'interface
Comparator
                Comparator comparatorDecroissant = new Comparator<Integer>() {
                        @Override
                        public int compare( Integer e1, Integer e2) {
                                if (e2 < e1)
                                        return -1;
                                else if (e2.equals(e1))
                                        return 0;
                                else // e2 > e1
                                        return 1;
                        };
                };
                Collections.sort(numeros,comparatorDecroissant);
                System.out.println("Ordre après tri décroissant:
"+numeros.toString()); // Affiche l'arrayList après le tri décroissant

        }
}
```

Output :

```
Ordre avant tri: [24, 17, 85, 44, 52, 12, 35, 85, 3, 54]
Ordre après tri croissant: [3, 12, 17, 24, 35, 44, 52, 54, 85, 85]
Ordre après tri décroissant: [85, 85, 54, 52, 44, 35, 24, 17, 12, 3]
```

5.5.2 Réaliser une opération map() sur une collection : Ex : ArrayList

Dans cette sous-section, nous montrons l'utilisation des expressions lambda pour réaliser des opérations de type map() sur une collection Java.

Rappelons qu'une opération map() est un traitement itératif qui applique la même transformation sur chacun des éléments pris en entrée. Généralement les éléments sont reçus sous forme de collections Java (List, Set, Map,Queue, etc..)[15]. Dans une opération map() pour chaque élément de la collection en entrée correspond un élément de la collection de sortie. Le code ci-dessous présente quelques opérations de map() sur des collections de type ArrayList.

```
package com.tuto.lambda;
import java.util.*;
import java.util.stream.Collectors;

public class Main {
        public static void main(String[] args) {
                // Initialiser un ArrayList dont les éléments sont de type
Integer
                List numeros= new ArrayList<Integer>(Arrays.asList(24, 17, 85,
```

[15] Nous reviendrons en détail les collections Java plus tard dans un chapitre dédié.

```
44, 52, 12, 35, 85, 3,54));
                System.out.println("numeros avant map: "+numeros.toString());
// Affiche l'arrayList avant map
                // Operation: Mutiplier chaque élément par 2
                List numerosDouble= (List) numeros.stream().map((e)->{return
(int)e*2;}).collect(Collectors.toList());;
                 System.out.println("numeros après map:
"+numerosDouble.toString()); // Affiche l'arrayList après map
                //Initaliser un ArrayList des Prénom en minuscule
                List noms =new ArrayList<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));
                System.out.println("noms avant map: "+noms.toString()); //
Affiche l'arrayList avant map
                // Operation: Transformer chaque élément en Majuscule
                List nomsMajuscule= (List) noms.stream().map((e)->{return
((String)e).toUpperCase();}).collect(Collectors.toList());;
                System.out.println("noms après map:
"+nomsMajuscule.toString()); // Affiche l'arrayList après map

        }
}
```

Output :

```
numeros avant map: [24, 17, 85, 44, 52, 12, 35, 85, 3, 54]
numeros après map: [48, 34, 170, 88, 104, 24, 70, 170, 6, 108]
noms avant map: [Laurie, Vincent, Ahmed, Vamouss]
noms après map: [LAURIE, VINCENT, AHMED, VAMOUSS]
```

Dans cet exemple, nous réalisons deux opérations de map(). La première porte sur un Arraylist dont les éléments sont de type Integer. La deuxième porte sur un ArrayList dont les éléments sont de type String. Dans la première opération, nous multiplions chaque élément par 2. Et dans la deuxième opération map(), nous transformons chaque élément en majuscule. Ici, il s'agit des opérations simples. Bien entendu, les opérations map() peuvent être complexes autant que possible. Et elles pourront toujours être spécifiées sous forme d'expressions lambda.

Quelques remarques importantes restent à faire concernant l'utilisation des fonctions map() sur les collections. D'abord, comme on peut le constater, pour pouvoir appeler la méthode map() sur une collection, nous appelons d'abord la méthode stream() sur l'objet collection. La méthode stream() est une méthode de l'interface Collection permettant de réaliser des opérations séquentielles sur une collection. Elle est donc applicable à tout objet représentant une collection. Après la création de l'objet stream en appelant la méthode stream(), on peut maintenant appeler la méthode map() pour spécifier l'expression lambda correspondant à la transformation que nous souhaitons réaliser. Dans les exemples ci-dessus, la transformation étant effectuée sur chaque élément individuelle, l'expression lambda est donc spécifiée avec un seul paramètre e dont il n'est pas nécessaire d'indiquer son type ou d'ailleurs même l'entourer des parenthèses.

La deuxième remarque concernant l'usage de la méthode map() sur une collection est l'appel de la méthode collect() avec comme argument Collectors.toList(). Cette instruction

permet de ramener le stream sous forme de List. Par ailleurs, on applique un opérateur de cast sur la liste collectée et la convertir en objet de type List.

5.5.3 Réaliser une opération filter() sur une collection. Ex : ArrayList

L'exemple ci-dessous montre l'usage d'une expression lambda pour réaliser une opération filter() sur une collection. Pour rappel, un filter() sur une collection est une opération de transformation à l'issue de laquelle on ne garde que les éléments répondant à une condition préalablement définie. Dans l'exemple ci-dessous nous utilisons deux cas. Le premier porte sur une collection dont les éléments sont de type Integer. Le deuxième cas porte sur une collection dont les éléments sont de type String. Pour le premier cas, nous filtrons et gardons les éléments dont la valeur est un nombre pair (modulo 2==0). Dans le deuxième cas, nous filtrons les éléments dont la valeur contient la lettre « V ». Voir exemple ci-dessous.

```
package com.tuto.lambda;
import java.util.*;
import java.util.stream.Collectors;

public class Main {
        public static void main(String[] args) {
                // Initialiser un ArrayList dont les éléments sont de type
Integer
                List numeros= new ArrayList<Integer>(Arrays.asList(24, 17, 85,
44, 52, 12, 35, 85, 3,54));
                System.out.println("numeros avant filter:
"+numeros.toString()); // Affiche l'arrayList avant map
                // Operation: Mutiplier chaque élément par 2
                List numerosPairs= (List) numeros.stream().filter((e)->{return
(int)e%2==0;}).collect(Collectors.toList());;
                 System.out.println("numeros après filter:
"+numerosDouble.toString()); // Affiche l'arrayList après map
                //Initaliser un ArrayList des Prénom en minuscule
                List noms =new ArrayList<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));
                System.out.println("noms avant filter: "+noms.toString()); //
Affiche l'arrayList avant map
                // Operation: Transformer chaque élément en Majuscule
                List nomsMajuscule= (List) noms.stream().filter((e)->{return
((String)e).toUpperCase().contains("V".toUpperCase());}).collect(Collectors.toL
ist());;
                System.out.println("noms après filter:
"+nomsMajuscule.toString()); // Affiche l'arrayList après map

        }
}
```

Output :

```
numeros avant filter: [24, 17, 85, 44, 52, 12, 35, 85, 3, 54]
numeros après filter: [24, 44, 52, 12, 54]
noms avant filter: [Laurie, Vincent, Ahmed, Vamouss]
noms après filter: [Vincent, Vamouss]
```

Contrairement à la méthode map() où l'expression lambda peut renvoyer n'importe quelle valeur, dans une opération filter(), l'expression lambda renvoie toujours une valeur booléenne : true ou false. Et un élément de la collection d'entrée est retenue uniquement lorsque la valeur renvoyée par l'expression lambda est true. Dans l'exemple ci-dessous, on constate que les collections générées par les filter ont moins d'éléments que les collections initiales.

6 LES COLLECTIONS

Les collections de Java regroupent un ensemble d'interfaces et de classes prévues pour la manipulation et le traitement des données organisées sous formes de séquences.

Dans les précédents chapitres, nous avons déjà vu les Arrays et les Arrays multidimensionnels qui sont aussi des structures organisées sous formes de séquence de valeurs. Toutefois, ces structures manquent de flexibilité à certains égards. Par exemple, lorsque la dimension d'un Array est définie, elle n'est plus modifiable. Les collections Java apportent plus de flexibilité en offrant un ensemble d'algorithmes permettant de restructurer à souhait les séquences de données. Ce présent chapitre est consacré à l'étude des collections Java.

6.1 Les principales classes de collections Java

Le framework Collection est constitué d'une interface principale Collection <E> qui est étendue par un certain nombre de sous-interfaces correspondant chacune à un type d'organisation spécifique des séquences de données. Il s'agit notamment des interfaces List, Set, SortedSet, NavigableSet, Queue et Deque. Ces sous-interfaces sont implémentées à leur tour des classes représentant des données réelles. Le tableau 10 ci-dessous fournit les principales classes d'implémentation des interfaces mentionnées.

Tableau 10: Principales classes de collection et les interfaces correspondants

Classe d'implémentation	Interfaces	Exemple de séquence
ArrayList	List	[1, 2, 3, 1, 20, 3]
LinkedList	List, Queue, Deque	[1, 2, 3, 1, 20, 3]
Vector	List	[1, 2, 3, 1, 20, 3]
HashSet	Set	[1, 3,2, 20]
TreeSet	Set, SortedSet, NavigableSet	[1, 2, 3,20]

HashMap	Map<K,V>	{
		<"Victor",52>,
		<"James",24>,
		<"Valerie",17>,
		<"Ivan",35>,
		<"Jhon",44>
		}

TreeMap	Map<K,V>,
	NavigableMap<K,V>,
	SortedMap<K,V>
		{
		<"Ivan",35>,
		<"James",24>,
		<"Jhon",44>,
		<"Valerie",17>,
		<"Victor",52>
		}

PriorityQueue	Queue	[1, 2, 3, 1, 20, 3]
ArrayDeque	Deque	[1, 2, 3, 1, 20, 3]

L'un des avantages des classes de collections est qu'elles offrent un ensemble d'algorithmes permettant de réaliser des traitements itératifs sur les éléments de la séquence de données mais aussi plusieurs autres opérations de traitement comme des tris, des recherches de minimum et de maximum, etc.

6.2 Les types des éléments d'une collection

A noter que tous les éléments d'une même classe de collection doivent être de même type E. Par exemple une collection ArrayList<String> ne peut contenir que des éléments de type String. A noter également que le type E des éléments d'une collection ne peut pas être un type primitif ; il doit nécessairement être d'un type non primitif (String, Integer, Double, Object, etc.). Ce qui constitue une différence fondamentale avec les séquences de données comme les Arrays qui, eux, supportent les types primitifs.

Le tableau 11 ci-dessous fournit les types des éléments correspondant au types primitifs dans le cadre d'une collection.

Tableau 11: Correspondance entre les types primitifs et les types de collection

Type primitif	Type de collection	Description
byte	Byte	Octet
short	Short	Entier court
int	Integer	Entier
long	Long	Entier long
float	Float	Décimal flottant à simple précision
double	Double	Décimal flottant à double précision
char	Character	Caractère
boolean	Boolean	Booléen

6.3 Etude de la collection ArrayList

Un ArrayList est une structure permettant de présenter les données sous forme d'un tableau dynamique. A la différence de l'Array standard, l'ArrayList offre à l'utilisateur un contrôle total sur la taille du tableau qui peut être redimensionnable à souhait. Pour une documentation complète sur la collection ArrayList, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html

6.3.1 Créer un ArrayList

On peut créer un ArrayList en procédant de deux façons : soit déclarer un ArrayList vide et ajouter les éléments dans un second temps, soit définir l'ArrayList en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un ArrayList.

6.3.1.1 Créer un ArrayList vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        // Initialise un ArrayList vide avec éléments de type Integer
        List numero= new ArrayList<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
l'arrayList


    }
 }
```

Dans cet exemple, nous initialisons un ArrayList vide nommé numero dont les éléments sont prévus pour être de type Integer[16]. Remarquons dans cette déclaration que le type réel de l'objet numero est bien ArrayList<Integer> mais sont type référence est List qui correspond en fait l'interface implémentée par la classe ArrayList. Cela reflète le principe de polymorphisme que nous avons déjà étudié dans les chapitres précédents. A noter aussi qu'on pouvait déclarer la variable numero sous son type réel en remplaçant List par ArrayList<Interger>.

L'ArrayList numero étant initialisée à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés à liste. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

En exécutant le code ci-dessus, on obtient

[16] Rappelons que le type Integer est le type enveloppe (wrapper) pour le type primitif int.

```
La taille initiale est :0
La taille finale est :5
Les éléments sont: [24, 17, 85, 44, 52]
```

6.3.1.2 Créer un ArrayList à partir d'une séquence de valeurs

On peut aussi créer un ArrayList directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous illustre ce mode de création.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        // Initialise un ArrayList à partir d'une séquence initiale de données
        List numero= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
l'arrayList

    }
 }
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument de l'ArrayList. Néanmoins, une petite transformation est nécessaire pour que la séquence soit reconnue comme une liste de valeur. C'est l'utilisation de l'instruction Arrays.asList() qui permet de convertir un Array en une séquence de valeurs de type List. Cette petite conversion nécessite l'import de la librairie Arrays.

En exécutant le code, on obtient le résultat suivant :

```
La taille est :5
Les élements sont: [24, 17, 85, 44, 52]
```

6.3.1.3 Les types des éléments d'un ArrayList

Tous les éléments d'un ArrayList doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un ArrayList dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Arraylist dont les éléments sont String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
l'arrayList
    }
 }
```

6.3.2 Itérateur d'ArrayList : usage de la méthode iterator()

Toutes les collections Java disposent d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection afin de réaliser une opération de traitement. L'objet Iterator est obtenu en appliquant la méthode iterator() sur l'objet représentant la classe de collection. Cette sous-section montre l'utilisation de la méthode iterator() pour parcourir les éléments d'un objet ArrayList.

Soit un ArrayList nommé aL1 défini comme suit.

```
List aL1= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cet ArrayList et renvoyer un nouvel ArrayList nommé aL2 dont chaque élément est égal au double de l'élément initial de l'ArrayList aL1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément de l'ArrayList aL1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Arraylist initial
        List aL1= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("Les élements de aL1 sont: "+aL1.toString());
        // Déclaration de l'ArrayList double
        List aL2 = new ArrayList<Integer>();

        Iterator iter=aL1.iterator(); //Création de l'itérator sur aL1

       //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            aL2.add(new_elem);
        }
        System.out.println("Les élements de aL2 sont: "+aL2.toString());
```

```
    }

 }
```

Output

```
Les élements de aL1 sont: [24, 17, 85, 44, 52]
Les élements de aL2 sont: [48, 34, 170, 88, 104]
```

Cet exemple appelle un certain nombre de commentaires.

D'abord, puisque l'ArrayList aL2 est censé recueillir les doubles des valeurs de aL1, nous positionnons d'abord l'ArrayList aL2 en l'initialisant à vide.

Dans un deuxième temps, nous créons l'iterator iter sur l'objet aL1 en appelant la méthode iterator().

Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément de l'ArrayList aL1. Cette boucle est réalisée en combinant la structure de contrôle while() en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui, chaque fois qu'il est appelé, se déplace d'un pas et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments d'un itérateur jusqu'au dernier. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur.

Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter l'ArrayList de sortie aL2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de aL1 par 2. L'ensemble des opérations de récupération des éléments de aL1 et d'alimentation de aL2 a été réalisé dans la boucle suivante :

```
Iterator iter=aL1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    aL2.add(new_elem);
}
```

Cette boucle est un cas typique de la récupération des éléments d'un ArrayList. Mais comme nous allons le voir plus tard, la même structure de boucle est applicable à toutes les autres classes de collections.

6.3.3 Opérations courantes sur un ArrayList

En plus de la méthode iterator(), l'objet ArrayList fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs.

Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection ArrayList, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html

6.3.3.1 Ajouter un élément à un ArrayList : la méthode add()

La méthode add() permet d'ajouter un élément à un ArrayList. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Ajoute un élément en fin de liste
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un élément à une position i donnée
        noms.add(2,"Valentin"); // Insère à l'indice 2 (troisième position)
        System.out.println("noms: "+noms.toString());
    }

 }
```

Output

```
noms: [Laurie, Vincent, Ahmed, Vamouss, Julien]
noms: [Laurie, Vincent, Valentin, Ahmed, Vamouss, Julien]
```

Par défaut, l'appel de la méthode add() ajoute l'élément en fin de liste. C'est le cas par exemple de l'instruction noms.add("Julien"). Mais avec la méthode add(), il est également possible de spécifier l'indice de position auquel on souhaite insérer un élément dans la liste. C'est le cas de l'instruction noms.add(2,"Valentin") qui insère l'élément à l'indice 2 (position 3 de la liste). A noter que la première position commence toujours par l'indice 0. Et l'ajout d'un élément autre qu'en fin de liste décale tous les éléments à droite d'une position.

6.3.3.2 Ajouter plusieurs éléments à un ArrayList : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un ArrayList, la méthode addAll() permet d'ajouter plusieurs éléments à un ArrayList en un seule fois. Par défaut, ces éléments sont ajoutés en fin de liste. Mais ils peuvent aussi être ajoutés en

commençant à une position donné dans l'ArrayList. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un ArrayList.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de Integer
        List nums=new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
        // Ajoute les éléments à une position i donnée
        nums.addAll(2,Arrays.asList(14, 18, 20)); // Insère à l'indice 2
(troisième position)
        System.out.println("nums: "+nums.toString());
    }

 }
```

Ouput :

```
nums: [24, 17, 85, 44, 52, 63, 45, 10, 100, 91]
nums: [24, 17, 14, 18, 20, 85, 44, 52, 63, 45, 10, 100, 91]
```

6.3.3.3 Vérifier si un ArrayList contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un ArrayList contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Vérifier si l'arrayList contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("Liste contient Ahmed: "+a);

        // Vérifier si l'arrayList contient "Adams"
        boolean b= noms.contains("Adams");
```

```
        System.out.println("Liste contient Adams: "+b);

    }

}
```

Output

```
Liste contient Ahmed: true
Liste contient Adams: false
```

Dans l'exemple ci-dessous, la liste contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la liste ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.3.3.4 Récupérer un élément donné dans une ArrayList : la méthode get()

La méthode get() permet de récupérer et de renvoyer un élément d'un ArrayList en spécifiant sa position dans la liste. L'exemple ci-dessous illustre l'utilisation de la méthode get()

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Recupérer l'élément à l'indice 2 ( position 3)
        noms.get(0); // Renvoie Laurie
        // Recupérer l'élément à l'indice 2 ( position 3)
        noms.get(2); // Renvoie Ahmed

    }
}
```

6.3.3.5 Renvoyer l'indice d'un élément donné d'un ArrayList : la méthode indexOf()

Pour retrouver l'indice d'un élément d'un ArrayList, on utilise la méthode indexOf(). L'exemple ci-dessous montre l'utilisation de la méthode indexOf().

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        noms.indexOf("Vincent"); // Renvoie 1
        noms.indexOf("Vamouss"); // Renvoie 3

    }
}
```

6.3.3.6 Supprimer un élément spécifique d'un ArrayList : la méthode remove()

La méthode remove() permet de supprimer un élément d'un ArrayList. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de String
        List noms =new ArrayList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        noms.remove("Ahmed");
        System.out.println("La liste finale noms :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Laurie, Vincent, Ahmed, Vamouss]
La liste finale noms :[Laurie, Vincent, Vamouss]
```

6.3.3.7 Supprimer un ensemble de valeurs d'un ArrayList : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'un ArrayList. L'exemple ci-dessous montre l'utilisation de la méthode.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de Integer
        List nums= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :[17, 85, 44, 52, 58]
```

6.3.3.8 Modifier la valeur située à une position donnée : la méthode set()

La méthode set() permet de modifier une valeur située à une position donnée dans un ArrayList. L'exemple ci-dessous fournit une illustration.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de Integer
        List nums= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Change la valeur à l'indice 2 à 55
        nums.set(1,55);
        //Change la valeur à l'indice 5 à 100
        nums.set(5,100);
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :   [24, 55, 85, 44, 52, 100, 26, 58]
```

6.3.3.9 Déterminer le nombre d'éléments d'un ArrayList : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un ArrayList.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de Integer
        List nums= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.3.3.10 Convertir un ArrayList en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet ArrayList en un objet de type Array. Cette conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Object, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Arraylist de Integer
        List nums= new ArrayList<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le type ArrayList est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        for (int i=0;i<=nums.size()-1;i++ ){
            nums_array[i]=(Integer) nums.get(i);
        }
         System.out.println("Le type Array est :
"+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type ArrayList est : [24, 17, 85, 44, 52, 20, 26, 58]
Le type Array est : [24, 17, 85, 44, 52, 20, 26, 58]
```

6.4 Etude de la collection LinkedList

Le LinkedList est une collection qui a beaucoup de similarités avec la collection ArrayList. Ces deux collections implémentent en commun l'interface List. Et la plupart des méthodes utilisable sur un ArrayList sont également utilisables sur un LinkedList.

Cependant, la collection LinkedList présente de nombreuses différences avec l'ArrayList. D'abord, la classe LinkedList implémente deux interfaces supplémentaires par rapport au ArrayList que sont Queue et Deque. Ensuite, alors que l'ArrayList présente les données sous forme d'Array constitué d'une séquence d'éléments contigus, le LinkedList représente les éléments sous formes d'objets non contigus stockés chacun dans un container dédié. Chaque objet est constitué d'une donnée et une adresse pour le retrouver. Dans un linkedList, les éléments sont appelés des nœuds. A la différence d'un ArrayList où on peut directement accéder directement à un élément, dans un LinkedList pour accéder à un élément (un nœud), il faut d'abord passer par la tête de la liste et parcourir le chemin jusqu'à atteindre l'élément souhaité.

Cette section a pour but de présenter les principales caractéristiques de la collection LinkedList. Compte tenu de la très grande similarité entre le linkedList et l'ArrayList, nous reprenons les mêmes exemples et les mêmes commentaires de résultats comme ceux présentés dans la section consacrée à l'ArrayList. Pour une documentation complète sur la collection LinkedList, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html

6.4.1 Créer un LinkedList

Tout comme un ArrayList, on peut créer un LinkedList en procédant de deux façons : soit déclarer un LinkedList vide et ajouter ensuite les éléments, soit définir le LinkedList en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un LinkedList.

6.4.1.1 Créer un LinkedList vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
```

```
        // Initialise un LinkedLis vide avec éléments de type Integer
        List numero= new LinkedList<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
le LinkedList


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous initialisons un LinkedList vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons dans cette déclaration que le type réel de l'objet numero est bien LinkedList<Integer> mais sont type référence est List qui correspond à l'une des interfaces implémentées par la classe LinkedList.

Le LinkedList numero étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

En exécutant le code ci-dessus, on obtient

```
La taille initiale est :0
La taille finale est :5
Les éléments sont: [24, 17, 85, 44, 52]
```

6.4.1.2 Créer un LinkedList à partir d'une séquence de valeurs

On peut aussi créer un LinkedList directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création du LinkedList à partir une séquence de valeurs.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;
```

```
public class Main {
    public static void main(String[] args) {

        // Initialise un LinkedList à partir d'une séquence initiale de données
        List numero= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le LinkedList

    }
 }
```

Output

```
La taille est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du LinkedList. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.4.1.3 Les types des éléments d'un LinkedList

Tous les éléments d'un LinkedList doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un LinkedList dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // LinkedList dont les éléments sont String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le LinkedList
    }
 }
```

Output

```
Les élements sont: [Laurie, Vincent, Ahmed, Vamouss, Julien]
```

6.4.2 Itérateur d'un LinkedList : usage de la méthode iterator()

Tout comme l'ArrayList, le LinkedList dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection afin de réaliser une opération de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un LinkedList.

Soit un LinkedList nommé linked1 défini comme suit.

```
List linked1= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cet LinkedList et renvoyer un nouvel LinkedList nommé linked2 dont chaque élément est égal au double de l'élément initial du LinkedList linked1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du LinkedList linked1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        // LinkedList initial
        List linked1= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("Les éléments de linked1 sont:
"+linked1.toString());
        // Déclaration du LinkedList double
        List linked2 = new LinkedList<Integer>();

        Iterator iter=linked1.iterator(); //Création de l'itérator sur linked1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            linked2.add(new_elem);
        }
        System.out.println("Les éléments de linked2 sont:
"+linked2.toString());
    }
}
```

Output

```
Les éléments de linked1 sont: [24, 17, 85, 44, 52]
Les éléments de linked2 sont: [48, 34, 170, 88, 104]
```

Dans cet exemple, le LinkedList linked2 est d'abord initialisé à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet linked1 en appelant la

méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du LinkedList linked1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter le LinkedList de sortie linked2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de linked1 par 2. L'ensemble des opérations de récupération des éléments de linked1 et d'alimentation de linked2 a été réalisé dans la boucle suivante :

```
Iterator iter=linked1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    linked2.add(new_elem);
}
```

6.4.3 Opérations courantes sur un LinkedList

En plus de la méthode iterator(), l'objet LinkedList fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection LinkedList, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html

6.4.3.1 Ajouter un élément à un LinkedList : la méthode add()

La méthode add() permet d'ajouter un élément à un LinkedList. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Ajoute un élément en fin de liste
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un élément à une position i donnée
```

```
        noms.add(2,"Valentin"); // Insère à l'indice 2 (troisième position)
        System.out.println("noms: "+noms.toString());
    }

 }
```

Output

```
noms: [Laurie, Vincent, Ahmed, Vamouss, Julien]
noms: [Laurie, Vincent, Valentin, Ahmed, Vamouss, Julien]
```

Par défaut, l'appel de la méthode add() ajoute l'élément en fin de liste. C'est le cas par exemple de l'instruction noms.add("Julien"). Mais avec la méthode add(), il est également possible de spécifier l'indice de position auquel on souhaite insérer une élément dans la liste. C'est le cas de l'instruction noms.add(2,"Valentin") qui insère l'élément à l'indice 2 (position 3 de la liste). A noter que la première position commence toujours par l'indice 0. Et l'ajout d'un élément autre qu'en fin de liste décale tous les éléments à droite d'une position.

NB : Il existe aussi plusieurs variantes de la méthode add() que sont notamment addFirst() qui ajoute un élément en première position dans la liste et addLast() qui ajoute un élément en dernière position dans la liste.

6.4.3.2 Ajouter plusieurs éléments à un LinkedList : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un LinkedList, la méthode addAll() permet d'ajouter plusieurs éléments à un LinkedList en un seule fois. Par défaut, ces éléments sont ajoutés en fin de liste. Mais ils peuvent aussi être ajoutés en commençant à une position donnée dans le LinkedList. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un LinkedList.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de Integer
        List nums=new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
        // Ajoute les éléments à une position i donnée
        nums.addAll(2,Arrays.asList(14, 18, 20)); // Insère à l'indice 2
(troisième position)
        System.out.println("nums: "+nums.toString());
    }

 }
```

Ouput :

```
nums: [24, 17, 85, 44, 52, 63, 45, 10, 100, 91]
nums: [24, 17, 14, 18, 20, 85, 44, 52, 63, 45, 10, 100, 91]
```

6.4.3.3 Vérifier si un LinkedList contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un LinkedList contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Vérifier si Le LinkedList contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("Liste contient Ahmed: "+a);

        // Vérifier si Le LinkedList contient "Adams"
        boolean b= noms.contains("Adams");
        System.out.println("Liste contient Adams: "+b);

    }

}
```

Output

```
Liste contient Ahmed: true
Liste contient Adams: false
```

Dans l'exemple ci-dessous, la liste contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la liste ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.4.3.4 Récupérer un élément donné dans une LinkedList : la méthode get()

La méthode get() permet de récupérer et de renvoyer un élément d'un LinkedList en spécifiant sa position dans la liste. L'exemple ci-dessous illustre l'utilisation de la méthode get()

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Recupérer l'élément à l'indice 2 ( position 3)
        System.out.println("Premier élément: "+noms.get(0)); // Renvoie Laurie
        // Recupérer l'élément à l'indice 2 ( position 3)
        System.out.println("Troisième élément: "+noms.get(2)); // Renvoie Ahmed

    }
}
```

Output :

```
Premier élément: Laurie
Troisième élément: Ahmed
```

NB : Il existe aussi plusieurs variantes de la méthode get() que sont notamment getFirst() qui récupère l'élément en première position dans la liste et getLast() qui récupère l'élément en dernière position dans la liste.

6.4.3.5 Renvoyer l'indice d'un élément donné d'un LinkedList : la méthode indexOf()

Pour retrouver l'indice d'un élément d'un LinkedList, on utilise la méthode indexOf(). L'exemple ci-dessous montre l'utilisation de la méthode indexOf().

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        System.out.println("L'indice de Vincent est:
"+noms.indexOf("Vincent")); // Renvoie 1
        System.out.println("L'indice de Vamouss est:
"+noms.indexOf("Vamouss")); // Renvoie 3

    }
}
```

Output

```
L'indice de Vincent est: 1
L'indice de Vamouss est: 3
```

6.4.3.6 Supprimer un élément spécifique d'un LinkedList : la méthode remove()

La méthode remove() permet de supprimer un élément d'un LinkedList. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de String
        List noms =new LinkedList<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        noms.remove("Ahmed");
        System.out.println("La liste finale noms :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Laurie, Vincent, Ahmed, Vamouss]
La liste finale noms :[Laurie, Vincent, Vamouss]
```

NB : Il existe aussi plusieurs variantes de la méthode remove() que sont notamment removeFirst() qui récupère l'élément en première position dans la liste et removeLast() qui récupère l'élément en dernière position dans la liste.

6.4.3.7 Supprimer un ensemble de valeurs d'un LinkedList : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'un LinkedList. L'exemple ci-dessous montre l'utilisation de la méthode.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de Integer
        List nums= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
```

```
    }
}
```

Output :

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :[17, 85, 44, 52, 58]
```

6.4.3.8 Modifier la valeur située à une position donnée : la méthode set()

La méthode set() permet de modifier une valeur située à une position donnée dans un LinkedList. L'exemple ci-dessous fournit une illustration.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de Integer
        List nums= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Change la valeur à l'indice 2 à 55
        nums.set(1,55);
        //Change la valeur à l'indice 5 à 100
        nums.set(5,100);
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :   [24, 55, 85, 44, 52, 100, 26, 58]
```

6.4.3.9 Déterminer le nombre d'éléments d'un LinkedList : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un LinkedList.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de Integer
        List nums= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
```

```
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.4.3.10 Convertir un LinkedList en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet LinkedList en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.LinkedList;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un LinkedList de Integer
        List nums= new LinkedList<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("Le type LinkedList est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        for (int i=0;i<=nums.size()-1;i++ ){
            nums_array[i]=(Integer) nums.get(i);
        }
         System.out.println("Le type Array est :
"+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type LinkedList est : [24, 17, 85, 44, 52, 20, 26, 58]
Le type Array est : [24, 17, 85, 44, 52, 20, 26, 58]
```

6.5 Etude de la collection Vector

Le Vector est la collection la plus proche de l'ArrayList. Les deux classes implémentent la même interface List et partagent les mêmes méthodes. La seule différence majeure entre un Vector et un ArrayList est que le Vecteur est un objet synchronisé contrairement à l'ArrayList. En effet, il n'est pas possible d'avoir un accès concurrent à un même élément d'un Vector contrairement à un ArrayList. Cette particularité fait qu'il est conseillé d'utiliser

le Vector lorsqu'on a des traitements à réaliser sur des séquences de valeurs dans un environnement de multi-threadings. En effet, les ArrayList ne sont pas adaptés aux traitement en multi-threads car ses éléments ne sont pas synchronisés. Plusieurs threads peuvent accéder au même élément pour des usages différents : lecture ou modification. Ce qui peut générer des problèmes de cohérence. Dans un Vector, lorsqu'un élément est en cours d'utilisation par un thread, un lock est automatiquement mis sur l'élément pour empêcher des threads concurrents d'accéder en même temps à la même valeur. De ce point de vue, les Vectors sont plus thread-safe que les ArrayLists. Cette section a pour but de présenter les principales caractéristiques de la collection Vector. Compte tenu de la très grande similarité entre le Vector et l'ArrayList, nous reprenons les mêmes exemples et les mêmes commentaires de résultats comme ceux présentés dans la section consacrée à l'ArrayList. Pour une documentation complète sur la collection Vector, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/Vector.html

6.5.1 Créer un Vector

Tout comme un ArrayList, on peut créer un Vector en procédant de deux façons : soit déclarer un Vector vide et ajouter ensuite les éléments, soit définir le Vector en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un Vector.

6.5.1.1 Créer un Vector vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.Vector;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        // Initialise un Vector vide avec éléments de type Integer
        List numero= new Vector<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
le Vector


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous initialisons un Vector vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons que dans cette déclaration que le type réel de l'objet numero est bien Vector<Integer> mais sont type référence est List qui correspond à l'interface implémentée par la classe Vector.

Le Vector numero étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

6.5.1.2 Créer un Vector à partir d'une séquence de valeurs

On peut aussi créer un Vector directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création du Vector à partir une séquence de valeurs.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        // Initialise un Vector à partir d'une séquence initiale de données
        List numero= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le Vector

    }
 }
```

Output

```
La taille est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du Vector. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.5.1.3 Les types des éléments d'un Vector

Tous les éléments d'un Vector doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un Vector dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Vector dont les éléments sont String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le Vector
    }
 }
```

Output

```
Les élements sont: [Laurie, Vincent, Ahmed, Vamouss, Julien]
```

6.5.2 Itérateur d'un Vector: usage de la méthode iterator()

Tout comme l'ArrayList, le Vector dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection afin de réaliser une opération de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un Vector.

Soit un Vector nommé vec1 défini comme suit.

```
List vec1= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cet Vector et renvoyer un nouvel Vector nommé vec2 dont chaque élément est égal au double de l'élément initial du Vector vec1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du Vector vec1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;
public class Main {
```

```
    public static void main(String[] args) {
        // Vector initial
        List vec1= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("Les éléments de vec1 sont: "+vec1.toString());
        // Déclaration du Vector double
        List vec2 = new Vector<Integer>();

        Iterator iter=vec1.iterator(); //Création de l'itérator sur vec1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            vec2.add(new_elem);
        }
        System.out.println("Les éléments de vec2 sont: "+vec2.toString());
    }
}
```

Output

```
Les éléments de vec1 sont: [24, 17, 85, 44, 52]
Les éléments de vec2 sont: [48, 34, 170, 88, 104]
```

Dans cet exemple, le Vector vec2 est d'abord initialisé à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet vec1 en appelant la méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du Vector vec1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter le Vector de sortie vec2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de vec1 par 2. L'ensemble des opérations de récupération des éléments de vec1 et d'alimentation de vec2 a été réalisé dans la boucle suivante :

```
Iterator iter=vec1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    vec2.add(new_elem);
}
```

6.5.3 Opérations courantes sur un Vector

En plus de la méthode iterator(), l'objet Vector fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection Vector, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/Vector.html

6.5.3.1 Ajouter un élément à un Vector : la méthode add()

La méthode add() permet d'ajouter un élément à un Vector. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Ajoute un élément en fin de liste
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un élément à une position i donnée
        noms.add(2,"Valentin"); // Insère à l'indice 2 (troisième position)
        System.out.println("noms: "+noms.toString());
    }

 }
```

Output

```
noms: [Laurie, Vincent, Ahmed, Vamouss, Julien]
noms: [Laurie, Vincent, Valentin, Ahmed, Vamouss, Julien]
```

Par défaut, l'appel de la méthode add() ajoute l'élément en fin de liste. C'est le cas par exemple de l'instruction noms.add("Julien"). Mais avec la méthode add(), il est également possible de spécifier l'indice de position auquel on souhaite insérer une élément dans la liste. C'est le cas de l'instruction noms.add(2,"Valentin") qui insère l'élément à l'indice 2 (position 3 de la liste). A noter que la première position commence toujours par l'indice 0. Et l'ajout d'un élément autre qu'en fin de liste décale tous les éléments à droite d'une position.

6.5.3.2 Ajouter plusieurs éléments à un Vector : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un Vector, la méthode addAll() permet d'ajouter plusieurs éléments à un Vector en un seule fois. Par défaut, ces éléments sont ajoutés en fin de liste. Mais ils peuvent aussi être ajoutés en commençant à une position donné dans le Vector. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un Vector.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de Integer
        List nums=new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
        // Ajoute les éléments à une position i donnée
        nums.addAll(2,Arrays.asList(14, 18, 20)); // Insère à l'indice 2
(troisième position)
        System.out.println("nums: "+nums.toString());
    }

 }
```

Ouput :

```
nums: [24, 17, 85, 44, 52, 63, 45, 10, 100, 91]
nums: [24, 17, 14, 18, 20, 85, 44, 52, 63, 45, 10, 100, 91]
```

6.5.3.3 Vérifier si un Vector contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un Vector contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
```

```
        // Vérifier si Le Vector contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("Vector contient Ahmed: "+a);

        // Vérifier si Le Vector contient "Adams"
        boolean b= noms.contains("Adams");
        System.out.println("Vector contient Adams: "+b);

    }

}
```

Output

```
Vector contient Ahmed: true
Vector contient Adams: false
```

Dans l'exemple ci-dessous, la liste contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la liste ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.5.3.4 Récupérer un élément donné dans une Vector : la méthode get()

La méthode get() permet de récupérer et de renvoyer un élément d'un Vector en spécifiant sa position dans la liste. L'exemple ci-dessous illustre l'utilisation de la méthode get()

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Recupérer l'élément à l'indice 2 ( position 3)
        System.out.println("Premier élément: "+noms.get(0)); // Renvoie Laurie
        // Recupérer l'élément à l'indice 2 ( position 3)
        System.out.println("Troisième élément: "+noms.get(2)); // Renvoie Ahmed

    }
}
```

Output :

```
Premier élément: Laurie
Troisième élément: Ahmed
```

6.5.3.5 Renvoyer l'indice d'un élément donné d'un Vector : la méthode indexOf()

Pour retrouver l'indice d'un élément d'un Vector, on utilise la méthode indexOf(). L'exemple ci-dessous montre l'utilisation de la méthode indexOf().

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        System.out.println("L'indice de Vincent est:
"+noms.indexOf("Vincent")); // Renvoie 1
        System.out.println("L'indice de Vamouss est:
"+noms.indexOf("Vamouss")); // Renvoie 3

    }
}
```

Output

```
L'indice de Vincent est: 1
L'indice de Vamouss est: 3
```

6.5.3.6 Supprimer un élément spécifique d'un Vector : la méthode remove()

La méthode remove() permet de supprimer un élément d'un Vector. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List noms =new Vector<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        noms.remove("Ahmed");
        System.out.println("La liste finale noms :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Laurie, Vincent, Ahmed, Vamouss]
La liste finale noms :[Laurie, Vincent, Vamouss]
```

6.5.3.7 Supprimer un ensemble de valeurs d'un Vector : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'un Vector. L'exemple ci-dessous montre l'utilisation de la méthode.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de Integer
        List nums= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :[17, 85, 44, 52, 58]
```

6.5.3.8 Modifier la valeur située à une position donnée : la méthode set()

La méthode set() permet de modifier une valeur située à une position donnée dans un Vector. L'exemple ci-dessous fournit une illustration.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de Integer
        List nums= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Change la valeur à l'indice 2 à 55
        nums.set(1,55);
        //Change la valeur à l'indice 5 à 100
        nums.set(5,100);
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :   [24, 55, 85, 44, 52, 100, 26, 58]
```

6.5.3.9 Déterminer le nombre d'éléments d'un Vector : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un Vector.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de String
        List nums= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.5.3.10 Convertir un Vector en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet Vector en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.Vector;
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Définit un Vector de Integer
        List nums= new Vector<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le type Vector est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        for (int i=0;i<=nums.size()-1;i++ ){
            nums_array[i]=(Integer) nums.get(i);
        }
```

```
        System.out.println("Le type Array est :
"+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type Vector est : [24, 17, 85, 44, 52, 20, 26, 58]
Le type Array est : [24, 17, 85, 44, 52, 20, 26, 58]
```

6.6 Etude de la collection HashSet

Le HashSet est la collection qui permet de représenter les séquences de valeurs non dupliquées (le Set). La classe HashSet implémente l'interface Set. Dans un ArrayList, une même valeur d'élément peut se répéter plusieurs fois dans la liste. Alors que dans un HashSet, chaque valeur d'élément est représentée de manière unique. C'est une séquence organisée de telle sorte que si une valeur existe déjà dans la séquence, tout ajout de la même valeur dans la séquence est ignoré.

Hormis le fait qu'il n'autorise pas des valeurs dupliquées dans la séquence, le HashSet partage les mêmes caractéristiques qu'un ArrayList. De nombreuses méthodes applicables sur une séquence ArrayList sont aussi applicables sur une séquence HashSet.

Par ailleurs, il est important de noter que le HashSet n'attribue pas un indice fixe à un élément dans une séquence. Par conséquent, il n'est pas possible d'effectuer des traitements sur les éléments en se basant sur leur indice. C'est pourquoi des méthodes de type get(i) où i est l'index de l'élément ne sont pas applicables dans le cadre d'un HashSet. De même, lorsqu'on affiche les éléments d'un HashSet, l'ordre d'apparition des éléments n'est pas toujours le même d'un lancement à un autre.

L'objet de cette section est d'illustrer à travers des exemples les modes d'utilisation de la classe HashSet. Compte tenu de la très grande similarité entre le HashSet et l'ArrayList, nous reprenons les mêmes exemples et les mêmes commentaires de résultats comme ceux présentés dans la section consacrée à l'ArrayList. Pour une documentation complète sur la collection HashSet, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/HashSet.html

6.6.1 Créer un HashSet

Tout comme un ArrayList, on peut créer un HashSet en procédant de deux façons : soit déclarer un HashSet vide et ajouter ensuite les éléments, soit définir le HashSet en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un HashSet.

6.6.1.1 Créer un HashSet vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {

        // Initialise un HashSet vide avec éléments de type Integer
        Set numero= new HashSet<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
le HashSet


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous initialisons un HashSet vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons dans cette déclaration que le type réel de l'objet numero est bien HashSet<Integer> mais sont type référence est Set qui correspond à l'interface implémentée par la classe HashSet.

Le HashSet numero étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

6.6.1.2 Créer un HashSet à partir d'une séquence de valeurs

On peut aussi créer un HashSet directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création du HashSet à partir une séquence de valeurs.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {

        // Initialise un HashSet à partir d'une séquence initiale de données
        Set numero= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le HashSet

    }
}
```

Output

```
La taille est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du HashSet. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.6.1.3 Les types des éléments d'un HashSet

Tous les éléments d'un HashSet doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un HashSet dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // HashSet dont les éléments sont String
```

```
        Set noms =new HashSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
         noms.add("Julien");
         System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le HashSet
    }
}
```

Output

```
Les élements sont: [Laurie, Vincent, Ahmed, Vamouss, Julien]
```

6.6.2 Itérateur d'un HashSet: usage de la méthode iterator()

Tout comme les autres classes de collection, le HashSet dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection afin de réaliser une opération de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un HashSet.

Soit un HashSet nommé hs1 défini comme suit.

```
List hs1= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cet HashSet et renvoyer un nouvel HashSet nommé hs2 dont chaque élément est égal au double de l'élément initial du HashSet hs1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du HashSet hs1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;
public class Main {
    public static void main(String[] args) {
        // HashSet initial
        Set hs1= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("Les éléments de hs1 sont: "+hs1.toString());
        // Déclaration du HashSet double
        Set hs2 = new HashSet<Integer>();

        Iterator iter=hs1.iterator(); //Création de l'itérator sur hs1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            hs2.add(new_elem);
        }
        System.out.println("Les éléments de hs2 sont: "+hs2.toString());
    }
}
```

Output

```
Les éléments de hs1 sont: [24, 17, 85, 44, 52]
Les éléments de hs2 sont: [48, 34, 170, 88, 104]
```

Dans cet exemple, le HashSet hs2 est d'abord initialisé à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet hs1 en appelant la méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du HashSet hs1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter le HashSet de sortie hs2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de hs1 par 2. L'ensemble des opérations de récupération des éléments de hs1 et d'alimentation de hs2 a été réalisé dans la boucle suivante :

```
Iterator iter=hs1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    hs2.add(new_elem);
}
```

6.6.3 Opérations courantes sur un HashSet

En plus de la méthode iterator(), l'objet HashSet fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection HashSet, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/HashSet.html

6.6.3.1 Ajouter un élément à un HashSet : la méthode add()

La méthode add() permet d'ajouter un élément à un HashSet. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add(). A noter que le HashSet ne respecte pas nécessairement l'ordre d'insertion dans la séquence. A l'affichage, chaque valeur peut se trouver à une position aléatoirement définie.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
```

```
    public static void main(String[] args) {
        // Définit un HashSet de String
        Set noms =new HashSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Ajoute un élément
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un autre élément
        noms.add("Valentin");
        System.out.println("noms: "+noms.toString());
        // Ajoute une valeur déjà existante
        noms.add("Laurie");
        System.out.println("noms: "+noms.toString());
    }

}
```

Output

```
noms: [Laurie, Ahmed, Vincent, Vamouss, Julien]
noms: [Laurie, Valentin, Ahmed, Vincent, Vamouss, Julien]
noms: [Laurie, Valentin, Ahmed, Vincent, Vamouss, Julien]
```

L'appel de la méthode add() ajoute l'élément dans le set à une position aléatoirement choisie. De ce fait l'ordre d'insertion dans un HashSet n'a pas d'importance. De plus On remarque que lorsqu'on ajoute une valeur qui existe déjà dans le Set, cet ajout est ignoré car un Set n'autorise pas la duplication de valeurs comme pour le ArrayList.

6.6.3.2 Ajouter plusieurs éléments à un HashSet : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un HashSet, la méthode addAll() permet d'ajouter plusieurs éléments à un HashSet en un seule fois. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un HashSet.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de Integer
        Set nums=new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
    }

}
```

Ouput :

```
nums: [17, 52, 100, 85, 24, 10, 91, 44, 45, 63]
```

A l'affichage, chaque valeur ajoutée peut se trouver à une position aléatoirement définie. Car le HashSet ne respecte pas nécessairement la règle habituelle de l'insertion en fin de séquence.

6.6.3.3 Vérifier si un HashSet contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un HashSet contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de String
        Set noms =new HashSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Vérifier si Le HashSet contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("HashSet contient Ahmed: "+a);

        // Vérifier si Le HashSet contient "Adams"
        boolean b= noms.contains("Adams");
        System.out.println("HashSet contient Adams: "+b);

    }

}
```

Output

```
HashSet contient Ahmed: true
HashSet contient Adams: false
```

Dans l'exemple ci-dessous, la liste contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la liste ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.6.3.4 Supprimer un élément spécifique d'un HashSet : la méthode remove()

La méthode remove() permet de supprimer un élément d'un HashSet. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de String
        Set noms =new HashSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        noms.remove("Ahmed");
        System.out.println("La liste finale noms :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Laurie, Vincent, Ahmed, Vamouss]
La liste finale noms :[Laurie, Vincent, Vamouss]
```

6.6.3.5 Supprimer un ensemble de valeurs d'un HashSet : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'un HashSet. L'exemple ci-dessous montre l'utilisation de la méthode.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de Integer
        Set nums= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :[17, 85, 44, 52, 58]
```

6.6.3.6 Déterminer le nombre d'éléments d'un HashSet : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un HashSet.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de String
        Set nums= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.6.3.7 Convertir un HashSet en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet HashSet en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.HashSet;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
public class Main {
    public static void main(String[] args) {
        // Définit un HashSet de Integer
        Set nums= new HashSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le type HashSet est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        Iterator iter= nums.iterator();
        int i=0;
        while (iter.hasNext() && i<nums.size() ){
            nums_array[i]=((Integer) iter.next());
            i++;
        }
        System.out.println("Le type Array est : "+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type HashSet est : [24, 17, 85, 44, 52, 20, 26, 58]
Le type Array est : [24, 17, 85, 44, 52, 20, 26, 58]
```

6.7 Etude de la collection TreeSet

Tout comme le HashSet, le TreeSet permet une représentation des séquences de valeurs non dupliquées. Chaque élément apparaît une et une seule fois dans la séquence. Mais à la différence Hashet, le TreeSet trie et ordonne les éléments dans un ordre croissant. La classe TreeSet implémente trois interfaces que sont Set, SortedSet et NavigableSet. Etant donné que le TreeSet partage les mêmes caractéristiques qu'un HashSet, de nombreuses méthodes applicables sur une séquence HashSet le sont aussi applicables sur une séquence TreeSet. Par ailleurs, il est important de noter que, comme le HashSet le TreeSet n'attribue pas un indice fixe à un élément dans une séquence. Par conséquent, il n'est pas possible d'effectuer des traitements sur les éléments en se basant sur leur indice. C'est pourquoi des méthodes de type get(i) où i est l'index de l'élément ne sont pas applicables dans le cadre d'un TreeSet.

L'objet de cette section est d'illustrer à travers des exemples les modes d'utilisation de la classe TreeSet. Compte tenu de la très grande similarité entre le TreeSet et le HashSet, nous reprenons les mêmes exemples et les mêmes commentaires de résultats. Pour une documentation complète sur la collection TreeSet, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/TreeSet.html

6.7.1 Créer un TreeSet

Tout comme un HashSet, on peut créer un TreeSet en procédant de deux façons : soit déclarer un TreeSet vide et ajouter ensuite les éléments, soit définir le TreeSet en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un TreeSet.

6.7.1.1 Créer un TreeSet vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {

        // Initialise un TreeSet vide avec éléments de type Integer
        Set numero= new TreeSet<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
```

```
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
le TreeSet


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [17, 24, 44, 52, 85]
```

Dans cet exemple, nous initialisons un TreeSet vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons dans cette déclaration que le type réel de l'objet numero est bien TreeSet<Integer> mais sont type référence est Set qui correspond à l'interface implémentée par la classe TreeSet.

Le TreeSet numero étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

A noter que le TreeSet ordonne toujours les éléments de façon croissant, qu'il s'agisse des nombres ou des chaînes de caractères.

6.7.1.2 Créer un TreeSet à partir d'une séquence de valeurs

On peut aussi créer un TreeSet directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création du TreeSet à partir une séquence de valeurs.

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {

        // Initialise un TreeSet à partir d'une séquence initiale de données
        Set numero= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le TreeSet

    }
}
```

Output

```
La taille est :5
Les élements sont: [17, 24, 44, 52, 85]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du TreeSet. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.7.1.3 Les types des éléments d'un TreeSet

Tous les éléments d'un TreeSet doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un TreeSet dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // TreeSet dont les éléments sont String
       Set noms =new TreeSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le TreeSet
    }
}
```

Output

```
Les élements sont: [Ahmed, Julien, Laurie, Vamouss, Vincent]
```

6.7.2 Itérateur d'un TreeSet: usage de la méthode iterator()

Tout comme les autres classes de collection, le TreeSet dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments pour réaliser des opérations de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un TreeSet.

Soit un TreeSet nommé hs1 défini comme suit.

```
List ts1= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cet TreeSet et renvoyer un nouvel TreeSet nommé ts2 dont chaque élément est égal au double de l'élément initial du TreeSet ts1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du TreeSet ts1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;
public class Main {
    public static void main(String[] args) {
        // TreeSet initial
        Set ts1= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("Les éléments de ts1 sont: "+ts1.toString());
        // Déclaration du TreeSet double
        Set ts2 = new TreeSet<Integer>();

        Iterator iter=ts1.iterator(); //Création de l'itérator sur ts1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            ts2.add(new_elem);
        }
        System.out.println("Les éléments de ts2 sont: "+ts2.toString());
    }
}
```

Output

```
Les éléments de ts1 sont: [17, 24, 44, 52, 85]
Les éléments de ts2 sont: [34, 48, 88, 104, 170]
```

Dans cet exemple, le TreeSet ts2 est d'abord initialisé à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet ts1 en appelant la méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du TreeSet ts1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter le TreeSet de sortie ts2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de ts1 par 2. L'ensemble des opérations de récupération des éléments de ts1 et d'alimentation de ts2 a été réalisé dans la boucle suivante :

```
Iterator iter=ts1.iterator();
```

```
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    ts2.add(new_elem);
}
```

6.7.3 Opérations courantes sur un TreeSet

En plus de la méthode iterator(), l'objet TreeSet fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection TreeSet, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/TreeSet.html

6.7.3.1 Ajouter un élément à un TreeSet : la méthode add()

La méthode add() permet d'ajouter un élément à un TreeSet. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de String
        Set noms =new TreeSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Ajoute un élément
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un autre élément
        noms.add("Valentin");
        System.out.println("noms: "+noms.toString());
        // Ajoute une valeur déjà existante
        noms.add("Laurie");
        System.out.println("noms: "+noms.toString());
    }

}
```

Output

```
noms: [Ahmed, Julien, Laurie, Vamouss, Vincent]
noms: [Ahmed, Julien, Laurie, Valentin, Vamouss, Vincent]
noms: [Ahmed, Julien, Laurie, Valentin, Vamouss, Vincent]
```

Le TreeSet ordonne les éléments par ordre croissant. Ainsi, lorsqu'on insère un élément dans la séquence, cet élément se positionne automatiquement entre le dernier élément inférieur et le premier élément supérieur dans la séquence. C'est le cas par exemple de l'instruction noms.add("Julien") qui insère entre les éléments « Ahmed » et « Laurie ». On

remarque que lorsqu'on ajoute une valeur qui existe déjà dans le Set, cet ajout est ignoré car un Set n'autorise pas la duplication de valeurs comme pour le ArrayList.

6.7.3.2 Ajouter plusieurs éléments à un TreeSet : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un TreeSet, la méthode addAll() permet d'ajouter plusieurs éléments à un TreeSet en un seule fois. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un TreeSet.

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de Integer
        Set nums=new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
    }

}
```

Ouput :

```
nums: [10, 17, 24, 44, 45, 52, 63, 85, 91, 100]
```

6.7.3.3 Vérifier si un TreeSet contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un TreeSet contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de String
        Set noms =new TreeSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));

        // Vérifier si Le TreeSet contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("TreeSet contient Ahmed: "+a);

        // Vérifier si Le TreeSet contient "Adams"
```

```
        boolean b= noms.contains("Adams");
        System.out.println("TreeSet contient Adams: "+b);

    }

}
```

Output

```
TreeSet contient Ahmed: true
TreeSet contient Adams: false
```

Dans l'exemple ci-dessous, la liste contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la liste ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.7.3.4 Supprimer un élément spécifique d'un TreeSet : la méthode remove()

La méthode remove() permet de supprimer un élément d'un TreeSet. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de String
        Set noms =new TreeSet<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        noms.remove("Ahmed");
        System.out.println("La liste finale noms :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Ahmed, Laurie, Vamouss, Vincent]
La liste finale noms :[Laurie, Vamouss, Vincent]
```

6.7.3.5 Supprimer un ensemble de valeurs d'un TreeSet : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'un TreeSet. L'exemple ci-dessous montre l'utilisation de la méthode.

```
package com.tuto.collection;
import java.util.TreeSet;
```

```
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de Integer
        Set nums= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [17, 20, 24, 26, 44, 52, 58, 85]
La liste finale nums :[17, 44, 52, 58, 85]
```

6.7.3.6 Déterminer le nombre d'éléments d'un TreeSet : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un TreeSet.

```
package com.tuto.collection;
import java.util.TreeSet;
import java.util.Arrays;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de String
        Set nums= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.7.3.7 Convertir un TreeSet en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet TreeSet en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.TreeSet;
```

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;
public class Main {
    public static void main(String[] args) {
        // Définit un TreeSet de Integer
        Set nums= new TreeSet<Integer>(Arrays.asList(24, 17, 85, 44, 52, 20,
26, 58));
        System.out.println("Le type TreeSet est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        Iterator iter= nums.iterator();
        int i=0;
        while (iter.hasNext() && i<nums.size() ){
            nums_array[i]=((Integer) iter.next());
            i++;
        }
        System.out.println("Le type Array est : "+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type TreeSet est : [17, 20, 24, 26, 44, 52, 58, 85]
Le type Array est : [17, 20, 24, 26, 44, 52, 58, 85]
```

6.8 Etude de la collection HashMap

A la différence de toutes les collections que nous avons étudiées jusqu'à présent et dont les éléments sont des valeurs ou des objets singuliers, le HashMap est une collection dont les éléments sont des objets à double entrée constituée d'une clé et d'une valeur. Le HashMap est une classe qui implémente l'interface Map<k,v> où k représente la clé et v la valeur.

Cette section est consacrée à l'étude de la collection HashMap. Pour une documentation complète sur la collection HashMap, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/HashMap.html

6.8.1 Créer un HashMap

On peut créer un HashMap en initialisant dans un premier temps un objet HashMap vide et ajouter dans un second temps les éléments en utilisant la méthode put(). L'exemple ci-dessous illustre la création et l'alimentation d'un HashMap.

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {

        // Initialise un HashMap vide dont les clés de type String et les
valeurs de type Integer
```

```
        Map mp= new HashMap<String,Integer>();
        System.out.println("La taille initiale est :"+mp.size()); // renvoie 0
        // Ajoute des élements
        mp.put("James",24);
        mp.put("Valerie",17);
        mp.put("Ivan",35);
        mp.put("Jhon",44);
        mp.put("Victor",52);
        System.out.println("La taille finale est :"+mp.size()); // Renvoie 5
        System.out.println("Les élements sont: "+mp.toString()); // Affiche le
HashMap


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: {Victor=52, James=24, Valerie=17, Ivan=35, Jhon=44}
```

Dans cet exemple, nous initialisons un HashMap vide nommé mp dont les éléments sont des Map dont les clés sont de type String et les valeurs de type Integer. Remarquons dans cette déclaration que le type réel de l'objet mp est bien HashMap<String, Integer> mais son type référence est Map qui correspond à l'interface implémentée par la classe HashMap.

Le HashMap mp étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode put() sur l'objet mp. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. Chaque clé correspond à un prénom à laquelle on associe une valeur qui correspond à son âge, ici de type Integer. Le HashMap est une séquence de valeurs représentant une correspondance entre les clés et les valeurs.

A la suite de l'ajout de ces 5 éléments, la taille de l'objet mp devient 5.

A noter que le HashMap n'organise pas les éléments dans un ordre prédéfini. Par conséquent, on ne peut pas rechercher une clé en se basant sur son indice (sa position) dans la séquence. Comme nous allons le voir plus bas, pour récupérer une valeur donnée, il faut se baser sur sa clé en utilisant la méthode get(). Aussi, il faut noter qu'une clé ne peut pas se répéter dans un HashMap. En effet, à chaque ajout d'une nouvelle paire clé-valeur, cette paire remplace la paire existante qui a la même clé. En revanche, dans un HashMap, plusieurs clés peuvent avoir la même valeur.

6.8.2 Les types des éléments d'un HashMap

Tous les éléments d'un HashMap doivent avoir la même structure. C'est-à-dire que toutes les clés doivent être de même type et toutes les valeurs doivent également être de même type. Et ces types peuvent être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments

de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un HashMap dont les clés et les valeurs sont de type String.

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Arrays;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // HashMap dont les éléments sont String, String
        Map noms =new HashMap<String, String>();
        noms.put("001","Julien");
        noms.put("002","Laurie");
        noms.put("003","Vincent");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le HashMap
        noms.put("003","Vamouss"); // Modification de la valeur d'une clé
existante
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le HashMap
    }
}
```

Output

```
Les élements sont: {001=Julien, 002=Laurie, 003=Vincent}
Les élements sont: {001=Julien, 002=Laurie, 003=Vamouss}
```

6.8.3 Itérateur d'un HashMap: usage de la méthode keySet() et iterator()

Le HashMap dispose d'une méthode keySet() et d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection et de réaliser une opération de traitement. Cette sous-section montre l'utilisation de la méthode keySet() combinée avec la méthode iterator() sur un HashMap.

Soit un HashMap nommé hm1 défini comme suit.

```
Map hm1= new HashMap<String,Integer>();
hm1.put("James",24);
hm1.put("Valerie",17);
hm1.put("Ivan",35);
hm1.put("Jhon",44);
hm1.put("Victor",52);
```

On souhaite parcourir les éléments de cet HashMap et renvoyer un nouvel HashMap nommé hm2 dont chaque élément est égal au double de l'élément initial du HashMap hm1.

Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du HashMap hm1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Iterator;


public class Main {
    public static void main(String[] args) {

        Map hm1= new HashMap<String,Integer>();
        hm1.put("James",24);
        hm1.put("Valerie",17);
        hm1.put("Ivan",35);
        hm1.put("Jhon",44);
        hm1.put("Victor",52);

        // Déclaration du HashMap double
        Map hm2 = new HashMap<String,Integer>();
        //Boucle sur l'iterator
        Set keys= hm1.keySet();//Récupère les clés sous forme de set
        Iterator iter =keys.iterator();//Iterateur sur le set
        while (iter.hasNext()) {
          String k=(String) iter.next();
          Integer v= (Integer) hm1.get(k);
          hm2.put(k,v*2);
        }
        System.out.println("Les éléments de hm1 sont: "+hm1.toString());
        System.out.println("Les éléments de hm2 sont: "+hm2.toString());
    }
}
```

Output

```
Les éléments de hm1 sont: {Victor=52, James=24, Valerie=17, Ivan=35, Jhon=44}
Les éléments de hm2 sont: {Victor=104, James=48, Valerie=34, Ivan=70, Jhon=88}
```

Cet exemple nécessite un certain nombre de commentaires.

D'abord nous avons créé un premier HashMap vide nommé hm1 que nous alimentons avec cinq éléments dont les clés sont des prénoms et les valeurs l'âge correspondant à chaque prénom. Pour ajouter ces éléments, nous avons utilisé la méthode put() qui permet d'ajouter une clé et sa valeur correspondante.

Le HashMap hm2 est initialisé à vide et sert à recueillir les doubles des âges pour les prénoms déjà renseignés dans hm1. Cependant pour alimenter hm1, nous devons d'abord faire une itération sur les éléments de hm1 pour récupérer à la fois les clés mais aussi les valeurs et ainsi calculer les doubles des valeurs avant les insérer dans hm2. Ces opérations se déroulent en plusieurs instructions décrites ci-dessous.

Dans un premier temps, nous récupérons toutes les clés de hm1 en appelant la méthode keySet(). Cette méthode renvoie les clés sous forme d'une séquence de valeurs de type Set

(Voir les classes HashSet et TreeSet pour plus de détails sur les collections de type Set). En récupérant les clés de la collection hm1 sous forme de Set, nous pouvons maintenant appeler la méthode iterator sur cette collection pour parcourir chaque clé et récupérer sa valeur correspondante depuis hm1 en utilisant la méthode get(). A noter que la récupération de la clé et de sa valeur correspondante nécessite un cast pour retrouver le type original car dans l'itérateur, les clés et les valeurs se présentent sous forme d'Object. La boucle et l'ensemble des opérations de cast et de retraitement est effectuée à travers le bloc d'instructions ci-dessous.

```
Set keys= hm1.keySet();//Récupère les clés sous forme de set
Iterator iter =keys.iterator();//Iterateur sur le set
while (iter.hasNext()) {
  String k=(String) iter.next();
  Integer v= (Integer) hm1.get(k);
  hm2.put(k,v*2);
}
```

6.8.4 Opérations courantes sur un HashMap

En plus de l'usage des méthodes entrySet() et iterator(), l'objet HashMap fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection HashMap, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/HashMap.html

6.8.4.1 Récupérer un élément donné dans une HashMap : la méthode get()

La méthode get() permet de récupérer et de renvoyer un élément d'un HashMap en spécifiant la clé. L'exemple ci-dessous illustre l'utilisation de la méthode get()

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        hm.put("Ivan",28);
        System.out.println("La valeur de la clé James est: "+hm.get("James"));

    }
}
```

Output :

```
La valeur de la clé James est: 24
```

6.8.4.2 Récupérer toutes les clés d'un HashMap dans un Set : la méthode keySet()

La méthode keySet() permet de récupérer toutes les clés d'un HashMap et les stocker dans un Set, qui, comme nous l'avons déjà vu, est une collection dont les valeurs sont non dupliquées dans la séquence. La récupération des clés dans un set est une opération qui peut s'avérer utile dans de nombreuses situations. En effet, grâce à la méthode Iterator() de cet Set, on peut effectuer une boucle sur le HashMap d'origine pour effectuer des traitements sur les valeurs. L'exemple ci-dessous montre l'utilisation de la méthode keySet().

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        //Recup de toute les clés
        Set keys= hm.keySet();//Récupère les clés sous forme de set
        System.out.println("Les clés de hm sont: "+keys.toString());
    }
}
```

Output

```
Les clés de hm sont: [Victor, James, Valerie, Ivan, Jhon]
```

Connaissant les clés du HashMap, on peut maintenant construire une boucle par exemple pour récupérer les valeurs correspondantes. Cette boucle peut se présenter comme suit :

```
package com.tuto.collection;
import java.util.*;

public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
```

```
        hm.put("Victor",52);

        //Recup de toute les clés
        Set keys= hm.keySet();//Récupère les clés sous forme de set
        System.out.println("Les clés de hm sont: "+keys.toString());

        Iterator iter =keys.iterator();
        List values= new ArrayList<Integer>(); // Initialise un ArrayList vide
pour recueillir les valeurs
        while (iter.hasNext()){
            String k= (String)iter.next();
            Integer v=(Integer) hm.get(k); // Recup de la valeur pour la clé
            values.add(v);
        }
        System.out.println("Les valeurs de hm sont: "+values.toString());
    }
}
```

Output

```
Les clés de hm sont: [Victor, James, Valerie, Ivan, Jhon]
Les valeurs de hm sont: [52, 24, 17, 35, 44]
```

6.8.4.3Ajouter un élément à un HashMap : la méthode put()

Comme nous l'avons déjà vu, la méthode put() permet d'ajouter un élément à un HashMap. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        hm.put("Ivan",28);
        System.out.println("Les éléments de hm sont: "+hm.toString());

    }
}
```

Output

```
Les éléments de hm sont: {Victor=52, James=24, Valerie=17, Ivan=28, Jhon=44}
```

L'appel de la méthode put() ajoute l'élément dans le HashMap à une position aléatoirement choisie. De ce fait l'ordre d'insertion dans un HashMap n'a pas d'importance. De plus, on remarque que lorsqu'on ajoute une clé qui existe déjà dans le Set, la valeur existante est

modifiée. C'est le cas par exemple de la clé « Ivan » qui a été initialement insérée avec la valeur 38. Une deuxième insertion sur la même clé met la valeur à 28. C'est la dernière valeur insérée qui est retenue.

6.8.4.4 Ajouter plusieurs éléments à un HashMap : la méthode putAll()

A la différence de la méthode put() qui n'ajoute qu'un seul élément à la fois à un HashMap, la méthode putAll() permet d'ajouter plusieurs éléments à un HashMap en un seule fois. Cependant ces objets doivent se présenter sous forme de HashMap. L'exemple ci-dessous illustre l'utilisation de la méthode putAll() pour ajouter des éléments à un HashMap.

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm1= new HashMap<String,Integer>();
        hm1.put("James",24);
        hm1.put("Valerie",17);
        hm1.put("Ivan",35);
        hm1.put("Jhon",44);
        hm1.put("Victor",52);

        // Création d'un deuxième HashMap
        Map hm2= new HashMap<String,Integer>();
        // Ajout de tous les éléments de hm1 à hm2.
        hm2.putAll(hm1);
        System.out.println("Les éléments de hm2: "+hm2.toString());

    }
}
```

Ouput :

```
Les éléments de hm2: {Victor=52, Ivan=35, James=24, Valerie=17, Jhon=44}
```

6.8.4.5 Vérifier si un HashMap contient une clé donnée : la méthode containsKey()

La méthode containsKey() permet de vérifier si un HashMap contient une clé donnée. Elle renvoie true si la clé spécifiée se trouve dans la séquence des clés et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode containsKey().

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {
```

```
        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        // Vérifier si hm contient la clé "Ivan"
        boolean a= hm.containsKey("Ivan");
        System.out.println("HashMap contient Ivan: "+a);

        // Vérifier si hm contient la clé "Noel"
        boolean b= hm.containsKey("Noel");
        System.out.println("HashMap contient Noel: "+b);
    }
}
```

Output

```
HashMap contient Ivan: true
HashMap contient Noel: false
```

Dans l'exemple ci-dessous, le HashMap contient la clé « Ivan ». La méthode containsKey() renvoie donc true. A l'inverse, le HashMap ne contient pas la valeur « Noel ». La méthode containsKey() renvoie donc false.

A noter que la méthode containsKey() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.8.4.6 Vérifier si un HashMap contient une valeur donnée : la méthode containsValue()

La méthode containsValue() permet de vérifier si un HashMap contient une valeur donnée. Elle renvoie true si la valeur spécifiée se trouve dans la séquence des valeurs et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode containsValue().

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        // Vérifier si hm contient la valeur 17
        boolean a= hm.containsValue(17);
        System.out.println("HashMap contient 17: "+a);
```

```
        // Vérifier si hm contient la valeur 30
        boolean b= hm.containsValue(30);
        System.out.println("HashMap contient 30: "+b);
    }
}
```

Output

```
HashMap contient 17: true
HashMap contient 30: false
```

Dans l'exemple ci-dessous, le HashMap contient la valeur « 17 ». La méthode containsValue() renvoie donc true. A l'inverse, le HashMap ne contient pas la valeur « 30 ». La méthode containsValue() renvoie donc false.

A noter que, tout comme la méthode containsKey(), la méthode containsKey() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.8.4.7 Supprimer un élément spécifique d'un HashMap : la méthode remove()

La méthode remove() permet de supprimer un élément spécifique d'un HashMap en se basant sur sa clé. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        System.out.println("Le HashMap avant remove: "+hm.toString());
        // Supprimer l'élément dont la clé est Jhon
        hm.remove("Jhon");
        System.out.println("Le HashMap après remove: "+hm.toString());
    }
}
```

Output

```
Le HashMap avant remove: {Victor=52, James=24, Valerie=17, Ivan=35, Jhon=44}
Le HashMap après remove: {Victor=52, James=24, Valerie=17, Ivan=35}
```

6.8.4.8 Déterminer le nombre d'éléments d'un HashMap : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un HashMap.

```
package com.tuto.collection;
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm= new HashMap<String,Integer>();
        hm.put("James",25);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",35);
        hm.put("Jeremy",25);
        hm.put("Josiane",40);

        System.out.println("Le nombre d'éléments de hm est : "+hm.size());

    }
}
```

Output

```
Le nombre d'éléments de hm est : 7
```

6.9 Etude de la collection TreeMap

La collection TreeMap est une variante de la collection Map<k,v> où les éléments sont ordonnés suivant l'ordre naturel des clés k contrairement à la collection HashMap où les éléments ne sont pas du tout triés. Cependant en dehors de cette différence notable, le TreeMap et le HashMap partagent les mêmes cractéristiques. Et les méthodes applicables dans un HashMap le sont également dans le cadre d'un TreeMap. Cette section est consacrée à l'étude de la collection TreeMap en passant en revue les traitements courants réalisables sur cette collection. Pour une documentation complète sur la collection TreeMap, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html

6.9.1 Créer un TreeMap

On peut créer un TreeMap en initialisant dans un premier temps un objet TreeMap vide et ajouter dans un second temps les éléments en utilisant la méthode put(). L'exemple ci-dessous illustre la création et l'alimentation d'un TreeMap.

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {

        // Initialise un TreeMap vide dont les clés de type String et les
valeurs de type Integer
        Map mp= new TreeMap<String,Integer>();
        System.out.println("La taille initiale est :"+mp.size()); // renvoie 0
        // Ajoute des élements
        mp.put("James",24);
        mp.put("Valerie",17);
        mp.put("Ivan",35);
        mp.put("Jhon",44);
        mp.put("Victor",52);
        System.out.println("La taille finale est :"+mp.size()); // Renvoie 5
        System.out.println("Les élements sont: "+mp.toString()); // Affiche le
TreeMap


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: {Ivan=35, James=24, Jhon=44, Valerie=17, Victor=52}
```

Dans cet exemple, nous initialisons un TreeMap vide nommé mp dont les éléments sont des Map dont les clés sont de type String et les valeurs de type Integer. Remarquons dans cette déclaration que le type réel de l'objet mp est bien TreeMap<String, Integer> mais sont type référence est Map qui correspond à l'une des interfaces implémentées par la classe TreeMap.

Le TreeMap mp étant initialisé à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode put() sur l'objet mp. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. Chaque clé correspond à un prénom à laquelle on associe une valeur qui correspond à son âge, ici de type Integer. Le TreeMap est une séquence de valeurs représentant une correspondance entre les clés et les valeurs.

A la suite de l'ajout de ces 5 éléments, la taille de l'objet mp devient 5.

A noter que contrairement au HashMap, le TreeMap organise les éléments dans un ordre croisssant des clés. Et comme nous allons le voir plus bas, pour récupérer une valeur donnée, il faut se baser sur sa clé en utilisant la méthode get(). Aussi, il faut noter qu'une clé ne peut pas se répéter dans un TreeMap. En effet, à chaque ajout d'une nouvelle paire clé-valeur, cette paire remplace la paire existante qui a la même clé. En revanche, dans un TreeMap, plusieurs clés peuvent avoir la même valeur.

6.9.2 Les types des éléments d'un TreeMap

Tous les éléments d'un TreeMap doivent avoir la même structure. C'est-à-dire que toutes les clés doivent être de même type et toutes les valeurs doivent également être de même type. Et ces types peuvent être n'importe quel objet représentant une classe Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un TreeMap dont les clés et les valeurs sont de type String.

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Arrays;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // TreeMap dont les éléments sont String, String
        Map noms =new TreeMap<String, String>();
        noms.put("001","Julien");
        noms.put("002","Laurie");
        noms.put("003","Vincent");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le TreeMap
        noms.put("003","Vamouss"); // Modification de la valeur d'une clé
existante
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le TreeMap
    }
}
```

Output

```
Les élements sont: {001=Julien, 002=Laurie, 003=Vincent}
Les élements sont: {001=Julien, 002=Laurie, 003=Vamouss}
```

6.9.3 Itérateur d'un TreeMap: usage de la méthode keySet() et iterator()

Comme toutes les collections Java, le TreeMap dispose d'une méthode keySet() et d'un objet appelé Iterator permettant de faire une boucle sur les éléments de la collection et de réaliser une opération de traitement. Cette sous-section montre l'utilisation de la méthode keySet() combinée avec la méthode iterator() sur un TreeMap.

Soit un TreeMap nommé hm1 défini comme suit.

```
Map hm1= new TreeMap<String,Integer>();
hm1.put("James",24);
hm1.put("Valerie",17);
```

```
hm1.put("Ivan",35);
hm1.put("Jhon",44);
hm1.put("Victor",52);
```

On souhaite parcourir les éléments de cet TreeMap et renvoyer un nouvel TreeMap nommé hm2 dont chaque élément est égal au double de l'élément initial du TreeMap hm1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du TreeMap hm1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
import java.util.Set;
import java.util.Iterator;


public class Main {
    public static void main(String[] args) {

        Map hm1= new TreeMap<String,Integer>();
        hm1.put("James",24);
        hm1.put("Valerie",17);
        hm1.put("Ivan",35);
        hm1.put("Jhon",44);
        hm1.put("Victor",52);

        // Déclaration du TreeMap double
        Map hm2 = new TreeMap<String,Integer>();
        //Boucle sur l'iterator
        Set keys= hm1.keySet();//Récupère les clés sous forme de set
        Iterator iter =keys.iterator();//Iterateur sur le set
        while (iter.hasNext()) {
          String k=(String) iter.next();
          Integer v= (Integer) hm1.get(k);
          hm2.put(k,v*2);
        }
        System.out.println("Les éléments de hm1 sont: "+hm1.toString());
        System.out.println("Les éléments de hm2 sont: "+hm2.toString());
    }
}
```

Output

```
Les éléments de hm1 sont: {Ivan=35, James=24, Jhon=44, Valerie=17, Victor=52}
Les éléments de hm2 sont: {Ivan=70, James=48, Jhon=88, Valerie=34, Victor=104}
```

Dans cet exemple, nous avons d'abord créé un premier TreeMap vide nommé hm1 que nous alimentons avec cinq éléments dont les clés sont des prénoms et les valeurs l'âge correspondant à chaque prénom. Pour ajouter ces éléments, nous avons utilisé la méthode put() qui permet d'ajouter une clé et sa valeur correspondante.

Le TreeMap hm2 est initialisé à vide et sert à recueillir les doubles des âges pour les prénoms déjà renseignés dans hm1. Cependant pour alimenter hm1, nous devons d'abord faire une itération sur les éléments de hm1 pour récupérer à la fois les clés mais aussi les

valeurs et ainsi calculer les doubles des valeurs avant les insérer dans hm2. Ces opérations se déroulent en plusieurs instructions décrites ci-dessous.

Dans un premier temps, nous récupérons toutes les clés de hm1 en appelant la méthode keySet(). Cette méthode renvoie les clés sous forme d'une séquence de valeurs de type Set (Voir les classes HashSet et TreeSet pour plus de détails sur les collections de type Set). En récupérant les clés de la collection hm1 sous forme de Set, nous pouvons maintenant appeler la méthode iterator sur cette collection pour parcourir chaque clé et récupérer sa valeur correspondante depuis hm1 en utilisant la méthode get(). A noter que la récupération de la clé et de sa valeur correspondante nécessite un cast pour retrouver le type original car dans l'itérateur, les clés et les valeurs se présentent sous formes d'Object. La boucle et l'ensemble des opérations de cast et de retraitement est effectuée à travers le bloc d'instructions ci-dessous.

```
Set keys= hm1.keySet();//Récupère les clés sous forme de set
Iterator iter =keys.iterator();//Iterateur sur le set
while (iter.hasNext()) {
  String k=(String) iter.next();
  Integer v= (Integer) hm1.get(k);
  hm2.put(k,v*2);
}
```

6.9.4 Opérations courantes sur un TreeMap

En plus de l'usage des méthodes entrySet() et iterator(), l'objet TreeMap fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection TreeMap, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html

6.9.4.1 Récupérer un élément donné dans une TreeMap : la méthode get()

La méthode get() permet de récupérer et de renvoyer un élément d'un TreeMap en spécifiant la clé. L'exemple ci-dessous illustre l'utilisation de la méthode get()

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
import java.util.Set;
public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        hm.put("Ivan",28);
        System.out.println("La valeur de la clé James est: "+hm.get("James"));
```

```
    }
}
```

Output :

```
La valeur de la clé James est: 24
```

6.9.4.2 Récupérer toutes les clés d'un TreeMap dans un Set : la méthode keySet()

La méthode keySet() permet de récupérer toutes les clés d'un TreeMap et les stocker dans un Set, qui, comme nous l'avons déjà vu, est une collection dont les valeurs sont non dupliquées dans la séquence. La récupération des clés dans un set est une opération qui peut s'avérer utile dans de nombreuses situations. En effet, grâce à la méthode Iterator() de cet Set, on peut effectuer une boucle sur le TreeMap d'origine pour effectuer des traitements sur les valeurs. L'exemple ci-dessous montre l'utilisation de la méthode keySet().

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
import java.util.Set;
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        //Recup de toute les clés
        Set keys= hm.keySet();//Récupère les clés sous forme de set
        System.out.println("Les clés de hm sont: "+keys.toString());
    }
}
```

Output

```
Les clés de hm sont: [Ivan, James, Jhon, Valerie, Victor]
```

Connaissant les clés du TreeMap, on peut maintenant construire une boucle par exemple pour récupérer les valeurs correspondantes. Cette boucle peut se présenter comme suit :

```
package com.tuto.collection;
import java.util.*;

public class Main {
    public static void main(String[] args) {
```

```
        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        //Recup de toute les clés
        Set keys= hm.keySet();//Récupère les clés sous forme de set
        System.out.println("Les clés de hm sont: "+keys.toString());

        Iterator iter =keys.iterator();
        List values= new ArrayList<Integer>(); // Initialise un ArrayList vide
pour recueillir les valeurs
        while (iter.hasNext()){
            String k= (String)iter.next();
            Integer v=(Integer) hm.get(k); // Recup de la valeur pour la clé
            values.add(v);
        }
        System.out.println("Les valeurs de hm sont: "+values.toString());
    }
}
```

Output

```
Les clés de hm sont: [Ivan, James, Jhon, Valerie, Victor]
Les valeurs de hm sont: [35, 24, 44, 17, 52]
```

6.9.4.3Ajouter un élément à un TreeMap : la méthode put()

Comme nous l'avons déjà vu, la méthode put() permet d'ajouter un élément à un TreeMap. L'exemple ci-dessous montre deux modes d'utilisation de la méthode add().

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
import java.util.Set;
public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        hm.put("Ivan",28);
        System.out.println("Les éléments de hm sont: "+hm.toString());

    }
}
```

Output

```
Les éléments de hm sont: {Ivan=28, James=24, Jhon=44, Valerie=17, Victor=52}
```

L'appel de la méthode put() ajoute l'élément dans le TreeMap à une position située entre la dernière clé inférieure et la première clé supérieure à la clé de l'élément courant. De plus, on remarque que lorsqu'on ajoute une clé qui existe déjà dans le Set, la valeur existante est modifiée. C'est le cas par exemple de la clé « Ivan » qui a été initialement insérée avec la valeur 38. Une deuxième insertion sur la même clé met la valeur à 28. C'est la dernière valeur insérée qui est retenue.

6.9.4.4Ajouter plusieurs éléments à un TreeMap : la méthode putAll()

A la différence de la méthode put() qui n'ajoute qu'un seul élément à la fois à un TreeMap, la méthode putAll() permet d'ajouter plusieurs éléments à un TreeMap en un seule fois. Cependant ces objets doivent se présenter sous forme de TreeMap. L'exemple ci-dessous illustre l'utilisation de la méthode putAll() pour ajouter des éléments à un TreeMap.

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm1= new TreeMap<String,Integer>();
        hm1.put("James",24);
        hm1.put("Valerie",17);
        hm1.put("Ivan",35);
        hm1.put("Jhon",44);
        hm1.put("Victor",52);

        // Création d'un deuxième TreeMap
        Map hm2= new TreeMap<String,Integer>();
        // Ajout de tous les éléments de hm1 à hm2.
        hm2.putAll(hm1);
        System.out.println("Les éléments de hm2: "+hm2.toString());

    }
}
```

Ouput :

```
Les éléments de hm2: {Ivan=35, James=24, Jhon=44, Valerie=17, Victor=52}
```

6.9.4.5Vérifier si un TreeMap contient une clé donnée : la méthode containsKey()

La méthode containsKey() permet de vérifier si un TreeMap contient une clé donnée. Elle renvoie true si la clé spécifiée se trouve dans la séquence des clés et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode containsKey().

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);

        // Vérifier si hm contient la clé "Ivan"
        boolean a= hm.containsKey("Ivan");
        System.out.println("TreeMap contient Ivan: "+a);

        // Vérifier si hm contient la clé "Noel"
        boolean b= hm.containsKey("Noel");
        System.out.println("TreeMap contient Noel: "+b);
    }
}
```

Output

```
TreeMap contient Ivan: true
TreeMap contient Noel: false
```

Dans l'exemple ci-dessus, le TreeMap contient la clé « Ivan». La méthode containsKey() renvoie donc true. A l'inverse, le TreeMap ne contient pas la valeur « Noel ». La méthode containsKey() renvoie donc false.

A noter que la méthode containsKey() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.9.4.6 Vérifier si un TreeMap contient une valeur donnée : la méthode containsValue()

La méthode containsValue() permet de vérifier si un TreeMap contient une valeur donnée. Elle renvoie true si la valeur spécifiée se trouve dans la séquence des valeurs et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode containsValue().

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;
public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
```

```
        hm.put("Jhon",44);
        hm.put("Victor",52);

        // Vérifier si hm contient la valeur 17
        boolean a= hm.containsValue(17);
        System.out.println("TreeMap contient 17: "+a);

        // Vérifier si hm contient la valeur 30
        boolean b= hm.containsValue(30);
        System.out.println("TreeMap contient 30: "+b);
    }
}
```

Output

```
TreeMap contient 17: true
TreeMap contient 30: false
```

Dans l'exemple ci-dessous, le TreeMap contient la valeur « 17 ». La méthode containsValue() renvoie donc true. A l'inverse, le TreeMap ne contient pas la valeur « 30 ». La méthode containsValue() renvoie donc false.

A noter que, tout comme la méthode containsKey(), la méthode containsValue() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.9.4.7 Supprimer un élément spécifique d'un TreeMap : la méthode remove()

La méthode remove() permet de supprimer un élément spécifique d'un TreeMap en se basant sur sa clé. L'exemple ci-dessous illustre l'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",24);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",52);
        System.out.println("Le TreeMap avant remove: "+hm.toString());
        // Supprimer l'élément dont la clé est Jhon
        hm.remove("Jhon");
        System.out.println("Le TreeMap après remove: "+hm.toString());
    }
}
```

Output

```
Le TreeMap avant remove: {Ivan=35, James=24, Jhon=44, Valerie=17, Victor=52}
Le TreeMap après remove: {Ivan=35, James=24, Valerie=17, Victor=52}
```

6.9.4.8 Déterminer le nombre d'éléments d'un TreeMap : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un TreeMap.

```
package com.tuto.collection;
import java.util.TreeMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {

        Map hm= new TreeMap<String,Integer>();
        hm.put("James",25);
        hm.put("Valerie",17);
        hm.put("Ivan",35);
        hm.put("Jhon",44);
        hm.put("Victor",35);
        hm.put("Jeremy",25);
        hm.put("Josiane",40);

        System.out.println("Le nombre d'éléments de hm est : "+hm.toString());

    }
}
```

Output

```
Le nombre d'éléments de hm est : 7
```

6.10 Etude de la collection PriorityQueue

Comme son nom l'indique, la PriorityQueue est une collection qui attribue un ordre de priorité aux éléments qui la compose. En effet dans une Queue standard, les éléments sont traités en mode FIFO (First-In-First-Out). Ce qui signifie que l'ordre de priorité est défini en fonction de l'ordre d'arrivée. C'est l'élément qui est ajouté en premier à la queue (en utilisant la méthode add() ou la méthode offer()) est aussi l'élément qui sera retiré en premier de la queue (en utilisant soit la méthode remove() ou la méthode poll()). Dans le cas d'une PriorityQueue, la priorité est définie non pas sur la base d'un ordre d'arrivée mais plutôt sur la base d'une règle de tri. Par défaut, les éléments sont triés suivants l'ordre croissant des valeurs. Ainsi l'élément qui a la plus petite valeur occupe la tête de la queue et aura donc la priorité par rapport aux autres. Ainsi, en appelant la méthode remove() ou poll(), cette valeur sera retirée en premier même si elle venait d'être ajoutée à la queue. L'ordre de priorité peut être également définie suivant une règle que l'utilisateur aura, dans ce cas lui-même, spécifiée.

L'objet de cette section est d'illustrer à travers des exemples concrets les modes d'utilisation de la classe PriorityQueue. Pour une documentation complète sur la collection PriorityQueue, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/PriorityQueue.html

6.10.1 Créer une PriorityQueue

On peut créer une PriorityQueue en procédant de deux façons : soit déclarer un PriorityQueue vide et ajouter ensuite les éléments, soit définir la PriorityQueue en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un PriorityQueue.

6.10.1.1 Créer une PriorityQueue vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {

        // Initialise un PriorityQueue vide avec éléments de type Integer
        Queue numero= new PriorityQueue<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
la PriorityQueue


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [17, 24, 85, 44, 52]
```

Dans cet exemple, nous initialisons une PriorityQueue vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons dans cette déclaration que le type réel de l'objet numero est bien PriorityQueue<Integer> mais sont type référence est Queue qui correspond à l'interface implémentée par la classe PriorityQueue.

La PriorityQueue numero étant initialisée à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés à la liste en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

A noter que, par défaut la PriorityQueue détermine la priorité en se basant sur l'ordre naturel des valeurs des éléments. L'élément qui la plus petite valeur sera celui qui sera consommée en premier en utilisant par exemple la méthode poll(). Attention à ne pas confondre l'ordre de priorité et l'ordre d'affichage des éléments lorsqu'on fait un print sur le contenu de la PriorityQueue. En effet, l'ordre d'affichage des éléments reste aléatoire (voir exemple ci-dessus). Nous reviendrons plus tard sur les méthodes courantes de traitement d'une PriorityQueue.

6.10.1.2 Créer un PriorityQueue à partir d'une séquence de valeurs

On peut aussi créer une PriorityQueue directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création du PriorityQueue à partir une séquence de valeurs issue d'une List.

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {

        // Initialise un PriorityQueue à partir d'une séquence initiale de
données
        Queue numero= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le PriorityQueue

    }
}
```

Output

```
La taille est :5
Les élements sont: [17, 24, 85, 44, 52]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du PriorityQueue. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.10.1.3 Les types des éléments d'un PriorityQueue

Tous les éléments d'une PriorityQueue doivent être de même type. Et ce type peut être n'importe quel objet représentant une clase Java (String, Integer, Float, classe d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un PriorityQueue dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // PriorityQueue dont les éléments sont String
        Queue noms =new PriorityQueue<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le PriorityQueue
    }
}
```

Output

```
Les élements sont: [Ahmed, Julien, Laurie, Vamouss, Vincent]
```

6.10.2 Itérateur d'un PriorityQueue: usage de la méthode iterator()

Tout comme les autres classes de collection, la PriorityQueue dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments pour réaliser des opérations de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un PriorityQueue.

Soit un PriorityQueue nommé hs1 défini comme suit.

```
List pq1= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cette PriorityQueue et renvoyer un nouvel PriorityQueue nommé pq2 dont chaque élément est égal au double de l'élément initial du PriorityQueue pq1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du PriorityQueue pq1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
```

```
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;
public class Main {
    public static void main(String[] args) {
        // PriorityQueue initial
        Queue pq1= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("Les éléments de pq11 sont: "+pq1.toString());
        // Déclaration du PriorityQueue double
        Queue pq2 = new PriorityQueue<Integer>();

        Iterator iter=pq1.iterator(); //Création de l'itérator sur pq1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            pq2.add(new_elem);
        }
        System.out.println("Les éléments de pq2 sont: "+pq2.toString());
    }
}
```

Output

```
Les éléments de pq11 sont: [17, 24, 85, 44, 52]
Les éléments de pq2 sont: [34, 48, 170, 88, 104]
```

Dans cet exemple, la PriorityQueue pq2 est d'abord initialisée à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet pq1 en appelant la méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du PriorityQueue pq1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter la PriorityQueue de sortie pq2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de pq1 par 2. L'ensemble des opérations de récupération des éléments de pq1 et d'alimentation de pq2 a été réalisé dans la boucle suivante :

```
Iterator iter=pq1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    pq2.add(new_elem);
}
```

6.10.3 Opérations courantes sur un PriorityQueue

En plus de la méthode iterator(), l'objet PriorityQueue fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs. Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection PriorityQueue, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/PriorityQueue.html

6.10.3.1 Ajouter un élément à une PriorityQueue : les méthodes add() et offer()

La méthode add() permet d'ajouter un élément à un PriorityQueue. On peut aussi utiliser la méthode offer() comme alternative à la méthode add() pour ajouter un éléméent à une PriorityQueue. L'exemple ci-dessous montre l'utilisation de la méthode add() et de la méthode offer().

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de String
        Queue noms =new PriorityQueue<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));
        // Ajoute un élément
        noms.add("Julien");
        System.out.println("noms: "+noms.toString());
        // Ajoute un autre élément
        noms.offer("Valentin");
        System.out.println("noms: "+noms.toString());
        // Ajoute une valeur déjà existante
        noms.add("Laurie");
        System.out.println("noms: "+noms.toString());
    }

}
```

Output

```
noms: [Ahmed, Julien, Laurie, Vincent, Vamouss]
noms: [Ahmed, Julien, Laurie, Vincent, Vamouss, Valentin]
noms: [Ahmed, Julien, Laurie, Vincent, Vamouss, Valentin, Laurie]
```

A la différence des collections de type Set, la collection PriorityQueue ajoute les valeurs même si celles-ci existent déjà. C'est le cas par exemple de la valeur « Laurie » que nous avons ajouté deux fois dans la queue.

6.10.3.2 Ajouter plusieurs éléments à un PriorityQueue : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un PriorityQueue, la méthode addAll() permet d'ajouter plusieurs éléments à un PriorityQueue en un seule fois. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un PriorityQueue.

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de Integer
        Queue nums=new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
    }
}
```

Ouput :

```
nums: [10, 17, 45, 24, 52, 85, 63, 44, 100, 91]
```

6.10.3.3 Vérifier si une PriorityQueue contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un PriorityQueue contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de String
        Queue noms =new PriorityQueue<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));

        // Vérifier si La PriorityQueue contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("PriorityQueue contient Ahmed: "+a);

        // Vérifier si La PriorityQueue contient "Adams"
        boolean b= noms.contains("Adams");
```

```
        System.out.println("PriorityQueue contient Adams: "+b);

    }

}
```

Output

```
PriorityQueue contient Ahmed: true
PriorityQueue contient Adams: false
```

Dans l'exemple ci-dessous, la queue contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la queue ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.10.3.4 Récupérer le premier élément d'une PriorityQueue : les méthodes poll(), peek() et element()

Une priorityQueue propose trois méthodes pour récupérer l'élément le plus prioritaire, c'est-à-dire l'élément en tête de la queue. Il s'agit des méthodes poll(), peek() et element(). Il existe cependant une différence légère entre les trois méthodes. La méthode poll() récupère le premier élément et ensuite le supprime de la queue. Les méthodes peek() et element() renvoient toutes les deux le premier élément sans le supprimer dans la queue. Par contre, la méthode element() renvoie un exception lorsque la queue est vide, contrairement à la méthode peek() qui renvoie une valeur nulle. L'exemple ci-dessous illustre l'utilisation des trois méthodes pour récupérer le premier élément de la queue (observer l'état de la queue après l'appel de la méthode poll()).

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit une PriorityQueue initiale
        Queue nums=new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("Etat initiale Queue: "+nums.toString());
        // Appel de la méthode element
        System.out.println("Premier element avec element(): "+nums.element());
        System.out.println("Etat Queue après element(): "+nums.toString());
        // Appel de la méthode peek
        System.out.println("Premier element avec peek(): "+nums.peek());
        System.out.println("Etat Queue après peek(): "+nums.toString());
        // Appel de la méthode poll
        System.out.println("Premier element avec poll(): "+nums.poll());
        System.out.println("Etat Queue après poll(): "+nums.toString());
```

```
    }
}
```

Ouput :

```
Etat initiale Queue: [17, 24, 85, 44, 52]
Premier element avec element(): 17
Etat Queue après element(): [17, 24, 85, 44, 52]
Premier element avec peek(): 17
Etat Queue après peek(): [17, 24, 85, 44, 52]
Premier element avec poll(): 17
Etat Queue après poll(): [24, 44, 85, 52]
```

6.10.3.5 Supprimer un élément spécifique d'une PriorityQueue : la méthode remove()

La méthode remove() permet de supprimer un élément d'une PriorityQueue. Cet élément est spécifié en argument de la méthode. En revanche, lorsque la méthode remove() est appelée sans argument, elle renvoie l'élément situé en tête de la queue et le supprime de la queue. La méthode remove() sans argument se comporte de la même manière que la méthode poll() à la seule différence que la méthode poll() renvoie une valeur nulle si la queue est vide alors que remove() renvoie une exception. L'exemple ci-dessous illustre les deux modes d'utilisation de la méthode remove().

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de String
        Queue noms =new PriorityQueue<String>(Arrays.asList("Laurie",
"Vincent", "Ahmed", "Vamouss"));
        System.out.println("La liste initiale noms :"+noms.toString());
        // Appel avec argument
        noms.remove("Ahmed");
        System.out.println("La liste après remove 1 :"+noms.toString());
        // Appel sans argument
        noms.remove();
        System.out.println("La liste après remove 2 :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Ahmed, Vamouss, Laurie, Vincent]
La liste après remove 1 :[Laurie, Vamouss, Vincent]
La liste après remove 2 :[Vamouss, Vincent]
```

6.10.3.6 Supprimer un ensemble de valeurs d'une PriorityQueue : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'une PriorityQueue. L'exemple ci-dessous montre l'utilisation de la méthode removeAll().

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de Integer
        Queue nums= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52, 20, 26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [17, 24, 20, 44, 52, 85, 26, 58]
La liste finale nums :[17, 44, 52, 85, 58]
```

6.10.3.7 Déterminer le nombre d'éléments d'une PriorityQueue : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un PriorityQueue.

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de Integer
        Queue nums= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52, 20, 26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.10.3.8 Convertir un PriorityQueue en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet PriorityQueue en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.PriorityQueue;
import java.util.PriorityQueue;
import java.util.Arrays;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Définit un PriorityQueue de Integer
        Queue nums= new PriorityQueue<Integer>(Arrays.asList(24, 17, 85, 44,
52, 20, 26, 58));
        System.out.println("Le type PriorityQueue est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        for (int i=0;i<=nums_objects.length-1;i++ ){
            nums_array[i]=(Integer) nums_objects[i];
        }
        System.out.println("Le type Array est : "+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type PriorityQueue est : [17, 24, 20, 44, 52, 85, 26, 58]
Le type Array est : [17, 24, 20, 44, 52, 85, 26, 58]
```

6.11 Etude de la collection ArrayDeque

L'ArrayDeque (*Array Double Ended Queue*) et prononcé ArrayDeck est une queue spéciale qui permet de traiter les éléments des deux côtés de la queue : à partir du début et à partir de la fin de la queue. Contrairement à la PriorityQueue, qui récupère le premier élément en appelant les méthodes poll(), peek(), element() ou remove(), l'ArrayDeque offre des méthodes supplémentaires pour récupérer le premier ou le dernier élément de la queue. Elle propose par exemples des méthodes comme removeFirst(), removeLast(), getFirst(), getLast(), addFirst(), addLast(). La classe ArrayDeque implémente l'interface Deque (*Double Ended Queue*). A noter que, dans un ArrayDeque, contrairement à une PriorityQueue, les éléments ne sont pas triés dans un ordre spécifique. Les éléments sont plutôt ordonnés selon leur ordre d'arrivée. De ce point de vue, l'ArrayDeque se présente comme un ArrayList. Toutefois, les éléments d'un ArrayDeque ne sont pas indicés, c'est-à-dire qu'il n'est pas possible de récupérer un élément spécifique en indiquant sa position dans la séquence, faisant par get(i) où i est l'indice positionnel. Par ailleurs, contrairement

à une queue standard, les éléments ne sont pas consommés en mode First-In-First-Out (FIFO). Dans un ArrayDeque, les éléments peuvent être consommés des deux côtés de la queue.

Dans cette cette section, nous allons passer en revue quelques propriétés de la classe ArrayDeque. Pour une documentation complète sur la classe ArrayDeque, consulter la page : https://docs.oracle.com/javase/10/docs/api/java/util/ArrayDeque.html

6.11.1 Créer une ArrayDeque

On peut créer un ArrayDeque en procédant de deux façons : soit déclarer un ArrayDeque vide et ajouter ensuite les éléments, soit définir l'ArrayDeque en lui passant directement une séquence de valeurs. Les exemples ci-dessous illustrent les deux modes de création d'un ArrayDeque.

6.11.1.1Créer un ArrayDeque vide et ajouter des éléments

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {

        // Initialise un ArrayDeque vide avec éléments de type Integer
        Deque numero= new ArrayDeque<Integer>();
        System.out.println("La taille initiale est :"+numero.size()); //
renvoie 0
        // Ajoute des élements
        numero.add(24);
        numero.add(17);
        numero.add(85);
        numero.add(44);
        numero.add(52);
        System.out.println("La taille finale est :"+numero.size()); // Renvoie
5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
la ArrayDeque


    }
}
```

Output :

```
La taille initiale est :0
La taille finale est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous initialisons un ArrayDeque vide nommé numero dont les éléments sont prévus pour être de type Integer. Remarquons dans cette déclaration que le type réel

de l'objet numero est bien ArrayDeque<Integer> mais son type référence est Deque qui correspond à l'interface implémentée par la classe ArrayDeque.

L'ArrayDeque numero étant initialisée à vide, sa taille (dimension) est alors égale à 0. Mais cette dimension évoluera au fur et à mesure que des éléments sont ajoutés. Les éléments sont ajoutés en fin de queue en utilisant la méthode add() sur l'objet numero. Dans l'exemple ci-dessus, nous avons ajouté cinq éléments. A la suite de ces ajouts, la taille de l'objet numero devient 5.

Concernant le positionnement des éléments et leur affichage, on constate que l'ArrayDeque se présente plutôt comme un ArrayList. Les éléments sont affichés tels qu'ils sont ajoutés dans la queue.

6.11.1.2 Créer un ArrayDeque à partir d'une séquence de valeurs

On peut aussi créer un ArrayDeque directement à partir d'une séquence de valeurs sans avoir à passer l'ajout des éléments avec la méthode *add()*. L'exemple ci-dessous montre la création de l'ArrayDeque à partir une séquence de valeurs issue d'une List.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {

        // Initialise un ArrayDeque à partir d'une séquence initiale de données
        Deque numero= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44,
52));
        System.out.println("La taille est :"+numero.size()); // renvoie 5
        System.out.println("Les élements sont: "+numero.toString()); // Affiche
Le ArrayDeque

    }
}
```

Output

```
La taille est :5
Les élements sont: [24, 17, 85, 44, 52]
```

Dans cet exemple, nous passons directement la séquence (24, 17, 85, 44, 52) en tant qu'argument du ArrayDeque. Néanmoins, la séquence de valeurs doit d'abord être préparée et présentée sous forme de liste ordinaire. D'où l'utilisation de l'instruction Arrays.asList().

6.11.1.3 Les types des éléments d'un ArrayDeque

Tous les éléments d'un ArrayDeque doivent être de même type. Et ce type peut être n'importe quel objet représentant une classe Java (String, Integer, Float, classe

d'utilisateur, etc..). Comme déjà évoqué précédemment, les éléments de types primitifs sont représentés par leur type wrapper. Par exemple Integer pour le type primitif int, Boolean pour le type primitif boolean (voir Tableau 11 pour plus de détails sur la correspondance entre les types primitifs et les classes wrappers).

L'exemple ci-dessous illustre la création et la modification d'un ArrayDeque dont les éléments sont de type String.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // ArrayDeque dont les éléments sont String
        Deque noms =new ArrayDeque<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        noms.add("Julien");
        System.out.println("Les élements sont: "+noms.toString()); // Affiche
Le ArrayDeque
    }
}
```

Output

```
Les élements sont: [Laurie, Vincent, Ahmed, Vamouss, Julien]
```

6.11.2 Itérateur d'un ArrayDeque: usage de la méthode iterator()

Tout comme les autres classes de collection, l'ArrayDeque dispose d'un objet appelé Iterator permettant de faire une boucle sur les éléments pour réaliser des opérations de traitement. Cette sous-section montre l'utilisation de la méthode iterator() sur un ArrayDeque.

Soit un ArrayDeque nommé hs1 défini comme suit.

```
List adq1= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52));
```

On souhaite parcourir les éléments de cette ArrayDeque et renvoyer un nouvel ArrayDeque nommé adq2 dont chaque élément est égal au double de l'élément initial du ArrayDeque adq1. Pour cela, on peut élaborer un itérateur pour parcourir chaque élément du ArrayDeque adq1. Le programme de traitement qui permet de réaliser ces opérations se présente comme suit :

```
package com.tuto.collection;
import java.util.Iterator;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;
public class Main {
    public static void main(String[] args) {
        // ArrayDeque initial
```

```
        Deque adq1= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52));
        System.out.println("Les éléments de adq1 sont: "+adq1.toString());
        // Déclaration du ArrayDeque double
        Deque adq2 = new ArrayDeque<Integer>();

        Iterator iter=adq1.iterator(); //Création de l'itérator sur pq1
        //Boucle sur l'iterator
        while (iter.hasNext()){
            Integer elem=(Integer) iter.next(); // Recap élément et cast
            Integer new_elem=elem*2; // Double élément
            adq2.add(new_elem);
        }
        System.out.println("Les éléments de adq2 sont: "+adq2.toString());
    }
}
```

Output

```
Les éléments de adq1 sont: [24, 17, 85, 44, 52]
Les éléments de adq2 sont: [48, 34, 170, 88, 104]
```

Dans cet exemple, l'ArrayDeque adq2 est d'abord initialisée à vide dans un premier temps. Dans un deuxième temps, nous créons l'iterator iter sur l'objet adq1 en appelant la méthode iterator(). Dans un troisième temps, nous faisons une boucle sur l'iterator afin de récupérer chaque élément du ArrayDeque adq1. Cette boucle est réalisée en combinant la structure de contrôle while() et en appelant la méthode hasNext() sur l'objet Iterator. La méthode hasNext() est un pointeur qui se déplace d'un pas pour chaque itération de la boucle et renvoie la valeur true pour l'élément courant. Ce qui permet donc de parcourir tous les éléments. L'élément courant est récupéré en appelant la méthode next() sur l'itérateur. Avec la méthode next(), chaque élément est récupéré avec le type Object qui nécessite parfois d'être casté dans le type d'origine en utilisant l'opérateur de cast symbolisé par les parenthèses.

Pour alimenter l'ArrayDeque de sortie adq2, nous utilisons la méthode add() afin de pouvoir ajouter les éléments individuels obtenus en multipliant les éléments initiaux de adq1 par 2. L'ensemble des opérations de récupération des éléments de adq1 et d'alimentation de adq2 a été réalisé dans la boucle suivante :

```
Iterator iter=adq1.iterator();
while (iter.hasNext()){
    Integer elem=(Integer) iter.next();
    Integer new_elem=elem*2;
    adq2.add(new_elem);
}
```

6.11.3 Opérations courantes sur un ArrayDeque

En plus de la méthode iterator(), l'objet ArrayDeque fournit plusieurs méthodes qui permettent de réaliser de multiples opérations de traitement sur les séquences de valeurs.

Dans cette sous-section, nous allons passer en revue certaines de ces méthodes. Pour une documentation complète sur la collection ArrayDeque, consulter la page :

https://docs.oracle.com/javase/10/docs/api/java/util/ArrayDeque.html

6.11.3.1 Ajouter un élément à un ArrayDeque : les méthodes add(), offer(), addFirst(), offerFirst(), addLast() et offerLast().

Les méthodes add() et offer() ainsi que leur différentes variantes permettent chacune d'ajouter un élément à un ArrayDeque. Par défaut, les méthodes add() et offer() ajoutent l'élément en fin de queue (dernier élément). Mais avec les variante *First() et *Last(), on peut expliciter la position à la quelle on souhaite insérer l'élément. Avec les méthodes *First(), l'élément est ajouté en début de queue. Et avec les méthodes *Last(), l'élément est ajouté en fin de queue.. L'exemple ci-dessous montre l'utilisation des méthodes add() et offer() et leurs différentes variantes.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de String
        Deque noms =new ArrayDeque<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss"));
        // Utilisation de add
        noms.add("Julien");
        System.out.println("add noms: "+noms.toString());
        // Utilisation de offer
        noms.offer("Valentin");
        System.out.println("offer noms: "+noms.toString());
        // Utilisation addFirst
        noms.addFirst("Robert");
        System.out.println("addFirst noms: "+noms.toString());
        // Utilisation offerFirst
        noms.offerFirst("Annick");
        System.out.println("offerFirst noms: "+noms.toString());
        // Utilisation addLast
        noms.addLast("Kelly");
        System.out.println("addLast noms: "+noms.toString());
        // Utilisation offerLast
        noms.offerLast("Omar");
        System.out.println("offerLast noms: "+noms.toString());
    }

}
```

Output

```
add noms: [Laurie, Vincent, Ahmed, Vamouss, Julien]
offer noms: [Laurie, Vincent, Ahmed, Vamouss, Julien, Valentin]
addFirst noms: [Robert, Laurie, Vincent, Ahmed, Vamouss, Julien, Valentin]
offerFirst noms: [Annick, Robert, Laurie, Vincent, Ahmed, Vamouss, Julien,
Valentin]
```

```
addLast noms: [Annick, Robert, Laurie, Vincent, Ahmed, Vamouss, Julien,
Valentin, Kelly]
offerLast noms: [Annick, Robert, Laurie, Vincent, Ahmed, Vamouss, Julien,
Valentin, Kelly, Omar]
```

6.11.3.2 Ajouter plusieurs éléments à un ArrayDeque : la méthode addAll()

A la différence de la méthode add() qui n'ajoute qu'un seul élément à la fois à un ArrayDeque, la méthode addAll() permet d'ajouter plusieurs éléments à un ArrayDeque en une seule fois. Ces éléments sont alors ajoutés en fin de queue. L'exemple ci-dessous illustre l'utilisation de la méthode addAll() pour ajouter des éléments à un ArrayDeque.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de Integer
        Deque nums=new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52));

        // Ajoute les éléments en fin de liste
        nums.addAll(Arrays.asList(63, 45, 10, 100, 91));
        System.out.println("nums: "+nums.toString());
    }
}
```

Ouput :

```
nums: [24, 17, 85, 44, 52, 63, 45, 10, 100, 91]
```

6.11.3.3 Vérifier si un ArrayDeque contient un élément donné : la méthode contains()

La méthode contains() permet de vérifier si un ArrayDeque contient un élément représenté par une valeur donnée. La méthode constains() renvoie true si la valeur spécifiée se trouve dans la liste et false sinon. L'exemple ci-dessous illustre l'utilisation de la méthode contains().

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de String
        Deque noms =new ArrayDeque<String>(Arrays.asList("Laurie", "Vincent",
```

```
"Ahmed", "Vamouss"));

        // Vérifier si L'ArrayDeque contient "Ahmed"
        boolean a= noms.contains("Ahmed");
        System.out.println("ArrayDeque contient Ahmed: "+a);

        // Vérifier si L'ArrayDeque contient "Adams"
        boolean b= noms.contains("Adams");
        System.out.println("ArrayDeque contient Adams: "+b);

    }

}
```

Output

```
ArrayDeque contient Ahmed: true
ArrayDeque contient Adams: false
```

Dans l'exemple ci-dessous, la queue contient l'élément « Ahmed ». La méthode contains() renvoie donc true. A l'inverse, la queue ne contient pas la valeur « Adams ». La méthode contains() renvoie donc false.

A noter que la méthode contains() peut être utilisée pour définir et exécuter des instructions conditionnelles en utilisant les structures de contrôle if.. else. Ainsi, on peut prévoir un certain nombre d'instructions lorsque la valeur est true ou lorsque la valeur est false.

6.11.3.4 Récupérer le premier élément d'un ArrayDeque : les méthodes peek(), element(),getFirst(), poll(), peekFirst(),pollFirst() , getLast(),peekLast(),pollLast()

Les méthodes peek(), element(),getFirst(), poll(), peekFirst() et pollFirst() récupèrent et renvoie le premier élément de la queue. Et les méthodes getLast(),peekLast() et pollLast() renvoie le dernier élément de la queue. A noter que la méthode poll() et ses deux variantes pollFirst() et pollLast() suppriment l'élément dans la queue tandis que les autres méthodes laisent la queue inchangée. L'exemple ci-dessous illustre l'utilisation des différentes méthodes (observer l'état de la queue après l'appel de la méthode poll() ou des deux variantes pollFirst() et pollLast()).

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque initiale
        Deque nums=new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52,
55, 9,21, 34));
        System.out.println("Etat initial Queue: "+nums.toString());
        // Appel de la méthode peek()
```

```
        System.out.println("Premier élément avec peek(): "+nums.peek());
        System.out.println("Etat Queue après peek(): "+nums.toString());
        // Appel de la méthode element()
        System.out.println("Premier élément avec element(): "+nums.element());
        System.out.println("Etat Queue après element(): "+nums.toString());
        // Appel de la méthode poll()
        System.out.println("Premier élément avec poll(): "+nums.poll());
        System.out.println("Etat Queue après poll(): "+nums.toString());
        // Appel de la méthode getFirst()
        System.out.println("Premier élément avec getFirst():
"+nums.getFirst());
        System.out.println("Etat Queue après getFirst(): "+nums.toString());
        // Appel de la méthode poll()
        System.out.println("Premier élément avec poll(): "+nums.poll());
        System.out.println("Etat Queue après poll(): "+nums.toString());
        // Appel de la méthode peekFirst()
        System.out.println("Premier élément avec peekFirst() :
"+nums.peekFirst());
        System.out.println("Etat Queue après peekFirst(): "+nums.toString());
        // Appel de la méthode pollFirst()
        System.out.println("Premier élément avec pollFirst()  :
"+nums.pollFirst());
        System.out.println("Etat Queue après pollFirst(): "+nums.toString());
        // Appel de la méthode getLast
        System.out.println("Premier élément avec getLast()  :
"+nums.getLast());
        System.out.println("Etat Queue après getLast(): "+nums.toString());
        // Appel de la méthode peekLast()
        System.out.println("Premier élément avec peekLast()  :
"+nums.peekLast());
        System.out.println("Etat Queue après peekLast(): "+nums.toString());
        // Appel de la méthode pollLast()
        System.out.println("Premier élément avec pollLast()  :
"+nums.pollLast());
        System.out.println("Etat Queue après pollLast(): "+nums.toString());
    }
}
```

Ouput :

```
Etat initial Queue: [24, 17, 85, 44, 52, 55, 9, 21, 34]
Premier élément avec peek(): 24
Etat Queue après peek(): [24, 17, 85, 44, 52, 55, 9, 21, 34]
Premier élément avec element(): 24
Etat Queue après element(): [24, 17, 85, 44, 52, 55, 9, 21, 34]
Premier élément avec poll(): 24
Etat Queue après poll(): [17, 85, 44, 52, 55, 9, 21, 34]
Premier élément avec getFirst(): 17
Etat Queue après getFirst(): [17, 85, 44, 52, 55, 9, 21, 34]
Premier élément avec poll(): 17
Etat Queue après poll(): [85, 44, 52, 55, 9, 21, 34]
Premier élément avec peekFirst() : 85
Etat Queue après peekFirst(): [85, 44, 52, 55, 9, 21, 34]
Premier élément avec pollFirst()  : 85
Etat Queue après pollFirst(): [44, 52, 55, 9, 21, 34]
Premier élément avec getLast()  : 34
Etat Queue après getLast(): [44, 52, 55, 9, 21, 34]
Premier élément avec peekLast()  : 34
Etat Queue après peekLast(): [44, 52, 55, 9, 21, 34]
```

```
Premier élément avec pollLast()  : 34
Etat Queue après pollLast(): [44, 52, 55, 9, 21]
```

L'élément en tête de queue est toujours supprimé après l'appel d'une des variantes de la méthode poll().

6.11.3.5 Supprimer un élément spécifique d'un ArrayDeque : la méthode remove(), removeFirst() et removeLast()

La méthode remove() permet de supprimer un élément d'une ArrayDeque. Cet élément est spécifié en argument de la méthode. En revanche, lorsque la méthode remove() est appelée sans argument, elle renvoie l'élément situé en tête de la queue et le supprime de la queue. La méthode remove() sans argument se comporte de la même manière que la méthode poll() à la seule différence que la méthode poll() renvoie une valeur nulle si la queue est vide alors que remove() renvoie une exception.

Comme leur nom indique, les méthodes removeFirst() et removeLast() renvoient et suppriment respectivement le premier et le dernier élément de la queue.

L'exemple ci-dessous illustre l'utilisation de chacune des variantes de la méthode remove().

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de String
        Deque noms =new ArrayDeque<String>(Arrays.asList("Laurie", "Vincent",
"Ahmed", "Vamouss","Braham"));
        System.out.println("La liste initiale noms :"+noms.toString());
        // Appel remove avec argument
        noms.remove("Ahmed");
        System.out.println("La liste après remove 1 :"+noms.toString());
        // Appel remove sans argument
        noms.remove();
        System.out.println("La liste après remove 2 :"+noms.toString());
        // Appel removeFirst
        noms.removeFirst();
        System.out.println("La liste après remove 3 :"+noms.toString());
        // Appel removeLast
        noms.removeLast();
        System.out.println("La liste après remove 4 :"+noms.toString());
    }
}
```

Output

```
La liste initiale noms :[Laurie, Vincent, Ahmed, Vamouss, Braham]
La liste après remove 1 :[Laurie, Vincent, Vamouss, Braham]
La liste après remove 2 :[Vincent, Vamouss, Braham]
La liste après remove 3 :[Vamouss, Braham]
La liste après remove 4 :[Vamouss]
```

6.11.3.6 Supprimer un ensemble de valeurs d'un ArrayDeque : la méthode removeAll()

La méthode removeAll() permet de supprimer un ensemble de valeurs d'une ArrayDeque. L'exemple ci-dessous montre l'utilisation de la méthode removeAll().

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de Integer
        Deque nums= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("La liste initiale nums : "+nums.toString());
        //Supprimer les valeurs 24, 20 et 26
        nums.removeAll(Arrays.asList(24, 20, 26));
        System.out.println("La liste finale nums :"+nums.toString());
    }
}
```

Output :

```
La liste initiale nums : [24, 17, 85, 44, 52, 20, 26, 58]
La liste finale nums :[17, 85, 44, 52, 58]
```

6.11.3.7 Déterminer le nombre d'éléments d'un ArrayDeque : la méthode size()

L'exemple ci-dessous illustre l'utilisation de la méthode size() pour déterminer le nombre d'éléments d'un ArrayDeque.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de Integer
        Deque nums= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("Le nombre d'éléments de nums est : "+nums.size());
    }
}
```

Output

```
Le nombre d'éléments de nums est : 8
```

6.11.3.8 Convertir un ArrayDeque en Array : la méthode toArray()

La méthode toArray() permet de convertir un objet ArrayDeque en un objet de type Array. Toutefois, la conversion se fait en deux étapes. D'abord, on utilise la méthode toArray() pour construire un Array dont les éléments sont des objects. Ensuite, on fait une boucle sur les éléments de cet Array d'Objects, caster chaque élément pour reconstituer les valeurs initiales des éléments. L'exemple ci-dessous illustre l'utilisation de cette procédure.

```
package com.tuto.collection;
import java.util.ArrayDeque;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Définit un ArrayDeque de Integer
        Deque nums= new ArrayDeque<Integer>(Arrays.asList(24, 17, 85, 44, 52,
20, 26, 58));
        System.out.println("Le type ArrayDeque est : "+nums.toString());
        Object [] nums_objects=nums.toArray();
        Integer [] nums_array= new Integer [nums.size()]; // Initialiser
l'Array de type Integer
        for (int i=0;i<=nums_objects.length-1;i++ ){
            nums_array[i]=(Integer) nums_objects[i];
        }
        System.out.println("Le type Array est : "+Arrays.toString(nums_array));
    }
}
```

Output

```
Le type ArrayDeque est : [24, 17, 85, 44, 52, 20, 26, 58]
Le type Array est : [24, 17, 85, 44, 52, 20, 26, 58]
```

7 GESTION DES FLUX ENTRÉES/SORTIES

7.1 Généralités sur les flux Entrées/Sorties

7.1.1 Présentation

Les opérations Entrées/Sorties (E/S) encore connues sous le terme d'opérations Input/Output (I/O) sont des opérations incontournables dans les programmes Java. Elles concernent notamment les opérations d'écriture et de lecture sur des sources externes (mémoire, réseaux distant, écran, etc.), la gestion de fichiers (création, suppression, lecture et écriture), la gestion des objets sérialisés (création, enregistrement et récupération), etc. Les opérations Entrées/Sorties portent exclusivement sur des flux de données. D'où l'appellation « flux Entrées/Sorties ».

Les flux Entrées/Sorties se comportent comme des tuyaux de drainage permettant de conduire les données d'un point A à un point B. Lorsqu'il s'agit de transporter les données d'une source externe vers l'intérieur du programme, on parle de flux Entrées (Input Stream). Et lorsqu'il s'agit de transporter les données du programme vers un système externe, on parle de flux Sorties (Output Stream). Le système externe fournissant ou accueillant les données peut être soit un fichier, un réseau distant, un espace mémoire, un périphérique de saisie (clavier) ou un écran-utilisateur. Ces systèmes sont couramment appelés réservoirs de données ou *sinks* .

7.1.2 Types des flux Entrées/Sorties : les flux texte et flux binaires

On distingue deux principaux types de flux Entrées/Sorties : les flux texte et les flux binaires. Dans le cas des flux binaires, l'information est reçue depuis le sink (respectivement transmise au sink) sans subir de transformations entre la source et la destination. Les données des flux binaires étant transcrites en langage machine, elles ne sont donc pas directement lisibles par l'être humain. Quant aux flux textes, ils véhiculent des chaînes de caractères, qui contrairement aux flux binaires, sont lisibles par l'être humain. Cependant le drainage des données sous forme de texte plat nécessite parfois de transformer et de formater l'information de telle sorte que le flux renvoie une suite de caractères (lisible par l'être humain). L'ensemble des opérations Entrées/Sorties présentées dans ce chapitre portent soit sur des flux binaires, soit sur des flux texte.

7.1.3 Les principales classes de gestions des flux Entrées/Sorties

Java propose plusieurs classes abstraites permettant de gérer les flux Entrées/Sorties. Il s'agit notamment des classes :

- **InputStream** : permettant des lire des binaires
- **OutputStream** : permettant d'écrire des binaires

- **Reader** : permettant des lire des caractères
- **Writer** : permettant d'écrire des caractères
- **RandomAccessFile** : classe spécifique permettant une lecture en accès direct au contenu d'un fichier binaire plutôt qu'un accès séquentiel.

L'objectif de ce chapitre est présenter l'usage de chacune de ces classes et leur différentes classes dérivées dans la gestion des flux Entrées/Sorties.

7.2 Gestion des flux Entrées (Input streams)

Comme indiqué plus haut, les flux Entrées servent à conduire les flux de données depuis un réservoir de données (sink) vers l'intérieur d'un programme. Le sink peut être un périphérique de saisie (clavier), un fichier, un réseau distant (network socket) ou un espace mémoire.

Un flux Entrée est généralement matérialisé par une instance de l'une des classes dérivées de la classe abstraite InputStream. Les classes de gestion des flux Entrées les plus couramment utilisées sont : FileInputStream, DataInputStream, BufferedInputStream, ByteArrayInputStream, ObjectInputStream, InputStreamReader, BufferedReader, CharArrayReader, FileReader, FilterInputStream, Scanner[17], etc.

La page suivante présente la documentation de la classe InputStream ainsi que ses classes dérivées: https://docs.oracle.com/javase/10/docs/api/java/io/InputStream.html

L'utilisation d'un flux d'Entrée dans un programme Java se passe en quatre étapes :

1. **Création du flux d'Entrée** : cette étape consiste à instancier un objet de type stream (flux) en utilisant l'une des classes dérivées de la classe abstraite InputStream dont la liste a été présentée plus haut. En instanciant l'objet flux, on doit toujours préciser le sink à partir duquel les données sont consommées. Ex : clavier, fichier, réseau distant, mémoire, etc.
2. **Lecture/consommation de données**: cette étape se réalise généralement par l'appel de la méthode read() de l'objet flux créé à l'étape 1. A noter toutefois que la méthode à appeler dépend de la nature de la classe instanciée pour créer le flux. Par exemple pour un objet de la classe BufferedReader, la méthode de lecture est readLine() au lieu de read().
3. **Exploitation des données lues** : cette étape consiste à appliquer traitements sur les données lues dans le but de répondre à l'objectif de l'utilisateur : création d'un objet à partir des données récupérées, alimentation d'une base de données, etc…
4. **Fermeture du flux** : à la fin du traitement des données consommées, on procède à la fermeture du flux Entrée en appelant la méthode close() sur l'objet instancié au départ pour ouvrir le flux.

[17] La classe Scanner n'est pas une sous-classe de l'interface InputStream. C'est un Iterator permettant de lire un flux (fichier ou chaîne de caractères) "mot" par "mot" en se basant sur un délimiteur défini à l'avance.

Dans cette section, nous allons présenter quelques cas d'utilisation des flux Entrées. Il s'agit notamment de la lecture de flux depuis un écran de saisie et la lecture de flux d'un fichier et d'un fichier binaire.

7.2.1 Lecture des flux Entrées à partir d'une saisie-écran (clavier)

La création de flux Entrée à partir d'une saisie-écran est faite en combinant trois classes. D'abord une instance de la classe InputStreamReader dont l'argument d'instanciation est la variable static System.in dont l'appel invite l'utilisateur à faire une saisie via le clavier. Ensuite, cette instance de la classe InputStreamReader sert d'argument d'instanciation de la classe BufferedReader qui permet de bufférisier les textes saisis par l'utilisateur. L'exemple ci-dessous résume toutes ces étapes et illustre la création d'un flux Entrée à partir d'une saisie-utilisateur depuis un clavier.

```
package com.tuto.io;
import java.io.* ;

public class Main {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            System.out.print("Saisissez votre nom, svp : "); // Ex : Mister
Kelly
            String nom = br.readLine();
            System.out.println("Hello" + nom);
            br.close();
        } catch (IOException e) {
            System.err.println(e);
        }

    }
}
```

Output

```
Hello Mister Kelly
```

Dans cet exemple, nous avons saisi le texte « Mister Kelly ». Et le programme nous a renvoyé « Hello Mister Kelly ».

A noter que les informations saisies par l'utilisateur sont récupérées par une variable statique *in* appartenant à la classe java.lang.System.in. Ce flux est d'abord récupéré dans l'instance de la classe InputStreamReader qui est une classe permettant de convertir un stream de bytes en un stream de caractères. Ensuite, en instanciant la classe BufferedReader, on transforme le stream en une suite de caractères bufférisés. L'appel de la méthode readLine() sur cet objet permet de récupérer la valeur du texte que nous affichons à l'écran par la suite en appelant la méthode println().

7.2.2 Lecture d'un fichier de texte plat : usage de la classe FileReader

Dans cette section nous montrons l'usage de la classe FileReader pour lire et afficher le contenu d'un fichier de texte plat. L'exemple ci-dessous illustre l'utilisation de la classe FileReader.

```
package com.tuto.io;
import java.io.* ;

public class Main {
    public static void main(String[] args) {
        try {
            String ligne ;
            BufferedReader br = new BufferedReader (new FileReader
("src/resources/monFichier.txt")) ;
            do
            { ligne = br.readLine() ;
                if (ligne != null) System.out.println (ligne) ;
            }
            while (ligne != null) ;
            br.close () ;
        } catch (IOException e) {
            System.err.println(e);
        }

    }
}
```

Output

```
Bonjour. Ceci est un exemple de fichier de texte.
Vous allez apprendre à lire le contenu d'un fichier texte
```

Dans l'exemple ci-dessus, nous disposons d'un fichier texte nommé monFichier.txt. Ce fichier est déposé dans le répertoire src/resources[18] et contient deux lignes de texte. L'exemple consiste à lire ce fichier et à afficher son contenu.

Comme on peut le constater, la lecture du fichier texte commence d'abord par instancier la classe FileReader qui est, par la suite encapsulée dans la classe BufferedReader pour instancier le flux Entrée. Et pour accéder au contenu du flux, on appelle la méthode readLine(). A noter que cette méthode est un curseur qui se place sur une nouvelle ligne à chaque appel. C'est la raison pour laquelle, pour pouvoir lire et afficher tout le contenu du fichier monFichier.txt, il faut construire une boucle DO.. WHILE. Cette boucle doit s'arrêter lorsque la valeur renvoyée par la méthode readLine() devient nulle. Ce qui signifie la fin du fichier. A noter toutefois qu'au lieu d'utiliser la boucle DO... WHILE avec la condition de la ligne non nulle marquant la fin du fichier, on pouvait aussi directement utiliser la méthode

[18] A noter que le fichier peut être situé dans n'importe quelle arborescence situé dans le classpath d'exécution du code Java.

ready() de l'objet BufferedReader qui permet automatiquement de détecter la fin du fichier. L'exemple ci-dessous montre l'utilisation de la méthod ready().

```
package com.tuto.io;
import java.io.* ;

public class Main {
        public static void main(String[] args) {
                try {
                        FileReader fileReader = new
FileReader("src/resources/monFichier.txt");
                        BufferedReader br = new BufferedReader(fileReader);
                        while (br.ready()) {
                                String line = br.readLine();

                                System.out.print(line);

                                System.out.println();
                        }
                        br.close();
                        fileReader.close();
                } catch (Exception e) {
                        e.printStackTrace();
                }

        }
}
```

Dans cet exemple, nous avons remplacé l'instruction while (ligne != null) par while br.ready().

Remarquons que pour accéder au contenu du fichier et afficher chaque ligne, nous avons appelé la méthode readLine() dans une boucle. Ce qui peut être parfois très long pour des fichiers de grande taille. Dans ce genre de situations, il est préférable de se trouver vers des classes spécifiques de gestion fichiers dont par example java.nio.file.Files. Files est une classe static qui offre une méthode readAllLines() permettant de lire d'un coup l'ensemble des lignes disponibles dans un fichier de texte plat. Nous présenterons plus tard l'usage de la classe Files dans le chapitre suivant consacré à la gestion des fichiers.

7.2.3 Lecture d'un fichier binaire : usage des classes DataInputStream et RandomAccessFile

On distingue deux modes de lecture des fichiers binaires : la lecture séquentielle et la lecture directe (random access). Dans la lecture séquentielle, on accède aux informations de gauche à droite et de haut en bas en respectant l'ordre dans lequel elles sont disposées dans le fichier. Dans la lecture directe, le curseur de lecture est placé directement là où l'information se trouve sans être obligé de parcourir toutes les informations qui la précède. Dans cette sous-section, nous présentons les deux modes de lecture des fichiers binaires.

7.2.3.1 Lecture séquentielle d'un fichier binaire : la classe DataInputStream

L'exemple ci-dessous montre la lecture séquentielle d'un fichier contenant des données binaires.

```
package com.tuto.io;
import java.io.* ;

public class Main {
        public static void main(String[] args) throws IOException   {

                /*Création d'un fichier binaire à partir d'un texte string */
                FileOutputStream fouts=new
FileOutputStream("src/resources/myBinaryFile.dat");
                DataOutputStream douts=new DataOutputStream(fouts);
                douts.writeUTF("Ceci est la première ligne");
                douts.writeUTF("Ceci est la deuxième ligne");
                douts.writeUTF("Ceci est la troisième ligne");
                fouts.close();
                douts.close();

                /*Lecture du fichier binaire précédemment créé */
                DataInputStream dis = new DataInputStream( new FileInputStream
("src/resources/myBinaryFile.dat")) ;
                boolean eof = false ;
                String line=null;
                while (!eof)  {
                        try {
                                line = dis.readUTF () ;
                        }
                catch (EOFException e) {
                                eof = true ;
                }
                        if (!eof) System.out.println (line) ;
                }
                dis.close () ;

        }
}
```

Output :

```
Ceci est la première ligne
Ceci est la deuxième ligne
Ceci est la troisième ligne
```

Dans cet exemple, nous commençons d'abord par créer et alimenter un fichier binaire nommé myBinaryFile.dat. Ce fichier est créé dans le dossier src/resources situé dans l'arborescence du projet Java. Nous avons créé ce fichier en ajoutant trois lignes de texte en utilisant la méthode writeUTF() de l'objet DataOutputStream.

Dans un deuxième temps, nous lisons ce fichier binaire en utilisant un objet de la classe DataInputStream. Ensuite, nous affichons chaque ligne du fichier en utilisant la méthode readUTF(). Il s'agit en effet d'une lecture séquentielle car la méthode est appelée plusieurs

fois dans une boucle qui permet de parcourir le fichier ligne par ligne jusqu'à atteindre une situation d'exception qui correspond techniquement à la fin du fichier.

7.2.3.2 Lecture directe d'un fichier binaire : usage de la classe RandomAccessFile

Le code ci-dessous montre un exemple de lecture directe de fichier binaire en utilisant la classe RandomAccessFile.

```
package com.tuto.io;
import java.io.* ;

public class Main {
        public static void main(String[] args) throws IOException   {

                /*Création d'un fichier binaire à partir d'un texte string */
                FileOutputStream fouts=new
FileOutputStream("src/resources/myBinaryFile.dat");
                DataOutputStream douts=new DataOutputStream(fouts);
                douts.writeUTF("Ceci est la première ligne");
                douts.writeUTF("Ceci est la deuxième ligne");
                douts.writeUTF("Ceci est la troisième ligne");
                fouts.close();
                douts.close();

                /*Lecture du fichier binaire précédemment créé */
                RandomAccessFile raf = new
RandomAccessFile("src/resources/myBinaryFile.dat", "r");
                int initialPosition=29;
                int nbCharacters=59;
                byte[] myText = new byte[nbCharacters];
                raf.seek(initialPosition);
                raf.read(myText , 0, nbCharacters);
                System.out.println(new String(myText));
                raf.close();

        }
}
```

Output

```
Ceci est la deuxième ligne Ceci est la troisième ligne
```

Dans cet exemple, nous commençons d'abord par créer un fichier binaire et ajouter trois lignes de texte.

Ensuite, nous lisons ce fichier en utilisant un mode direct en utilisant un objet de la classe RandomAccessFile. Cette lecture se fait en plusieurs étapes. Dans un premier temps, nous créons un objet de type RandomAccessFile pointant sur le fichier binaire à lire. Ensuite, nous définissons la position à partir de laquelle, nous souhaitons commencer la lecture du fichier. Cette position est définie par la variable initialPosition. Nous définissons également le nombre total de caractères que nous souhaitons lire à partir de la position initiale choisie. Le nombre de caractères est défini par la variable nbCharacters. Aussi, nous définissons un

tableau de bytes servant à récupérer les caractères lus. Dans cet exemple, nous avons défini la variable myText. La dimension de ce tableau est supérieure ou égale au nombre de caractères lus. Pour cet exemple, nous avons choisi une dimension égale au nombre de caractères que nous souhaitons lire, c'est-à-dire 59.

Les paramètres étant définis, nous appelons la méthode seek() sur l'objet RandomAccessFile pour pouvoir se positionner à la position souhaitée. Ici, la position choisie est le caractère 29. C'est à partir de cette position que nous allons commencer les lire le contenu du fichier myBinaryFile.dat. Ensuite, nous appelons la méthode read() pour récupérer l'ensemble des caractères situés dans la plage indiquée.

C'est en exécutant ces lignes de code que nous obtenons les deux dernières lignes de texte parmi les trois spécifiées lors de la création du fichier.

7.3 Gestion des flux Sorties (Output streams)

A l'inverse des flux Entrées, les flux Sorties servent à conduire les données vers un système extérieur au programme Java. Tout comme les flux Entrées, les flux de Sorties peuvent être orientés vers un périphérique d'affichage (écran), un fichier, un réseau distant(socket), un espace mémoire, etc.

Un flux Sortie est généralement matérialisé par une instance de l'une des classes dérivées de la classe abstraite OuputStream. Les classes de gestion des flux Sorties les plus couramment utilisées sont: FileOutputStream, DataOutputStream, BufferedOutputStream, ObjectOutputStream, ByteArrayOutputStream, BufferedWriter, Filewriter, PrintWriter, BufferedWriter, CharArrayWriter, FilterOutputStream, etc.

La page suivante présente la documentation de la classe OutputStream ainsi que les classes dérivées couramment utilisées pour la gestion des flux Sorties : https://docs.oracle.com/javase/10/docs/api/java/io/OutputStream.html

L'utilisation d'un flux Sorties passe par quatre principales étapes.

1. **Création du flux Sortie** : : cette étape consiste à instancier un objet de type stream (flux) en utilisant l'une des classes dérivées de la classe abstraite OutputStream dont la liste a été présentée plus haut. En instanciant l'objet flux, on doit toujours préciser le sink vers lequel les données sont orientées. Ex : sortie standard, écran, fichier, réseau distant, mémoire, etc.
2. **Ecriture des données** : cette étape se réalise généralement par l'appel de la méthode write() sur l'objet crée à l'étape 1. A noter toutefois que la méthode à appeler dépend de la classe de flux de Sortie instanciée. Par exemple pour un objet de la classe BufferedWriter, la méthode de lecture est writeLine(). Et pour un objet de type FileWriter, la méthode à appeler est write().
3. **Commiter l'écriture des données** : parfois, après l'écriture dans un sink, il faut appeler une méthode spécifique pour commiter les changements et persister les

données sur le sink. Par exemple, pour un objet de type FileWriter, on appelle la méthode flush() pour persister les données sur le système cible.

4. **Fermeture du flux** : à la fin de l'écriture des données et le commit des changements sur le sink, on ferme la connection en appelant la méthode close() sur l'objet instancié au départ pour ouvrir le flux.

Dans cette section, nous allons présenter quelques cas d'utilisation et d'exploitation des flux Sorties.

7.3.1 Ecriture sur la sortie standard et sur l'écran : les méthodes System.out.print() et Systme.out.println()

Comme on s'en est déjà rendu compte depuis le début de ce document, la méthode la plus fréquemment utilisée pour la sortie standard et à l'écran est la méthode println(). On peut aussi utiliser la méthode print(). Les méthodes println() et print() sont deux méthodes static de la classe System.out qui nécessitent peu de commentaires. La seule remarque qu'on peut faire est que, à chaque appel, la méthode println() affiche le résultat sur une nouvelle ligne dans la console, alors que la méthode print() affiche le résultat sur la même ligne que les appels précédents.

L'exemple ci-dessous illustre l'utilisation des deux méthodes :

```
package com.tuto.io;
public class Main {
        public static void main(String[] args) {
                System.out.print("Ce texte sera affiché dans la sortie
standard");
                System.out.print("Ce texte sera affiché dans la sortie
standard");
                System.out.println("");
                System.out.println("Ce texte sera affiché dans la sortie
standard");
        }
}
```

7.3.2 Ecriture dans un fichier de texte plat : usage de la classe FileWriter

Bien que Java propose plusieurs classes de flux de sortie pour écrire dans un fichier de texte plat, ici nous prenons l'exemple de la classe FileWriter pour écrire un texte dans un fichier plat. C'est la classe équivalente à la classe FileReader que nous avons déjà utilisé pour la lecture de fichier de texte plat dans le cadre de la gestion des flux Entrées. L'exemple ci-dessous illustre l'utilisation de la classe FileWriter pour l'écriture dans un fichier de texte plat.

```
package com.tuto.io;
import java.io.FileWriter;
import java.io.IOException;

public class Main {

	public static void main(String[] args) {

		try {
			// Création de l'objet FileWriter
			FileWriter fw = new
FileWriter("src/resources/monFichier.txt");

			// Ecriture des lignes dans le fichier
			fw.write("Cette phrase sera ajoutée à la première ligne
du fichier.\n");
			fw.write("Cette phrase sera ajoutée à la deuxième ligne
du fichier.\n");
			fw.write("Cette phrase sera ajoutée à la troisième
ligne du fichier.\n");

			//Comiter l'ajout des lignes et fermer l'objet
FileWriter
			fw.flush();
			fw.close();

		}catch (Exception e) {
			System.err.println(e);
		}
	}
}
```

Dans cet exemple, nous ouvrons d'abord un objet FileWriter qui pointe sur le fichier src/resources/monFichier.txt. Nous ajoutons les lignes de texte en appelant la méthode write(). Notons que, par défaut, la méthode write() ajoute les textes les uns à la suite des autres sur la même ligne. Pour pouvoir avoir chaque texte dans une ligne dédiée, nous avons ajouté l'opérateur "\n" qui ajoute un retour à la ligne pour chaque texte.

A la fin des appels successifs de write(), on doit commiter les changements pour matérialiser l'écriture réelle dans le fichier. Pour la classe FileWriter, la méthode de commit est flush(). Après le flush(), on peut maintenant fermer le flux de sortie en appelant la méthode close() sur l'objet FileWriter.

Pour info, il n'est pas nécessaire que le fichier sur lequel pointe le FileWriter soit préalablement créé. Et si le fichier existe, il sera automatiquement écrasé. Ici, le fichier est situé dans le dossier src/resources du code source java. Mais, il peut s'agir de n'importe quel fichier situé dans le classpath[19].

Rappelons que pour lire le contenu du fichier qu'on vient de créer, il suffit d'utiliser la classe FileReader telle que présentée dans la section sur les flux Entrées. Voir le rappel dans le code source ci-dessous.

[19] Nous reviendrons plus tard sur la définition du classpath lors de la section consacrée à l'exécution du code Java.

```
package com.tuto.io;
import java.io.*;

public class Main {

        public static void main(String[] args) {
                try {

                        String ligne ;
                        BufferedReader br = new BufferedReader (new FileReader
("src/resources/monFichier.txt" )) ;
                        do
                        { ligne = br.readLine() ;
                                if (ligne != null) System.out.println (ligne) ;
                        }
                        while (ligne != null) ;
                        br.close () ;

                }catch (Exception e) {
                        System.err.println(e);
                }
        }
}
```

7.3.3 Ecriture dans un fichier binaire : usage de la classe DataOutputStream

L'une des classes les plus couramment utilisées pour écrire dans un fichier binaire est la classe FileOutputStream. Cette classe offre plusieurs méthodes d'écriture dont notamment les méthodes write(), writeUTF(), writeBytes(), etc. L'exemple ci-dessous illustre l'utilisation de la méthode writeUTF() pour ajouter trois lignes de texte à un fichier binaire.

```
package com.tuto.io;
import java.io.* ;

public class Main {
        public static void main(String[] args) throws IOException   {

                /*Création d'un fichier binaire à partir d'un texte string */
                FileOutputStream fouts=new
FileOutputStream("src/resources/myBinaryFile.dat");
                DataOutputStream douts=new DataOutputStream(fouts);
                douts.writeUTF("Ceci est la première ligne");
                douts.writeUTF("Ceci est la deuxième ligne");
                douts.writeUTF("Ceci est la troisième ligne");
                fouts.close();
                douts.close();


        }
}
```

Dans cet exemple, nous créons un fichier binaire nommé myBinaryFile.dat. Ensuite, nous ajoutons trois lignes de texte en utilisant la méthode writeUTF() de l'objet DataOutputStream. Cette méthode est appelée à chaque fois qu'on a besoin d'ajouter une nouvelle valeur de texte au fichier.

On peut lire le contenu du fichier créé en utilisant l'une des méthodes de lecture des fichiers binaires présentées dans la section concernant les flux Entrées. Le bout de code ci-dessous montre la lecture séquentielle du fichier binaire précédemment créé. Le fichier étant alimenté avec la méthode writeUTF(), nous avons utilisé la méthode readUTF() pour lire le contenu.

```
package com.tuto.io;
import java.io.* ;

public class Main {
        public static void main(String[] args) throws IOException   {

                /*Lecture du fichier binaire précédemment créé */
                DataInputStream dis = new DataInputStream( new FileInputStream
("src/resources/myBinaryFile.dat")) ;
                boolean eof = false ;
                String line=null;
                while (!eof)  {
                        try {
                                line = dis.readUTF () ;
                        }
                catch (EOFException e) {
                                eof = true ;
                }
                        if (!eof) System.out.println (line) ;
                }
                dis.close () ;

        }
}
```

8 GESTION DES FICHIERS ET DES RÉPERTOIRES: USAGE DU PACKAGE java.nio.file

8.1 Présentation de la classe java.nio.file.Files et les classes complémentaires

La classe **java.nio.file.Files** est la classe dédiée à la gestion des fichiers et des répertoires en Java. Elle offre de nombreuses méthodes permettant de réaliser des opérations comme créer et supprimer des fichiers et des répertoires, lire et écrire dans des fichiers, tester l'existence d'un fichier ou d'un répertoire, etc..

Avant JDK 7, la classe java.io.File était la seule classe dédiée à la gestion des fichiers et des répertoires. Mais depuis la JDK 7, cette classe cède peu à peu la place à une nouvelle classe java.nio.file.Files. Cependant, bien que la classe java.io.File reste encore utilisée dans certains programmes, est plus recommandée d'utiliser la classe java.nio.file.Files. Car elle offre non seulement toutes les fonctionnalités pour gérer les fichiers et les répertoires. Elle apporte également de nombreuses améliorations face aux nombreuses limites de la classe java.io.File. Le package java.nio.file offre aussi de nombreuses autres classes complémentaires à la classe Files permettant la gestion des fichiers. Il s'agit notamment des classes :

- **java.nio.file.Path** : une interface offrant un cadre unifié pour la gestion des fichiers et des répertoires. En efffet, un objet de type Path resprésente un chemin d'accès à l'information. Ce chemin d'accès peut être soit un fichier soit un répertoire. La classe Path sert à représenter le chemin d'accès sous forme d'un objet Java.
- **java.nio.file.Paths**: une implémentation de l'interface java.nio.file.Path permettant de créer des objets de type Path et offrant des méthodes spécifiques pour la gestion de ces Paths (fichiers et répertoires).
- **FileSystem** : une classe abstraite permettant de gérer le FileSystem courant, c'est à dire le FileSystem sur lequel le code Java est exécuté.

Comme nous allons le voir ci-dessous, dans une opération de traitement de fichiers et de répertoires, ces classes complémentaires sont utilisées en combinaison avec la classe java.nio.file.Files afin de réaliser l'opération souhaitée.

8.2 Création d'un objet de type chemin d'accès (Path)

Un Path représente un chemin d'accès vers une information. Dans un FileSystem donné, il peut s'agir d'un fichier ou d'un répertoire. Java offre la possibilité de représenter le chemin d'accès sous forme d'une classe Path et offre la possibilité de créer des objets de type Path. La classe java.nio.file.Paths est la classe dédiée à la création des objets de type Path. Mais il existe aussi une classe autonome FileSystm permet qui indirectement de créer des objets de type Path.

A noter que la création d'un objet de type Path est la base de toute opération de gestion de fichiers et de répertoire dans un programme Java. La création de Path est parfois implicitement faite par certaines classes lorsque par exemple le chemin d'accès est spécifié sous forme de String. Mais dans le cadre l'usage de la classe java.nio.file.Files, la création de l'objet est parfois un préalable et qui doit être explicitement faite.

Cette section a pour but de montrer la création des objets de type Path en utilisant l'une des deux classes. Ci-dessous les exemples d'illustration des deux approches.

8.2.1 Création d'un objet Path : usage de la classe java.nio.file.Paths

Une première manière de créer un objet de type Path est l'utilisation de la classe java.nio.file.Paths à travers l'appel de la méthode get(). L'exemple ci-dessous illustre la création.

```
package com.tuto.io;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {
        public static void main(String[] args)   {

                Path path = Paths.get("/src/resources/myFolder");
                System.out.println(path.getFileName());
                System.out.println(path.getName(0));
                System.out.println(path.getName(1));
                System.out.println(path.getName(2));
        }
}
```

Output

```
myFolder
src
resources
myFolder
```

Dans cet exemple, nous avons créé l'objet Path nommé path en appelant la méthode get() de la classe Paths. Comme on peut le remarquer la classe Paths est une classe static. C'est-à-dire qu'elle n'est pas instanciée au préalable avant de pouvoir utiliser ses méthodes. Dans l'exemple ci-dessus, nous appelons directement la méthode get() auquel nous passons le path sous forme de String qui est alors automatiquement converti en type Path. A noter que le chemin spécifié peut aussi bien être un fichier (ou un répertoire). A ce stade, peut importe que ce Path soit déjà créé ou non. Car l'objet Path est simplement une référence, qui peut être matérialisé ou non sur le FileSystm. C'est l'usage qui sera fait du Path plus tard qui déterminera s'il s'agit d'un fichier réel ou d'un répertoire réel et si ce fichier ou ce répertoire est déjà créé ou non. Nous reviendrons plus tard sur tous ces détails.

L'objet de type Path offre plusieurs méthodes permettant de manipuler l'objet correspondant au chemin d'accès défini. Dans l'exemple ci-dessous, nous avons appelé la méthode getFileName() qui permet de renvoyer le dernier élément constituant le chemin qu'il s'agit d'un fichier ou d'un répertoire. Nous avons également appelé la méthode getName() qui permet de renvoyer un élément du chemin d'accès en se basant sur sa position. Ici, le séparateur du FileSystem étant le symbôle « / », la position 0 dans le Path correspond à src, la position 1 à *resources* et la position 2 à *myFolder*.

Pour plus de détails sur l'interface Path ainsi que ses méthodes d'exploitation, consulter la page suivante : https://docs.oracle.com/javase/10/docs/api/java/nio/file/Path.html

8.2.2 Création d'un objet Path : usage des méthodes de la classe FileSystem

La classe FileSystem est une classe autonome du package java.nio.file. Néanmoins, elle offre des méthodes pour créer des objets de type Path. C'est pour cette raison que nous avons décidé de la présenter dans cette section à titre illustratif.

La création d'un objet de type Path à partir de la classe FileSystem se fait en appelant successivement les deux méthodes : getDefault() et getPath(). Voir exemple ci-dessous.

```
package com.tuto.io;
import java.nio.file.FileSystems;
import java.nio.file.Path;

public class Main {
        public static void main(String[] args)   {

                Path path =
FileSystems.getDefault().getPath("/src/resources/myFolder");
                System.out.println(path.getFileName());
                System.out.println(path.getName(0));
                System.out.println(path.getName(1));
                System.out.println(path.getName(2));
        }
}
```

Output

```
myFolder
src
resources
myFolder
```

On obient ainsi un objet qui a les mêmes caractéristiques que celui créé avec la méthode get() de la classe Paths présentée dans la section précédente.

Rappelons que dans les deux exemples de création de Path, nous avons pointé le chemin d'accès /src/resources/myFolder. Mais il pouvait aussi s'agir de n'importe que chemin visible depuis le classpath.

8.3 Créer un répertoire vide : usage de la méthode createDirectory() ou createDirectories()

Pour créer un répertoire vide à partir d'un objet de type Path, on appelle la méthode createDirectory(). Cette méthode est l'équivalente de la commande shell mkdir. Et dans le cas où il s'agit de créer un répertoire ainsi que ses arborescences parentes, on utilise la méthode createDirectories(). L'exemple ci-dessous montre l'utilisation des deux méthodes de création de répertoires vides.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path newDirectory =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder");
                Files.createDirectory(newDirectory);
                Path directories =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder1\\myFolder2
");
                Files.createDirectories(directories);

        }
}
```

NB : Cet exemple étant exécuté sous le système d'exploitation Windows, les séparateurs d'arborescence sont d'abord représentés dans le code source par le symbole « \\ » contrairement au Système Unix dont le séparateur est « / ».

8.4 Créer un fichier vide : usage de la méthode createFile()

Pour créer un fichier vide, on appelle la méthode createFile() sur un objet de type Path. L'exemple ci-dessous montre la création d'un fichier vide.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path newFile =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile.txt
");
                Files.createFile(newFile);

        }
}
```

NB : Cet exemple étant exécuté sous Windows, les séparateurs d'arborescence sont représentés par le symbole « \\ » contrairement au Système Unix dont le séparateur est « / ».

8.5 Supprimer un répertoire : usage de la méthode delete() ou deleteIfExists()

La méthode delete() ou deleteIfExists() permet de supprimer un répertoire. La particularité de la méthode deleteIfExists() par rapport à la méthode delete() est que la méthode delete() renvoie une exception lorsque le répertoire n'existe pas alors que la méthode deleteIfExists() ne renvoie pas d'exception. L'exemple ci-dessous montre l'utilisation des deux méthodes.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path dir1 =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder1");
                Files.delete(dir1);
                Path dir2 =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder2");
                Files.deleteIfExists(dir1);
        }
}
```

NB : Cet exemple étant exécuté sous Windows, les séparateurs d'arborescence sont représentés par le symbole « \\ » contrairement au Système Unix dont le séparateur est « / ».

Attention, lorsque le répertoire à supprimer contient des fichiers ou des sous-répertoires, la méthode delete() et deleteIfExists() renvoient une exception de type DirectoryNotEmptyException. Dans ce cas, il faut d'abord supprimer les fichiers et les dossiers qu'il contient avant de lancer la suppression du répertoire (voir ci-dessous pour les méthodes de suppression de fichiers).

8.6 Supprimer un fichier : usage de la méthode delete() ou deleteIfExists()

La méthode delete() ou deleteIfExists() permet de supprimer aussi bien un fichier qu'un répertoire. Comme déjà indiqué plus haut, la méthode delete() renvoie une exception lorsque le fichier n'existe pas alors que la méthode deleteIfExists() ne renvoie pas d'exception. L'exemple ci-dessous montre l'utilisation des deux méthodes.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path f1 =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\monFichier.txt");
                Files.delete(f1);
                Path f2 =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\monFichier.log");
                Files.deleteIfExists(f2);
        }
}
```

NB : Cet exemple étant exécuté sous Windows, les séparateurs d'arborescence sont représentés par le symbole « \\ » contrairement au Système Unix dont le séparateur est « / ».

8.7 Tester si un fichier ou un répertoire existe : la méthode exists()

La méthode exists de la classe Files permet de tester si un Path existe ou pas qu'il s'agisse d'un répertoire ou d'un fichier. L'exemple ci-dessous illustre l'appel de la méthode exists().

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path myFile=
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile.txt
");
                boolean t1=Files.exists(myFile);
                System.out.println(myFile.getFileName()+" existe: "+t1);
                Path myDir=
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\mySubFolde
r44");
                boolean t2=Files.exists(myDir);
                System.out.println(myDir.getFileName()+" existe: "+t2);

        }
}
```

Output :

```
myFile.txt existe: true
mySubFolder44 existe: false
```

8.8 Tester si un Path est un répertoire ou un fichier : les méthodes isDirectory() et isRegularFile()

La classe java.nio.file.Files offre deux méthodes permettant de tester si un Path est un répertoire ou un fichier régulier. Il s'agit respectivement de la méthode isDirectory() et isRegularFile(). L'exemple ci-dessous illustre l'utilisation des deux méthodes pour tester la nature d'un Path.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path path1=
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile.txt
");
                boolean t1=Files.isDirectory(path1);
                System.out.println(path1.getFileName()+" est un dossier: "+t1);
                boolean t2=Files.isRegularFile(path1);
                System.out.println(path1.getFileName()+ " est un fichier:
"+t2);
                Path path2=
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\mySubFolde
r");
                boolean t3=Files.isDirectory(path2);
                System.out.println(path2.getFileName()+" est un dossier: "+t3);
                boolean t4=Files.isRegularFile(path2);
                System.out.println(path2.getFileName()+" est un fichier: "+t4);

        }
}
```

Output :

```
myFile.txt est un dossier: false
myFile.txt est un fichier: true
mySubFolder est un dossier: true
mySubFolder est un fichier: false
```

Dans cet exemple, le test porte sur un Path dont la cible est myFile.txt (qui est en réalité un fichier) et mySubFolder qui est un sous-répertoire (en fait un répertoire).

8.9 Récupérer et lister tous les éléments présents dans un répertoire : la méthode newDirectoryStream()

La méthode newDirectoryStream() de la classe Files permet de lister l'ensemble des éléments présents dans un répertoire spécifié sous forme de Path. Un répertoire peut contenir trois types d'objets : les fichiers, les sous-répertoires ou les liens symboliques. L'exemple ci-dessous liste l'ensemble des éléments présents dans le répertoire myFolder.

Pour info, nous avons préalablement créé dans ce dossier un fichier nommé « myFile.txt et un sous répertoire nommé « mySubFolder ». Voir l'exemple ci-dessous.

```
package com.tuto.io;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path dir =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder");
                try (
                        DirectoryStream<Path> elements =
Files.newDirectoryStream(dir))
                {
                        for (Path element : elements) {
                                System.out.println(element.getFileName());
                        }
                } catch (IOException e) {
                        e.printStackTrace();

        }
}
}
```

Output :

```
myFile.txt
mySubFolder
```

Dans cet exemple, nous avons appelé la méthode newDirectoryStream sur un objet de type Path pour lister et afficher l'ensemble des éléments qu'il contient. Nous constatons que le dossier myFolder contient deux éléments : myFile.txt et mySubFolder. Toutefois, un simple println() sur chaque élément ne permet pas de savoir s'il s'agit d'un fichier ou d'un répertoire. C'est pourquoi, on aurait pu utiliser en plus les méthodes isRegularFile() et isDirectory() pour tester le type de chaque élément. Ainsi, le code pourra être réécrit comme suit :

```
package com.tuto.io;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                Path dir =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder");
                try (
                        DirectoryStream<Path> elements =
Files.newDirectoryStream(dir))
                {
```

```
                        for (Path element : elements) {
                                if (Files.isDirectory(element)){

System.out.println(element.getFileName()+ " est un répertoire");
                                } else if (Files.isRegularFile(element)){

System.out.println(element.getFileName()+ " est un fichier");
                                }
                                else

System.out.println(element.getFileName()+ " n'est ni un répertoire, ni un
fichier");
                        }
                } catch (IOException e) {
                        e.printStackTrace();

                }
        }
}
```

Output :

```
myFile.txt est un fichier
mySubFolder est un répertoire
```

8.10 Ecriture et lecture d'un fichier : usage de la méthode write() et readAllLines()

La méthode write() permet d'écrire une liste de texte dans un fichier tandis que la méthode readAllLines() permet lire tout le contenu d'un fichier et renvoie le résultat sous forme d'une liste de texte. L'exemple ci-dessous illustre l'utilisation des deux méthodes.

```
package com.tuto.io;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Main {
        public static void main(String[] args) throws IOException {

                Path textFile
=Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile.tx
t");

                List<String> textLines= new ArrayList<String>();
                textLines.add("Ceci est la première ligne");
                textLines.add("Ceci est la deuxième ligne");
                textLines.add("Ceci est la troisième ligne");
                try {
                        Files.write(textFile, textLines);
                } catch (IOException ex) {
                        ex.printStackTrace();
                }
```

```
                // Lecture du fichier alimenté
                List<String> lines = null;
                try {
                        lines = Files.readAllLines(textFile);
                } catch (IOException ex) {
                        ex.printStackTrace();
                }
                if (lines != null) {
                        for (String line : lines) {
                                System.out.println(line);
                        }
                }
        }
}
```

Output :

```
Ceci est la première ligne
Ceci est la deuxième ligne
Ceci est la troisième ligne
```

Dans cet exemple, nous disposons au préalable d'un fichier vide nommé myFile.txt. Nous construisons d'abord une liste contenant les lignes de textes. En l'occurrence, nous créons trois lignes de textes. Ensuite, nous utilisons la méthode write() de la classe Files pour écrire cette liste de texte dans le fichier myFile.txt.

Par ailleurs, pour assurer de la bonne écriture des lignes, nous lisons le contenu du fichier en utilisant la méthode readAllLines() et nous faisons une boucle pour afficher chaque ligne lue.

8.11 Copier un fichier ou un répertoire : la méthode copy()

La méthode copy() de la classe Files permet de faire la copie d'un fichier ou d'un répertoire exprimé sous forme d'un objet Path. L'exemple ci-dessous crée la copie d'un fichier nommé myFile.txt et la copie du répertoire nommé myFolder.

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.nio.file.StandardCopyOption;

public class Main {
        public static void main(String[] args) throws IOException {


                // Copier un fichier
                Path src_file =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile.txt
");
                Path dest_file =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\myFile_cop
y.txt");
```

```
                try {
                        Files.copy(src_file, dest_file,
StandardCopyOption.REPLACE_EXISTING);
                } catch (IOException e) {
                        e.printStackTrace();
                }
                // Copier un répertoire
                Path src_dir=
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder");
                Path dest_dir =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder_copy");
                try {
                        Files.copy(src_dir, dest_dir,
StandardCopyOption.REPLACE_EXISTING);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

La méthode copy() propose plusieurs options à travers l'Enum StandardCopyOption. Dans cet exemple, nous utilisons l'option REPLACE_EXISTING qui signifie que le fichier de destination doit être écrasé s'il existe déjà. Dans l'exemple, nous faisons la copie du fichier myFile.text nommé myFile_copy.txt. Nous faisons également la copie du répertoire myFolder nommé myFolder_copy.

A noter que la méthode copy() ne copie pas les fichiers et les sous-répertoires contenus dans le répertoire copié. Lorsque le répertoire contient des fichiers et des répertoires, on peut penser à élaborer une moulinette récursive qui liste l'ensemble des éléments présents dans le répertoire. Pour cela, on peut se baser sur la méthode newDirectoryStream() que nous avons déjà utilisée. Et parmi les éléments listés, lorsqu'il s'agit d'un fichier, on lance la méthode copy(). Et lorsque l'élément est un répertoire, on relance de manière récursive la méthode newDirectoryStream(). Et ainsi de suite, jusqu'à la copie complète du contenu du répertoire à copier.

8.12 Déplacer un fichier ou un répertoire : la méthode move()

La méthode move() permet de déplacer un fichier ou un répertoire dans le FileSystem. L'exemple ci-dessous illustre l'utilisation de la méthode move().

```
package com.tuto.io;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.nio.file.StandardCopyOption;

public class Main {
        public static void main(String[] args) throws IOException {

                // Déplacer un fichier
                Path src_file =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\folder1\\myFile.txt"
```

```
);
                Path dest_file =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\folder2\\myFile.txt"
);
                try {

Files.move(src_file,dest_file,StandardCopyOption.REPLACE_EXISTING);
                } catch (IOException e) {
                        e.printStackTrace();
                }

                // Déplacer un répertoire
                Path src_dir =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\folder1");
                Path dest_dir =
Paths.get("C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\myFolder\\folder1");
                try {

Files.move(src_dir,dest_dir,StandardCopyOption.REPLACE_EXISTING);
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
}
```

Dans cet exemple, nous utilisons la méthode move() pour déplacer le fichier myFile.text du dosssier folder1 vers le dossier folder2. Ensuite, nous utilisons la méthode move() pour déplacer le dossier folder1 vers le dossier myFolder. Dans chacun des deux cas, nous utilisons l'option REPLACE_EXISTING qui remplace l'objet de destination s'il existe déjà.

A noter que contrairement à la méthode copy(), la méthode move() déplace un dossier et l'ensemble de son contenu (qu'il s'agisse des fichiers ou des sous-répertoires).

9 LES EXPRESSIONS RÉGULIÈRES

9.1 Généralités

Les expressions régulières en abrégé regex ou regexp sont des formalismes permettant le traitement et la manipulation de chaînes de caractères. Les regex sont universelles en ce sens qu'elles sont utilisées autant en Java que tout autre langage de programmation et cela avec les mêmes syntaxes. Les regex ont plusieurs usages dans un programme : recherche et reconnaissance de motifs (patterns) dans une chaîne de caractères, recherche de mots-clés dans un corps de texte, analyses de texte, définition de conditions de filtrage de lignes dans une base de données, contrôles et validations de valeurs et de formats, etc.

Dans le langage Java, les expressions régulières sont disponibles dans une API spécifique en l'occurrence java.util.regex. Ce chapitre vise à présenter l'utilisation des classes disponibles dans ce package dans les opérations de traitements de chaînes de caractères se présentant sous formes de texte.

9.2 Les principales classes de traitement de regex en Java: la classe Pattern et la classe Matcher

9.2.1 Présentation des classes Pattern et Matcher et leurs principales méthodes

Le package java.util.regex offre deux principales classes permettant le traitement des regex. Il s'agit en l'occurrence de la classe Pattern et de la classe Matcher. La classe Pattern permet d'avoir une représentation compilée du motif (pattern) sous forme d'un objet. La classe Pattern est une classe static, c'est-à-dire qu'on l'utilise sans avoir à l'instancier en utilisant l'opérateur new. Il suffit simplement d'importer la classe depuis le package java.util.regex et d'appeler ses méthodes. La classe Pattern dispose d'une méthode de base nommée compile() qui permet créer un objet regex à partir d'un motif préalablement indiqué. A noter que le motif représente le formalise traduisant l'expressions régulière. La classe Pattern dispose également de plusieurs méthodes qui permettent de manipuler l'objet regex préalablement créé par la méthode compile(). Les méthodes les plus utilisées sont :

- matcher() : permet de créer un objet Matcher à partir d'une chaîne de caractères spécifiée en argument. C'est contre cette chaîne de caractères qu'est testé l'objet regex obtenu du motif.
- matches() : permet de tester si le pattern (motif) matche complètement avec la chaîne de caractères fournie.
- lookingAt () : permet de tester si le pattern (motif) matche au moins partiellement avec la chaîne de caractères fournie. A la différence de la méthode matches() qui renvoie true uniquement lorsque le motif matche complètement la chaîne de caractères, la méthode lookingAt() renvoie true lorsque le motif se trouve dans la chaîne de caractères. De ce point de vue, la méthode lookingAt() se comporte comme

la méthode contains() de la classe String. La méthode matches() est donc plus restrictive que la méthode lookingAt(). Le choix entre les deux méthodes dépendra donc du contexte d'utilisation.

- split() : découpe la chaîne de caractères suivant le motif spécifié
- pattern () : renvoie le motif à partir duquel l'objet regex a été construit.
- …
- Etc.

Quant à la classe Matcher, elle permet d'avoir une représentation compilée de la chaîne de caractères dans laquelle le motif sera recherchée. L'usage de la classe Matcher va de pair avec l'usage de la classe Pattern. D'ailleurs, la classe Pattern dispose d'une méthode nommée matcher() qui renvoie directement un objet de type Matcher sans avoir besoin d'appeler la classe Matcher. Les principales méthodes de la classe Matcher sont :

- matches() : permet de tester si le motif matche complètement la chaîne de caractères compilée.
- lookingAt () : permet de tester si le motif matche partiellement la chaîne de caractères compilée. A la différence de la méthode matches() qui renvoie true uniquement lorsque le motif matche complètement la chaîne de caractères, la méthode lookingAt() renvoie true lorsque le motif se trouve dans la chaîne de caractères. Elle se comporte donc comme la méthode contains() de la classe String. La méthode matches() est donc plus restrictive que la méthode lookingAt(). Encore une fois, le choix entre les deux méthodes dépendra donc du contexte d'utilisation.
- find() : renvoie une valeur booléenne true si le motif se trouve dans la chaîne de caractères ou false sinon. La méthode find() est souvent appelée de manière itérative. En effet lorsqu'une chaîne de caractères contient plusieurs matches d'un motif, chaque appel de la méthode find() se positionne sur la prochaine occurrence de match trouvé
- start() : renvoie la position (l'indice) du premier caractère du bout de chaîne auquel matche le motif dans la chaîne de caractères.
- end() : renvoie la position (l'indice) du dernier caractère du bout de chaîne auquel matche le motif dans la chaîne de caractères.
- group() : renvoie le morceau de chaînes de caractères qui match avec le motif. Lorsqu'une chaîne de caractères contient plusieurs occurrences qui matchent avec le motif. L'appel de la méthode group() suite à l'appel de la méthode find() renvoie l'occurrence qui matche avec le motif pour cet appel correspondante.
- groupCount() : renvoie le nombre total de bouts de chaînes qui matchent le motif dans la chaîne de caractères.
- replaceFirst() : remplace la première occurrence du motif par une chaîne de caractères fournie en paramètre de la méthode.

- replaceAll() : remplace toutes les occurrences du motif par une chaîne de caractères fournie en paramètre de la méthode.
- ...
- Etc

Pour plus de détails sur les classes Pattern et Matcher ainsi que leurs méthodes, consulter les pages ci-dessous:

https://docs.oracle.com/javase/10/docs/api/java/util/regex/Pattern.html

https://docs.oracle.com/javase/10/docs/api/java/util/regex/Matcher.html

9.2.2 Usage des méthodes compile(), matcher() et matches()

L'exemple ci-après montre un cas simple d'utilisation des méthodes compile(), matcher() et matches() des classes Pattern et Matcher. Nous reviendrons plus tard sur les cas les plus complexes.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Création d'un objet regex simple à partir d'un motif
        String motif=".*est.*";
        Pattern pattern = Pattern.compile(motif); // méthode compile()
        //Pattern pattern = Pattern.compile(motif, Pattern.CASE_INSENSITIVE) //
autre façon d'appeler la méthode compile();

        // Créer un objet matcher à partir d'une chaîne de caractères 1
        String cdc1="Elle est allée";
        Matcher matcher1 = pattern.matcher(cdc1); // Méthode matcher()
        // Tester si la chaîne de caractères 1 matche avec le motif
        boolean bool1 = matcher1.matches(); // méthode matches()
        // Voir si la chaîne de caractères 1 matche avec le motif
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        // Créer un matcher à partir d'une chaîne de caractères 2
        String cdc2="La ville se trouve à l'est du pays";
        Matcher matcher2 = pattern.matcher(cdc2); // Méthode matcher()
        // Tester si la chaîne de caractères 2 matche avec le motif
        boolean bool2 = matcher2.matches(); // méthode matches()
        // Voir si la chaîne de caractères 2 matche avec le motif
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
```

```
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        // Créer un matcher à partir d'une chaîne de caractères 3
        String cdc3="Centre universitaire";
        Matcher matcher3 = pattern.matcher(cdc3); // Méthode matcher()
        // Tester si la chaîne de caractères 3 matche avec le motif
        boolean bool3 = matcher3.matches(); // méthode matches()
        // Voir si la chaîne de caractères 3 matche avec le motif
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");

    }
}
```

Output :

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans l'exemple ci-dessous, nous avons d'abord défini un motif dont la valeur « .*est.* ». Ce motif sert à matcher toute chaîne de caractères contenant le mot « est » avec la possibilité qu'il y ait d'autres caractères à gauche ou à droite. En effet comme nous le verrons plus tard l'expression « .*» signifie n'importe quelle chaîne de caractères quelle que soit sa longueur. Le fait de spécifier cet opérateur à gauche et à droite du mot « est » signifie qu'on peut matcher n'importe quelle chaîne contenant le mot « est ».

Dans l'exemple ci-dessus, nous illustrons l'usage des méthodes compile(), matcher() et matche(), en testant le motif « .*est.* » contre trois chaînes de caractères différentes. La première chaine de caractères dans laquelle nous recherchons le motif est « Elle est allée ». La seconde est « La ville se trouve à l'est du pays ». Et la troisième est « Centre universitaire ».

Avant tout, il faut d'abord compiler le motif ".*est.*" en un objet de type regex en appelant la méthode compile(). Et pour chacune des trois chaines de caractères cdc1, cdc2 et cdc3, nous appelons d'abord la méthode matcher() sur l'objet initialement créé. La méthode matcher() permet de compiler la chaîne de caractères sous forme d'objet de type Matcher. Cette compilation permet ainsi de représenter la chaîne de caractères sous une forme de sorte qu'on puisse rechercher le motif déjà compilé sous forme d'objet de type Pattern. En fait, tout ceci signifie que pour rechercher un motif dans une chaîne de caractères, le motif doit d'abord être compilé en un objet de type Pattern en utilisant la méthode compile(). Ensuite, la chaîne de caractères doit être compilée en un objet de type Matcher en appelant la méthode matcher() sur l'objet de type Pattern préalablement créé.

Les objets Pattern et Matcher étant créés, nous appelons la méthode matches() pour rechercher le motif dans la chaîne de caractères fournie. La méthode matches() renvoie une valeur booléenne qui est true lorsque le motif matche avec la chaîne de caractères et false sinon. Le motif ".*est.*" matche avec les deux premières chaînes de caractères à savoir « Elle est allée » et « La ville se trouve à l'est du pays ». En revanche, le motif ne matche pas avec la troisième chaîne de caractère « Centre universitaire ».

Cet exemple introductif visait à présenter les principales classes et méthodes de gestion des expressions régulières Java. Le reste du chapitre sera consacré à l'usage de ces classes et méthodes pour les opérations d'expressions régulières.

9.2.3 Usage des méthode find(), start() et end()

La méthode find() est un curseur qui, lors de chaque appel, se positionne sur la prochaine occurrence du motif recherché. La méthode renvoie une valeur booléenne true si le motif est trouvé et false sinon. La méthode find() est généralement appelée dans une boucle while pour parcourir la chaîne de caractère et retrouver toutes les occurrences du motif spécifié. En complément de la méthode find(), nous pouvons aussi appeler les méthodes start() et end() sur l'objet Matcher. Ces deux méthodes renvoient respectivement les indices (positions) de début et de fin de chaque occurrence du motif dans la chaîne de caractères représentée par l'objet Matcher. L'exemple ci-dessous montre l'usage de la méthode find() pour retrouver toutes les occurrences du motif « est » dans la chaîne de caractères «Elle est allée à l'est du pays » . Nous utilisons également les méthodes start() et end() pour identifier les positions de chaque occurrence dans la chaîne de caractères principale.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Création d'un objet regex à partir d'un motif
        String motif="est";
        Pattern pattern = Pattern.compile(motif);

        // Créer un matcher à partir d'une chaîne de caractères
        String cdc="Elle est allée à l'est du pays";
        Matcher matcher = pattern.matcher(cdc);

        // Chercher le motif dans la chaîne de caractères cdc et afficher ses
positions de début et de fin
        while (matcher.find()){
            System.out.println("Indice début: " + matcher.start()+" Indice fin:
" + matcher.end());
        }

    }
}
```

Output

```
Indice début: 5 Indice fin: 8
Indice début: 19 Indice fin: 22
```

Dans cet exemple, nous définissons d'abord un objet Pattern à partir de la chaîne de caractères « est ». Ensuite nous définissons un objet Matcher à partir de la chaîne de caractères « Elle est allée à l'est du pays ». Pour vérifier si le motif « est » se trouve bien dans la chaîne de caractères cdc, nous appelons la méthode find() de l'objet Matcher. Cet appel est fait dans une boucle while afin de retrouver toutes les occurrences du motif.

S'agissant de l'utilisation des méthodes start() et end(), nous remarquons deux occurrences du motif « est ». La première occurrence se trouve entre la position 5 (inclus) et la position 8 (exclue) dans la chaîne de caractères. Et la deuxième occurrence se trouve entre les positions 19 et 22.

9.2.4 Usage de la méthode group()

La méthode group() permet de renvoyer le bout de chaîne correspondant à chaque groupe de caractères formant le motif lorsque ce motif matche avec la chaîne de caractères initiale. Le groupe de caractères est un assemblage de plusieurs caractères pour former un bloc compact. Ce bloc est considéré et traité comme un élément distinct formant un tout, au-delà des caractères individuels qui le compose. En langage regex, le groupe de caractères est défini par l'opérateur de groupage « () ». Nous reviendrons plus amplement sur l'utilisation de l'opérateur de groupage « () ». Mais à titre illustratif, lorsqu'on spécifie par exemple un bout de chaîne « (est) », on définit un groupe de caractères formé du bloc « abc » et non les caractères individuels « a » ou « b » ou « c ».

A noter qu'un motif peut contenir un ou plusieurs groupes de caractères. Ces groupes peuvent être définis soit de manière disjointe soit de manière imbriquée. Par exemples l'expression (A)(B)(C) est un exemple de groupes disjoints. Tandis que l'expression « ((A)(B(C))) » représente un cas de groupes imbriqués. Le nombre de groupes est identifiable par le nombre de parenthèses ouvrantes. Par exemple, l'expression ((A)(B(C))) contient quatre groupes : ((A)(B(C))) , (A), (B(C)) et (C). Le premier groupe correspond toujours à l'expression initiale toute entière. Chaque autre groupe correspond à une parenthèse complète et fermée distincte. Comme on peut le remarquer, un groupe peut contenir un ou plusieurs autre(s) groupe(s). D'où l'usage du terme groupes imbriqués. Le nombre de groupes d'une expression peut être automatiquement obtenu en utilisant la méthode groupCount(). Et pour renvoyer le bout de chaîne correspondant à un groupe donné, il suffit d'appeler la méthode group(i) où i correspond à l'indice dans la liste des groupes renvoyés. Par exemple, dans l'expression ((A)(B(C))), le groupe ((A)(B(C))) a pour indice 0, le groupe (A) a pour indice 1, le groupe (B(C)) a pour indice 2 et (C) a pour indice 3. Ainsi pour récupérer le groupe (A), il suffit d'appeler la méthode group(1). L'exemple ci-dessous illustre l'appel de la méthode group().

Le code ci-dessous montre quelques exemples d'utilisation de l'opérateur de groupage () et de l'appel de la méthode group() pour renvoyer le contenu de chaque groupe formant le motif lorsque celui-ci matche avec la chaîne initiale.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {

        // Création d'un objet regex à partir d'un motif
        String motif1="(A)(B)(C)"; //Cas de groupes disjoints
        Pattern pattern1 = Pattern.compile(motif1);
        // Créer un objet matcher à partir d'une chaîne de caractères 1
        String cdc1="ABC";
        Matcher matcher1 = pattern1.matcher(cdc1); // Méthode matcher()
        // Tester si la chaîne de caractères 1 matche avec le motif
        boolean bool1 = matcher1.matches(); // méthode matches()
        // Voir si la chaîne de caractères 1 matche avec le motif
        if (bool1==true) {
            System.out.println("Motif1: "+motif1);
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif1");
            System.out.println("Les groupes de cdc1 sont");
            int nb_grp=matcher1.groupCount();
            int i=0;
            while (i<=nb_grp) {
                String grp=matcher1.group(i);
                System.out.println("Group " + i + ": " + grp);
                i++;
            }
        }
        // Création d'un objet regex à partir d'un motif
        String motif2="((A)(B(C)))"; //Cas de groupes imbriqués
        Pattern pattern2 = Pattern.compile(motif2);
        // Créer un objet matcher à partir d'une chaîne de caractères 2
        String cdc2="ABC";
        Matcher matcher2 = pattern2.matcher(cdc2); // Méthode matcher()
        // Tester si la chaîne de caractères 2 matche avec le motif
        boolean bool2 = matcher2.matches(); // méthode matches()
        // Voir si la chaîne de caractères 2 matche avec le motif
        if (bool2==true) {
            System.out.println("Motif2: "+motif2);
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif2");
            System.out.println("Les groupes de cdc2 sont");
            int nb_grp=matcher2.groupCount();
            int i=0;
            while (i<=nb_grp) {
                String grp=matcher2.group(i);
                System.out.println("Group " + i + ": " + grp);
                i++;
            }
        }
        // Création d'un objet regex à partir d'un motif
        String motif3="(2023)-(05)-(28)"; //Cas de groupes disjoints séparés
```

```
par des caractères supplémentaires
        Pattern pattern3 = Pattern.compile(motif3);
        // Créer un objet matcher à partir d'une chaîne de caractères 3
        String cdc3="2023-05-28";
        Matcher matcher3 = pattern3.matcher(cdc3); // Méthode matcher()
        // Tester si la chaîne de caractères 3 matche avec le motif
        boolean bool3 = matcher3.matches(); // méthode matches()
        // Voir si la chaîne de caractères 3 matche avec le motif
        if (bool3==true) {
            System.out.println("Motif3: "+motif3);
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif3");
            System.out.println("Les groupes de cdc3 sont");
            int nb_grp=matcher3.groupCount();
            int i=0;
            while (i<=nb_grp) {
                String grp=matcher3.group(i);
                System.out.println("Group " + i + ": " + grp);
                i++;
            }
        }
    }
}
```

Output :

```
Motif1: (A)(B)(C)
La chaîne de caractères cdc1 matche avec le motif1
Les groupes de cdc1 sont
Group 0: ABC
Group 1: A
Group 2: B
Group 3: C
Motif2: ((A)(B(C)))
La chaîne de caractères cdc2 matche avec le motif2
Les groupes de cdc2 sont
Group 0: ABC
Group 1: ABC
Group 2: A
Group 3: BC
Group 4: C
Motif3: (2023)-(05)-(28)
La chaîne de caractères cdc3 matche avec le motif3
Les groupes de cdc3 sont
Group 0: 2023-05-28
Group 1: 2023
Group 2: 05
Group 3: 28
```

Cet exemple présente trois cas d'utilisation de la méthode group(). Chaque cas est défini sur la base d'un motif particulier. D'abord concernant le motif1, il est défini en combinant trois groupes de caractères distincts (A), (B) et (C). Sa forme finale est (A)(B)(C). Et pour qu'une chaîne de caractères matche ce motif, il doit matcher chacun des trois groupes individuellement. Pour réaliser ce test, nous avons défini une chaîne de caractères cdc1 dont la valeur est « ABC ». Nous matchons d'abord cette chaîne contre motif1. Lorsque le match est vérifié, nous appelons la méthode group() sur la chaîne de caractères cdc1 pour renvoyer

le bout de chaîne correspondant à chaque groupe dans motif1. Le renvoi des bouts de chaîne est fait dans une boucle while pour pouvoir avoir l'indice de chaque groupe. Rappelons que le nombre total de groupe est capturé par l'appel de la méthode groupCount(). Connaisant le nombre total de groupes et sachant que le premier groupe a toujours un indice égal à 0, nous faisons une boucle pour récupérer chaque groupe grâce à son indice. Nous appliquons le même principe sur les motifs motif2 et motif3 qui sont testés respectivement contre les chaînes de caractères cdc1 et cdc2. Voir l'output suite à l'exécution du code.

9.3 Les opérateurs regex

On distingue plusieurs type d'opérateur regex : les opérateurs de base, les opérateurs de classe, les opérateurs de quantification, les opérateurs logiques et les opérateurs d'échappement. Cette section a pour but de présenter chaque type d'opérateur ainsi que leur usage dans les opérations regex.

9.3.1 Les opérateurs regex de base : « . », « . *», « ^ » et « $ »

9.3.1.1 L'opérateur « . » : matcher n'importe quel caractère (standard ou spécial)

L'opérateur « . » permet de matcher n'importe quel caractère unique qu'il soit alphabétique, numérique ou un caractère spécial y compris le caractère espace: « », « _ », «-»,«&», «$»,«#», «/»,«\»,«,» , « :», «.»). Les exemples ci-dessous illustrent l'usage de l'opérateur « . » pour matcher n'importe quel type de caractère singulier.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif à partir l'opérateur "."
        String motif=".";
        Pattern pattern = Pattern.compile(motif);

        // Cas où le caractère est un lettre
        String cdc1="M"; // Une lettre
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");
        // Cas où le caractère est un lettre
        String cdc2="MS"; // Deux lettres
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches(); // Ne matchera pas.
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
```

```
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");
        // Cas où le caractère est un chiffre
        String cdc3="9"; // Un chiffre
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");

        // Cas où le caractère est un caractère spéciale: « _ », « _ », «-
»,«&», «$»,«#», «/»,«\»,«,» , « :», «.»
        String cdc4="_"; // Un underscore
        Matcher matcher4 = pattern.matcher(cdc4);
        boolean bool4 = matcher4.matches();
        if (bool4==true)
            System.out.println("La chaîne de caractères cdc4 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc4 ne matche pas avec
le motif");

    }
}
```

Output :

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 ne matche pas avec le motif
La chaîne de caractères cdc3 matche avec le motif
La chaîne de caractères cdc4 matche avec le motif
```

Dans cet exemple, nous voyons que l'opérateur « . » permet de matcher n'importe quel caractère singulier de type alphanumérique ou de n'importe quel type spécifié. En effet, l'opérateur « . » matche avec cdc1 dont la valeur est « M », cdc3 dont la valeur est « 9 » et cdc4 dont la valeur est « _ ».

A noter que l'opérateur « . » ne matche pas la chaîne de caractères cdc2 dont la valeur est « MS ». La raison est que l'opérateur « . » matche un caractère unique. Pour qu'il puisse matcher un group de caractères, il faut pour cela utiliser les opérateurs de groupage « ( ) » et les opérateurs de quantification « ?,+,*,{} ». Les opérateurs de groupage permettent de grouper un ensemble de caractères dans un motif tandis que les opérateurs de quantification permettent d'agir sur le nombre de caractères à considérer pour former un groupe de caractères. Nous reviendrons plus tard sur l'usage des opérateurs de groupage et de quantification.

9.3.1.2 L'opérateur «. * » : matcher n'importe quelle chaîne de caractères

Il ne faut pas confondre l'opérateur « . » avec le caractère générique « * ». Le caractère générique « * » encore connu sous le nom de wildcard character permet de représenter n'importe quel groupe constitué de 0 à n caractères (standards ou spéciaux). Le caractère générique « * » n'est pas un caractère propre seulement aux expressions régulières. Il est plus souvent utilisé dans les traitements de texte pour préfixer ou suffixer de manière générique un mot-clé afin de rendre celui-ci plus générique. Par exemple, une chaîne de caractères définie comme ceci «version_1.* » permt de représenter n'importe quelle chaîne parmi : «version_1.0 », «version_1.1.5x », «version_1.4 ». Cette faculté du caractère « * » de représenter de manière générique n'importe quel ensemble de caractères fait qu'en le combinant avec l'opérateur «. » tel que « .* », on obtient ainsi un opérateur regex à part entière. Cet opérateur « .* » permet de matcher sans restriction n'importe quel chaîne de caractères quelle que soit sa longueur. Pour illustrer l'usage de l'opérateur « .* », reprenons l'exemple déjà utilisé pour l'opérateur « . » et redéfinissons le motif comme ceci « .* ».

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif à partir l'opérateur "."
        String motif=".*";
        Pattern pattern = Pattern.compile(motif);

        // Cas où le caractère est un lettre
        String cdc1="M"; // Une lettre
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");
        // Cas où le caractère est un lettre
        String cdc2="MS"; // Deux lettres
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches(); // Ne matchera pas.
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");
        // Cas où le caractère est un chiffre
        String cdc3="9"; // Un chiffre
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
```

```
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");

        // Cas où le caractère est un caractère spéciale: « _ », « _ », «-
»,«&», «$»,«#», «/»,«\»,«,» , « :», «.»
        String cdc4="_"; // Un underscore
        Matcher matcher4 = pattern.matcher(cdc4);
        boolean bool4 = matcher4.matches();
        if (bool4==true)
            System.out.println("La chaîne de caractères cdc4 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc4 ne matche pas avec
le motif");

    }
}
```

Output :

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 matche avec le motif
La chaîne de caractères cdc4 matche avec le motif
```

Suite à l'ajout du caractère « * » en plus du caractère « . » pour former le motif « .* », nous constatons que toutes les restrictions sont levées et toutes les chaînes de caractères testées contre le pattern « .* » matchent désormais. C'est le cas en particulier de cdc2 qui ne matche pas quand on utilise seulement le motif « . ». Pour rappel l'opérateur « . » permet de matcher un seul caractère tandis que le caractère générique « * » matche toutes les chaînes de caractères envisageables quelle que soit leur longueur. La combinaison des deux caractères permet de définir un motif universel qui matche n'importe qu'elle chaîne de caractères.

9.3.1.3 L'opérateur « ^ » : matcher une chaîne de caractères débutant par un motif donné

L'opérateur « ^ » permet de vérifier si le motif indiqué matche avec le début d'une chaîne de caractères considérée. Attention, toutefois, l'opérateur « ^ » signifie une négation logique (not) lorsqu'il est indiqué à l'intérieur d'un opérateur de classe de caractères (Nous présenterons plus tard les opérateurs de classe de caractères). Dans cette section, nous utilisons l'opérateur dans le sens « signification de début de chaîne de caractères ». L'exemple ci-dessous montre quelques usages de l'opérateur « ^ ».

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
```

```
        // Définir un motif à partir l'opérateur "^"
        String motif="^chan.*";
        Pattern pattern = Pattern.compile(motif,Pattern.CASE_INSENSITIVE); //
Ignorer la casse

        // Cas 1: Une chaîne de caractères commençant par "chan"
        String cdc1="chanson";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        // Cas 2: Une chaîne de caractères commençant par "chan"
        String cdc2="chantier";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches(); // Ne matchera pas.
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        // Cas 2: Une chaîne de caractères ne commençant pas par "chan"
        String cdc3="chaton"; // Un chiffre
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");

    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Les deux premières chaînes de caractères cdc1 et cdc2 matchent avec le motif car elles commencent toutes les deux avec le bout de chaîne de caractères « chan ». A l'inverse, la chaîne de caractères cdc3 ne matche pas du fait qu'elle commence non pas par le bout de chaîne de caractères « chan » mais plutôt par « chat ».

9.3.1.4 L'opérateur $: matcher une chaine de caractères finissant un motif

A l'inverse de l'opérateur ^ qui permet d'indiquer les caractères de début chaîne, l'opérateur $ permet d'indiquer les caractères de fin de chaîne. L'exemple ci-dessous illustre l'usage de l'opérateur $.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif à partir l'opérateur "$"
        String motif=".*son$";
        Pattern pattern = Pattern.compile(motif,Pattern.CASE_INSENSITIVE); //
Ignorer la casse

        // Cas 1: Une chaîne de caractères finissant par "son"
        String cdc1="Je chante une chanson";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        // Cas 2: Une chaîne de caractères finissant par "son"
        String cdc2="Je construit une maison";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches(); // Ne matchera pas.
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        // Cas 2: Une chaîne de caractères ne finissant pas par "son"
        String cdc3="J'adopte un chaton"; // Un chiffre
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");

    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

9.3.2 Les opérateurs regex composés

Jusque-là, nous avons exprimé les motifs sous une forme simple, c'est-à-dire en utilisant les opérateurs de base avec des bouts de chaîne de caractères facilement reconnaissables.

Ex : « est.* », « ^chan.* », « .*son$ ». A présent, nous allons utiliser des opérateurs regex composés, c'est-à-dire les opérateurs obtenus en combinant plusieurs opérateurs de base. Mais le langage regex offre déjà plusieurs opérateurs regex composés. Il s'agit notamment des opérateurs de classe, des opérateurs de quantification, des opérateurs logiques, des opérateurs de groupage, etc. Cette section a pour but de montrer l'usage des opérateurs regex composés.

9.3.2.1 L'opérateur de classe « [ ] » : matcher une chaine de caractères contenant un ensemble connu de caractères

L'opérateur de classe [ ] permet de spécifier un ensemble de caractères formant un motif en vue de matcher une chaîne de caractères. L'opérateur [ ] appliqué sur un ensemble de caractères se comporte comme un opérateur logique « OU » en ce sens qu'il considère chaque caractère indivuellement et non pris ensemble. Par exemple en spécifiant un motif tel que que [abc], cela signifie que la chaîne de caractères matche la lettre « a » ou la lettre « b » ou la lettre « c » et non pas que la chaîne de caractères matche le bout de chaîne « abc » en bloc uniquement. Il est donc important de garder à l'esprit cette propriété de l'opérateur [ ]. Le code ci-dessous montre quelques exemples d'usage de l'opérateur [ ].

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif représentant n'importe quel caractère minuscule
allant de a à z
        String motif="[a-z]";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="m";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        String cdc2="M";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="mm";
```

```
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 ne matche pas avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple, nous utilisons l'opérateur [ ] pour définir le motif « [a-z] ». Ce motif signifie toute lettre minuscule allant de a à z. Ex : « a » ou « b » ou « c »,...., ou « z ». Il permet donc de matcher n'importe quel caractère singulier alphabétique minuscule allant de a à z. En effet, la chaîne de caractères cdc1 matche bien le motif. Cependant le motif « [a-z] » ne pourra matcher ni les caractères alphabétiques majuscules, ni une association de lettres minuscules. C'est la raison pour laquelle les chaînes de caractères cdc2 et cdc3 ne matchent pas. En efffet, la chaîne cdc2 est formée d'un caractère majuscule. Tandis que la chaîne cdc3 est formée de plusieurs caractères. En fait l'opérateur [ ] est prévu pour matcher un seul caractère à la fois. Pour pouvoir matcher plusieurs caractères, il faut lui associer des opérateurs de quantifications qui permettent d'agir sur le nombre de caractères à prendre en compte. Nous reviendrons plus tard sur les opérateurs de quantifications dans les sous-sections suivantes.

Le tableau ci-dessous montre quelques exemples de motifs construits avec l'opérateur de classe [ ], leur descriptions ainsi que ainsi que quelques exemples de chaînes de caractères qu'ils matchent.

Tableau 12: Quelques exemples de motifs utilisant l'opérateur de classe [ ]

Motif	**Description**	**Chaîne de caractères qui matche**	**Chaîne de caractères qui ne matche pas**
[a-z]	Toute lettre minuscule comprise entre a et z inclusivement. Ex : a ou b ou c, ..., ou z	m	M
[A-Z]	Toute lettre majuscule comprise entre A et Z inclusivement. Ex : A ou B ou C, ..., ou Z	M	m
[0-9]	Tout chiffre compris entre 0 et 9 inclusivement. Ex : 0 ou 1 ou 2,...,ou 9	6	06

[0-9A-Z]	Tout chiffre compris entre 0 et 9 et toute lettre majuscule allant de A à Z.	6 ou M	06 ou m ou 6M, etc..
[a-zA-Z]	Toute lettre alphabétique	m ou M ou J	7
[abc]	La lettre a ou b ou c	a	A ou Ab
[^abc]	Tout caractère sauf les lettres a, b et c. Noter ici l'usage de l'opérateur [^..] qui signifie ici « not » au lieu de début de chaîne comme dans les opérateurs regex de base.	d ou 5 ou A	a ou kl ou ab
[^0-9]	Tout caractère sauf les chiffres allant de 0 à 9	A ou b	3 ou Ab
[a-kvy]	Toute lettre minuscule allant de a à k ou la lettre minuscule v ou la minuscule lettre y	h ou j	2 ou l ou z

9.3.2.2 Les opérateurs de quantification : gérer le nombre de caractères renvoyé par un motif

Les opérateurs de quantification sont des opérateurs regex permettant d'agir sur le nombre de caractères dans un motif. Jusque-là tous les motifs que nous avons utilisés sont soit des motifs renvoyant de manière générique n'importe quel caractère (ex : « . ») soit des motifs renvoyant des caractères bien identifiés (ex : « [abc]). Dans chacun de ces cas, un seul caractère est matché. A présent, nous souhaitons utiliser ces mêmes opérateurs en y associant des opérateurs capables de matcher un nombre variable de caractères (0 à n). Les opérateurs de quantification jouent ce rôle. Un quantificateur permet de spécifier le nombre d'occurrences exact ou maximum possible d'un élément ou d'un groupe d'éléments dans un motif. L'opérateur de quantification est toujours spécifié à la suite de l'opérateur définissant le motif qui identifie les caractères à matcher. On l'utilise le plus souvent à la suite d'un opérateur de classe [ ].

Parmi les opérateurs de quantification communément utilisés, nous avons par exemple le quantificateur « + ». Ce quantificateur signifie 1 ou plusieurs éléments. Par exemple la spécification « [abc] + » matche la lettre a ou b ou c à condition que la chaîne finale soit constituée de 1 caractère ou plus. Quelques chaînes de caractères qui peuvent matcher ce motif sont : « a », « ab », « abc », « abcb », « abcba», « bbbbbb ». En somme, avec le motif « [abc] + « , la chaîne de caractères continuera de matcher tant que cette chaîne est formée de l'une des trois lettres indiquées. Le motif ne matchera pas dès que la chaîne contient un autre caractère que les trois lettres a,b ou c. Le code ci-dessous illustre le cas discuté.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif pouvant être les lettres a ou b ou c
        String motif="[abc]+";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="a";
```

```
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        String cdc2="abcacb";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="abcd";
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple la chaîne de caractères cdc3 ne matche pas parce qu'elle contient la lettre « d » qui n'est pas prévue dans le motif [abc] indépendamment de l'opérateur de quantification +. En revanche, comme on peut le constater, la chaîne de caractères cdc2 matche car elle n'est constituée que des trois caractères prévus : a, b ou c. Et cela quel que soit le nombre d'occurrences de chaque caractère.

En plus de l'opérateur « + », il existe plusieurs autres opérateurs de quantification. Le tableau 13 ci-dessous présente les principaux indicateurs de quantification utilisés dans les opérations regex.

Tableau 13: Les opérateurs de quantification regex

Opérateur de quantification	**Description**
?	Indique que le caractère ou la chaîne de caractères se répère 0 ou 1 fois.

+	Indique que le caractère ou la chaîne de caractères se répète 1 fois ou plus
*	Indique que le caractère ou la chaîne de caractères se répète 0 fois ou plus
{n,m}	Indique que le caractère ou la chaîne de caractères se répète au moins n fois et au maximum m fois.
{n,}	Indique que le caractère ou la chaîne de caractères se répète au moins n fois.
{n}	Indique que le caractère ou la chaîne de caractères se répète exactement n fois
{,n}	Indique que le caractère ou la chaîne de caractères se répète au plus n fois.

Le tableau 14 ci-dessous montre quelques exemples d'utilisation de chaque opérateur de quantification appliqué sur un opérateur de classe []

Tableau 14: Quelques cas d'utilisation des opérateurs de quantification

Motif	Chaîne de caractères qui matche	Chaîne de caractères qui ne matche pas
[abc]+	«a», «b», «c», «abaa»,	«d», «abcd»
[abc]?	«»,«a», «c»	«bc» , «d»
[abc]*	«»,«a», «c», «abcccc»	«d», «abcd»
[abc]{3}	«aaa», «bbb», «ccc»,	«abc», «d», «ddd», «abcd»
[0-9]{4}	«2023», «2010»	«20», «20231»,
[abc]{1,2}	«a», «b», «c», «aa», «ab»,	«ad», «d»
[0-9a-zA-Z]{4}	«2023», «A023», «2y23», «aklm»	«A0235», «10156»
[0-9a-zA-Z]{1,4}	«abcd»,«A23»,«101»	«A0235», «10156»

9.3.2.3 L'opérateur logique de groupage « ( ) » : grouper un ensemble de caractères pour former un élément dans un motif

L'opérateur de groupage permet de traiter un ensemble de caractères comme s'il s'agit d'un caractère unique. Dans les sous-sections précédentes, nous avons présenté l'usage de l'opérateur de classe [ ] en montrant que cet opérateur se comporte comme l'opérateur

« ou ». Par exemple en indiquant [abc], les caractères a, b et c sont considérés individuellement pour chercher les matchs dans la chaîne de caractères de matching. A la différence de l'opérateur de classe [ ], l'opérateur de groupage () considère l'ensemble des caractères comme un bloc unique. Par exemple en indiquant (abc), on ne considère pas chaque caractère individuellement. On considère plutôt l'ensemble des caractères « abc » comme un bloc pour chercher un match dans la chaîne de caractères. L'exemple ci-dessous illustre un cas simple d'utilisation de l'opérateur () pour définir un motif.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif constitué des lettres abc en bloc
        String motif="(abc)";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="abc";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractère cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractère cdc1 ne matche pas avec
le motif");

        String cdc2="ab";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="abcd";
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 ne matche pas avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple, seule la chaîne de caractères cdc1 matche avec le motif. Car elle correspond au bloc de caractères représenté par (abc).

Combiner l'opérateur de groupage avec un opérateur de classe et un opérateur de quantification

A noter qu'on peut aussi appliquer un opérateur de groupage sur un opérateur de classe sur lequel on applique un opérateur de quantification pour définir un motif. Les exemple ci-dessous montrent la combinaison de l'opérateur de groupage avec l'opérateur de classe quantifié.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir un motif combinant les opérateurs de classe, de groupage et
de quantification
        String motif="([abc]*)";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="abc";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        String cdc2="ab";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="abcd";
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple, les chaînes de caractères cdc1 et cdc2 matchent avec le motif traduisant la combinaison des différents opérateurs. La chaîne cdc3 ne matche pas, mais non pas à cause du nombre de caractères, mais plutôt à cause du caractère supplémentaire « d » qui n'est pas prévu dans le motif.

Le tableau 15 ci-dessous montre quelques cas de combinaisons d'opérateurs de classe, de groupage et de quantification.

Tableau 15: Quelques cas illustrant la combinaison des opérateurs de classe, de groupage et de quantification

Motif	Chaîne de caractères qui matche	Chaîne de caractères qui ne matche pas
([abc]*)	«abc», «ab», «aaaaa»,	«abcd»
([abc]?)	«»,«a», «c»	«bc» , «d»
([abc]{3})	«aaa», «bbb», «ccc»,	«abc», «d», «ddd», «abcd»
([0-9]{4})	«2023», «2010»	«20», «20231»,
(^(abc){3}$)	«abcabcabc»	«abcabcakl»

On remarque à travers ces exemples qu'au prime abord, l'usage de l'opérateur de groupage () pour encapsuler un opérateur de quantification appliqué sur un opérateur de classe n'apporte aucun changement par rapport au non usage de l'opérateur de groupage. Cependant le véritable intérêt de l'usage de l'opérateur de groupage apparaît quand il s'agit d'élaborer un motif complexe combinant plusieurs sous-motifs. Voir ci-après les cas où l'usage de l'opérateur de groupage montre toute sa pertinence.

Utiliser l'opérateur de groupage pour combiner plusieurs sous-motifs

Rappelons que le but ultime de l'usage de l'opérateur de groupage () est de pouvoir combiner plusieurs sous-motifs pour construire un motif capable de répondre aux cas les plus complexes. L'exemple ci-dessous montre un cas d'utilisation de l'opérateur de groupage () pour combiner des sous-motifs pour construire des motifs plus complexes.

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
```

```
    public static void main(String[] args)  {
        // Définir un motif représentant n'importe quel caractère miniscule
allant de a à z
        String motif="([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="2023-05-28";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        String cdc2="0000-00-00";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="202-05-28";
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple, nous combinons plusieurs sous-motifs pour construire un motif final. Chaque sous-motif est identifiable à travers l'usage de l'opérateur de groupage (). A l'intérieur de chaque opérateur de groupage, nous utilisons un opérateur de classe [ ] mais également un opérateur de quantification { }. Etant donné que le motif final que nous souhaitons construire est celui qui pourra matcher toute chaîne de caractères se présentant sous forme de date de format AAAA-MM-JJ, alors, nous ajoutons le caractère « -» pour relier chaque sous-motif formé par les opérateur de groupage (). Le motif final ainsi obtenu se présente comme suit : ([0-9]{4})-([0-9]{1,2})-([0-9]{1,2}).

Nous testons les trois chaînes de caractères cdc1, cdc2 et cdc3 contre ce motif. Les chaînes de caractères cdc1 et cdc2 matchent le motif. En revanche la chaîne cdc3 ne matche pas le motif construit (voir le code de l'exemple plus haut).

Opérateurs de groupage disjoint ou imbriqués

L'opérateur de groupage peut également être utilisé pour traduire des sous-motifs imbriqués. En effet, la plupart du temps, les opérateurs de groupage sont souvent utilisés de manière disjointe. Nous avons déjà évoqué cet aspect dans la section consacrée à l'appel de la méthode group(). Par exemples, un motif défini tel que « (A)(B)(C) » représente un cas d'utilisation de l'opérateur disjoint. Tandis qu'un motif défini tel que « ((A)(B(C))) » est un cas de groupes imbriqués.

Le tableau 16 ci-dessous montre quelques cas d'utilisation de l'opérateur de groupage () pour construire des motifs plus complexes.

Tableau 16: Construction de motif complexe en utilisant l'opérateur de groupage ()

Motif	**Chaîne de caractères qui matche**	**Chaîne de caractères qui ne matche pas**
^([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})$	2023-05-28	001986-05-2800
([0-9]{6})-([0-9]{1,2})-([0-9]{1,4})	001986-05-2800	001986-053-2800
^([a-zA-Z]{6})-([a-zA-Z]{4})$	Rendez-vous	Rendez-vous lundi
(A)(B)(C)	ABC	AbC, abc, ABc
((A)(B(C)))	ABC	AbC, abc, ABc

9.3.2.4 L'opérateur logique ou « | » : matcher une chaîne de caractères contre plusieurs motifs

L'opérateur logique « | » permet tester une chaîne de caractères contre plusieurs motifs (ou plus exactement plusieurs sous-motifs). La chaîne de caractères est considérée comme matchée lorsqu'elle matche au moins l'un des sous-motifs. Les sous-motifs sont séparés par l'opérateur « | ». L'exemple ci-dessous illustre quelques cas d'utilisation de l'opérateur « | ».

```
package com.tuto.regex;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Main {
    public static void main(String[] args)  {
        // Définir plusieurs motifs séparés par l'opérateur |
        String motif=".*(est|ouest).*";
        Pattern pattern = Pattern.compile(motif);

        String cdc1="Elle est allée au bureau";
        Matcher matcher1 = pattern.matcher(cdc1);
        boolean bool1 = matcher1.matches();
        if (bool1==true)
            System.out.println("La chaîne de caractères cdc1 matche avec le
```

```
motif");
        else
            System.out.println("La chaîne de caractères cdc1 ne matche pas avec
le motif");

        String cdc2="La ville se trouve à l'ouest du pays";
        Matcher matcher2 = pattern.matcher(cdc2);
        boolean bool2 = matcher2.matches();
        if (bool2==true)
            System.out.println("La chaîne de caractères cdc2 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc2 ne matche pas avec
le motif");

        String cdc3="La ville se trouve au sud du pays";
        Matcher matcher3 = pattern.matcher(cdc3);
        boolean bool3 = matcher3.matches();
        if (bool3==true)
            System.out.println("La chaîne de caractères cdc3 matche avec le
motif");
        else
            System.out.println("La chaîne de caractères cdc3 ne matche pas avec
le motif");
    }
}
```

Output :

```
La chaîne de caractères cdc1 matche avec le motif
La chaîne de caractères cdc2 matche avec le motif
La chaîne de caractères cdc3 ne matche pas avec le motif
```

Dans cet exemple, nous définissons un motif dans lequel nous utilisons l'opérateur « | » pour choisir entre deux bouts de chaînes « est » et « ouest ». Noter ici l'utilisation de l'opérateur de groupage () dont le rôle est de renvoyer un ensemble de caractères comme un seul bloc d'éléments, contrairement à l'opérateur de classe [ ] qui renvoie un seul caractère parmi l'ensemble des caractères spécifiés. Dans le cas présent, grâce à l'usage de l'opérateur (), l'ensemble des caractères « est » est renvoyé comme un seul bloc. Il en est de même pour l'ensemble des caractères « ouest ». Ainsi, étant donnés deux blocs de caractères distincts, l'usage de l'opérateur « | » permet de construire implicitement deux motifs finaux. Chaque motif est testé contre la chaîne de caractères fournie. Et la chaîne de caractères est considérée comme matchée si au moins un des motifs renvoyés matche. Dans l'exemple ci-dessous, la chaîne de caractères cdc1 et cdc2 matchent toutes les deux avec le motif général spécifié. La chaîne cdc1 contient le mot « est ». Tandis que la chaîne cdc2 contient « ouest ». En revanche la chaine cdc3 ne matche pas, car elle ne contient aucun des mots-clés spécifiés dans le motif général.

Le tableau 17 ci-dessous montre quelques exemples d'utilisation de l'opération logique « | ».

Tableau 17: Quelques exemples d'utilisation de l'opérateur « | »

Motif	Chaîne de caractères qui matche	Chaîne de caractères qui ne matche pas
(^chan.*)|(.*son$)	chant, chanson, maison	chaton
([abc]+)|([123]+)	Abaa, 1333	126
([abc]{3})|([0-9]{4})	abc,1333	abca
(^(abc){3}$)\|(^([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})$)	abcabcabc, 2023-05-28	abcabc,2023-0528

9.3.2.5 Les opérateurs regex prédéfinis

Il existe un certain nombre d'opérateurs regex prédéfinis qui traduisent dans un formalisme plus simple les opérateurs standards. La plupart des opérateurs prédéfinis sont représentés par une simple lettre alphabétique précédé du symbole « \ » . Parmi les opérateurs prédéfinis, nous avons par exemple, « \d » où d signifie digit. Cet opérateur permet de matcher n'importe quel chiffre entre 0 et 9. Il correspond donc à l'opérateur standard [0-9]. Notons qu'en écrivant avec la lettre d en majuscule c'est à dire « \D » on obtient un opérateur qui signifie tout caractère sauf les chiffres. En expression standard, cet opérateur s'écrit comme [^0-9]. Le tableau 18 ci-dessous liste quelques opérateurs prédéfinis ainsi que leur équivalent standard.

Tableau 18: les principaux opérateurs regex prédéfinis

Opérateur prédéfini	Description	Correspondance standard
\d	Tout chiffre entre 0 et 9	[0-9]
\D	Tout caractère à l'exception des chiffres	[^0-9]
\s	Tout caractère blanc (espace, tabulation, retour-chariot, etc…).	[\t\n\x0B\f\r]
\S	Tout caractère à l'exception des caractère blancs (espace, tabulation, retour-chariot, etc…).. Autre formulation : [^\s]	(^[\t\n\x0B\f\r])
\w	Tout caractère utilisable dans un mot à l'exception des caractères accentués : w pour word. Cet opérateur matche tous les caractères alphabétiques minuscules et majuscules, les chiffres et le underscore).	[a-zA-Z_0-9]
\W	Inverse de l'opérateur \w, cet opérateur matche tous les caractères ne servant pas écrire un mot (word). Autre formulation : [^\w]	[a-zA-Z_0-9]

\b	Matche le début ou la fin d'un mot.	
\B	Matche le début ou la fin d'un élément qui n'est pas un mot. Autre formulation : [^\b]	
\A	Matche le début d'une entrée.	
\G	Matche la fin d'un bout de texte qui a été trouvé précédemment.	
\Z	Matche la fin d'une entrée, sauf s'il s'agit de la fin du texte.	
\z	Matche la fin d'une entrée, y compris s'il s'agit de la fin du texte.	
\p{Digit}	Matche tout chiffre [0-9] et tout caractère unicode qui représente un chiffre en base décimal	
\p{Punct}	Matche un caractère de ponctuation	
\p{Space}	Match un caractère blanc	
\p{javaLowerCase}	Matche tout caractère minuscule y compris les caractères accentués	
\p{javaUpperCase}	Matche tout caractère majuscule y compris les caractères accentués	
\p{javaWhitespace}	Matche tout caractère représentant un espace.	

9.3.2.6 L'opérateur d'échappement de caractères spéciaux : \

Lorsque les symboles ^$|?*+. ( )[]{}\ apparaissent comme caractères libres dans un motif, ils se comportent systématiquement comme des opérateurs de regex. Pour pouvoir les traiter comme des caractères ordinaires, il faut donc leur appliquer le caractère d'échappement antislash « \ ». Par exemple, supposons un motif permettant de matcher tous les produits dont le prix est dix dollars et cinquante centimes, exprimé comme suit: « 10.5$ ». En appliquant une compilation regex sur cette chaîne de caractères, le motif renvoyé aura une toute autre signification. En effet, il signifie toute chaîne de caractères se terminant par le chiffre 10 suivi de n'importe quel caractère (représenté ici par le point « . »). Ainsi, compiler le motif suivant cette interprétation ne permet pas d'atteindre l'objectif recherché. Pour que les caractères « . » et « $ » puissent être considérés comme des caractères normaux dans le motif, il faut alors les faire précéder par le symbole antislash « \ » tel que : « 10\.5\$ »

Le tableau 19 ci-dessous montre les opérateurs regex concernés par l'utilisation du caractère d'échappement dans un motif regex.

Tableau 19: Traitement des caractères entrant dans les opérateurs regex

Opérateur regex	**Chaîne de caractères**

.	\.
^	\^
$	\$
?	\ ?
*	*
+	\+
(	\(
)	\)
{	\{
}	\}
\	\\

10 GESTION DES ERREURS ET EXCEPTIONS

10.1Généralités sur les erreurs et exceptions

La gestion des erreurs et exceptions occupe une place centrale dans la conception de tout programme de traitement. Lors de la compilation ou de l'exécution d'un programme, un certain nombre d'imprévus peuvent survenir, entrainant une rupture dans le séquencement des instructions et conduisant à l'arrêt du programme. Il s'agit des erreurs et des exceptions. Plusieurs situations peuvent être à l'origine des erreurs et exceptions dans un programme : lecture de fichier inexistant, insuffisance du droit d'accès à un fichier ou à un répertoire, problème de connection à un réseau, opération arithmétique sur une valeur en chaîne de caractères, division par zéro, appel de méthode sur un objet de valeur nulle, insuffisance de ressources mémoire, etc… La gestion des erreurs et exceptions vise donc à anticiper ces imprévus et à les gérer de manière adéquate.

Dans ce chapitre, nous passons en revue les principaux types d'erreurs et exceptions rencontrés dans un programme. En particulier, nous présentons les différentes manières de gérer les erreurs et exceptions à savoir « jeter » une exception et « capturer » une exception. Jeter et capturer des exceptions sont deux étapes essentielles dans la gestion des exceptions. Jeter une exception consiste à récupérer et à renvoyer à l'utilisateur les informations sur la nature et les causes de l'exception survenue. Et capturer une exception consiste à prévoir et à définir des instructions spécifiques à exécuter lorsque l'exception survient. Nous reviendrons plus en détail sur les notions de jeter et capturer les exceptions.

10.2 Différences entre erreur et exception

Notons d'entrée que toutes les classes d'erreurs et exceptions héritent de la classe Throwable. La classe Throwable est étendue par deux classes : Error et Exception. La classe Error est la classe mère de toutes les classes d'erreurs. Tandis que la classe Exception est la classe mère de toute les classes Exception[20]. Il existe ainsi une différence notable entre les erreurs et les exceptions.

Au sens strict, une erreur est toute situation anormale dont la survenue exige l'arrêt de l'exécution du programme. En principe, en cas de survenue d'une erreur, l'utilisateur doit absolument la corriger. Les erreurs n'ont pas vocation à être gérées dans le code. Une erreur peut provenir soit d'un problème de syntaxe d'écriture ou de structure de code, soit d'un problème lié à l'environnement d'exécution: ressources mémoires insuffisantes, erreurs de connection au réseau, droit d'accès, etc… De ce point de vue les erreurs sont incidentales.

Une exception, quant à elle, provient exclusivement des instructions définies dans le code: lecture d'un fichier inexistant, division par zéro, etc.. La survenue d'une exception ne

[20] Détails sur les classes Throwable, Error et Exception.
https://docs.oracle.com/javase/10/docs/api/java/lang/Throwable.html
https://docs.oracle.com/javase/10/docs/api/java/lang/Error.html
https://docs.oracle.com/javase/10/docs/api/java/lang/Exception.html

nécessite pas nécessairement l'arrêt de l'exécution du programme. Les exceptions sont en principes gérables dans le code.

Remarquons toutefois que même si les erreurs ne doivent pas être gérées dans le code, disons qu'il peut y avoir des « exceptions » à cette règle. En effet, tenant compte de leur degré de gravité relativement faible, certaines erreurs peuvent être anticipées et gérées dans le code. C'est le cas par exemples des erreurs de connection, des droits d'accès. Par exemple, en cas d'erreur d'accès à une répertoire sur le FileSystem, au lieu de stopper brusquement l'exécution du programme, on peut capturer[21] cette erreur, envoyer un log d'erreur personnalisé à l'utilisateur et peut-être même exécuter d'autres instructions.

Faisons aussi remarquer que la plupart des erreurs et exceptions de compilation peut être corrigée lors de l'écriture du programme. Et cela grâce aux fonctionnalités qu'offrent les IDEs (Eclipse, IntelliJ, Netbeans, etc..). Il s'agit en particulier des erreurs et exceptions dites « contrôlées ». Nous reviendrons plus tard sur la classification des erreurs et exceptions.

10.3 Quelques classes d'erreurs et exceptions

Dans cette section, nous allons présenter quelques cas d'erreurs et exceptions.

10.3.1 Lecture d'un fichier inexistant : FileNotFoundException

Le code ci-dessous montre l'exemple d'une exception dans le cas de la lecture d'un fichier inexistant.

```
package com.tuto.exception;
import java.io.FileReader;
import java.io.IOException;

public class Main {
        public static void main(String[] args) throws IOException {
                // Référence à un fichier
                FileReader fileReader = new FileReader("myFile.txt");
                System.out.println(fileReader.read()); // Lecture du fichier
inéxistant
                fileReader.close();
}
}
```

Output :

```
Exception in thread "main" java.io.FileNotFoundException: myFile.txt (Le
fichier spécifié est introuvable)
at java.base/java.io.FileInputStream.open0(Native Method)
at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
at java.base/java.io.FileInputStream.<init>(FileInputStream.java:158)
at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
at java.base/java.io.FileReader.<init>(FileReader.java:60)
at com.tuto.exception.Main.main(Main.java:8)
```

[21] Nous reviendrons plus tard sur les notions de capturer une erreur ou une exception.

```
Process finished with exit code 1
```

En exécutant cet exemple, on obtient une exception de type FileNotFoundException qui est une sous-classe de l'exception IOException.

10.3.2 Appel de méthode sur un objet null : NullPointerException

L'exemple ci-dessous illustre une exception qui survient en appelant une méthode sur un objet de valeur null.

```
package com.tuto.exception;
public class Main {
        // Définir une méthode qui renvoie la longeur d'une valeur String
        private static int getLength(String str){
                return str.length();
        }
        public static void main(String[] args)  {
                // Appel de la méthode getLength() avec une valeur null
                String myStr=null;
                int longeur= getLength(myStr);
                System.out.println(longeur);

        }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because "str" is null
at com.tuto.exception.Main.getLength(Main.java:5)
at com.tuto.exception.Main.main(Main.java:10)
Process finished with exit code 1
```

Dans cet exemple, la méthode getLength() renvoie la longueur de tout variable String passé en paramètre en appelant la méthode length(). Comme nous avons appelé la méthode en lui passant une valeur nulle (la variable myStr), cet appel renvoie une exception qui est de type NullPointerException.

10.3.3 Récupérer un élément hors périmètre : IndexOutOfBoundsException

Cette exception survient généralement lorsqu'on essaie d'accéder à un élément d'une séquence de valeurs en spécifiant un indice qui ne trouve pas dans la séquence. L'exemple ci-dessous illustre une situation où l'exception IndexOutOfBoundsException est renvoyée.

```
package com.tuto.exception;

import java.util.ArrayList;
import java.util.List;
public class Main {
```

```
        public static void main(String[] args) {
                // Définir une liste contenant quelques langages de
programmation
                List<String> langages = new ArrayList<>();
                langages.add("Java");
                langages.add("Python");
                langages.add("Scala");
                String choix = langages.get(4); // On tente de récupérer le
cinquième élément de la liste ( 0 est l'indice du premier élément)
        }
}
```

Output :

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 4 out of
bounds for length 3
at
java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:100)
at
java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.j
ava:106)
at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:302)
at java.base/java.util.Objects.checkIndex(Objects.java:385)
at java.base/java.util.ArrayList.get(ArrayList.java:427)
at com.tuto.exception.Main.main(Main.java:13)

Process finished with exit code 1
```

Dans l'exemple, nous avons créé d'abord une liste et ajouter trois éléments. Ensuite, nous appelons la méthode get() en spécifiant l'indice 4. Ce qui signifie récupérer le cinquième élément de la liste. Mais la liste ne contenant que trois éléments, cet appel renvoie nécessairement une exception en l'occurrence IndexOutOfBoundsException.

10.3.4 Convertir un objet en un type incompatible : ClassCastException

L'exemple ci-dessous montre un cas d'exception qui survient lorsqu'on essaie de convertir une valeur en un type incompatible.

```
package com.tuto.exception;
public class Main {
        public static void main(String[] args)  {
                // Définir une variable de type classe String
                String myStringVar="chaîne de caractères";
                // Convertir la variable de type String en une variable de type
classe Integer
                Integer myIntegerVar=(Integer)(Object)myStringVar;
                System.out.println(myIntegerVar);
        }
}
```

Output :

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.String
cannot be cast to class java.lang.Integer (java.lang.String and
java.lang.Integer are in module java.base of loader 'bootstrap')
at com.tuto.exception.Main.main(Main.java:7)

Process finished with exit code 1
```

Dans cet exemple, nous créons d'abord une variable de type String nommée myStringVar. Ensuite, nous créons une deuxième variable nommée myIntegerVar dont la valeur est égale à la valeur myStringVar que nous essayons de caster en un type Integer. Cette tentative de cast passe bien à la compilation, car nous castons d'abord myStringVar en type Object qui est la classe mère de toutes les classes concrètes Java. Ensuite, nous castons la classe Object en un type Integer. Ce n'est qu'à l'exécution du code que l'incompatibilité entre le type String et le type Integer est détecté. D'où la survenu de l'exception ClassCastException.

10.4 Classification des exceptions : les exceptions contrôlées et les exceptions non contrôlées

On peut classer les exceptions Java en deux catégories : les exceptions « contrôlées » (*checked exceptions*) et les exceptions « non contrôlées » (*unchecked exceptions*).

Les exceptions contrôlées : sont des exceptions qui sont évaluées au moment de la compilation du code. Lorsqu'une méthode est susceptible de renvoyer une exception contrôlée, le compilateur oblige le programmeur à gérer cette exception avant de soumettre le code à la compilation. Il existe deux manières de gérer une exception contrôlée: soit spécifier une instruction throws lors de la définition de la méthode qui contient l'instruction susceptible de renvoyer l'exception concernée, soit mettre dans un bloc try/catch l'instruction susceptible de renvoyer l'exception[22]. Parmi les exceptions contrôlées, on dénote entre autres les exceptions : IOException, FileNotFoundException, ClasseNotFoundException, InstanciationException, InterruptedException, SQLException, etc…

Les exceptions non contrôlées : correspondent aux exceptions qui sont évaluées à l'exécution et non à la compilation. Le compilateur n'oblige donc pas le développeur à gérer ces exceptions lors de l'écriture du programme. Parmi ces exceptions, on dénote entre autres les exceptions : NullPointerException, ClassCastException, ArithmeticException, IndexOutOfBoundsException, etc…

10.5 Jeter une exception : l'instruction throws/throw

Comme nous l'avons déjà indiqué, jeter une exception consiste à renvoyer à l'utilisateur les détails sur la nature et les causes de l'exception survenue. Une exception peut être soit incidentalement jetée, soit délibérément jetée. Une exception incidentale est une exception qui survient lors de la compilation ou de l'exécution sans que le programmeur l'ait voulu.

[22] Nous reviendrons plus pard sur l'usage de l'instructions throw et l'usage du bloc try/catch

L'exception survient donc comme un incident, car elle n'est pas souhaitée à l'avance par l'utilisateur. En principe toutes les erreurs et exceptions sont jetées de manière incidentale compte tenu de leur nature imprévue. En revanche, il arrive que le développeur jette délibérément une exception, par exemple lorsqu'une condition est (ou n'est pas) satisfaite. Dans cette section, nous allons présenter les deux manières de jeter les exceptions : cas où l'exception est jetée de manière incidentale et cas où l'exception est jetée de manière délibérée.

10.5.1 Cas où l'exception est jetée de manière incidentale

La manière de jeter une exception incidentale peut être différente selon qu'il s'agisse d'une exception « contrôlée » ou d'une exception « non contrôlée ». Pour rappel, les exceptions contrôlées sont des exceptions qui sont évaluées à la compilation tandis que les exceptions non contrôlées sont des exceptions qui ne sont évaluées qu'à l'exécution. Les deux exemples ci-après montre comment les exceptions incidentales sont jetées selon qu'il s'agisse d'une exception contrôlée ou d'une exception non contrôlée.

10.5.1.1 Jeter une exception controlée

```
package com.tuto.exception;
public class Main {
        public static void main(String[] args) throws ClassNotFoundException {
                // Récupérer le nom de la classe Product depuis le package
com.tuto.company
                String myClassName =
String.valueOf(Class.forName("com.tuto.company.Product"));
                // Afficher les caractéristiques de la classe
               Object
myClass=ClassLoader.getSystemClassLoader().loadClass(myClassName);
                System.out.println(myClass.toString());
        }
}
```

Output :

```
Exception in thread "main" java.lang.ClassNotFoundException:
com.tuto.company.Product
at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.j
ava:641)
at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoader
s.java:188)
at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
at java.base/java.lang.Class.forName0(Native Method)
at java.base/java.lang.Class.forName(Class.java:391)
at java.base/java.lang.Class.forName(Class.java:382)
at com.tuto.exception.Main.main(Main.java:5)

Process finished with exit code 1
```

Dans l'exemple ci-dessous, nous référençons d'abord une classe nommée Product située dans un package nommé com.tuto.company. Cette classe n'existe pas en réalité. Ensuite, nous appelons les méthodes Class.forName() et ClassLoader.getSystemClassLoader().loadClass() qui ont pour rôle respectivement de construire un nom de classe à partir d'une valeur String et de charger cette classe dans le programme courant. Rappelons que chacune des deux méthodes renvoient nativement une exception nommée ClassNotFoundException. Cette exception étant une exception contrôlée, le compilateur oblige le développeur à spécifier l'instruction throws ClassNotFoundException lors de la définition de la méthode qui appelle l'une des méthodes. C'est la raison pour laquelle, la définition de la méthode main(String[] args), nous avons été obligé d'ajouter l'instruction throws ClassNotFoundException afin de pouvoir compiler le code.

La compilation du code étant effectuée, en lançant le code nous avons reçu l'exception ClassNotFoundException car en réalité la classe Product n'existe pas.

10.5.1.2 Jeter une exception non contrôlée

Contrairement à une exception contrôlée, pour jeter une exception non contrôlée, il n'est pas nécessaire d'indiquer l'instruction throws lors de la définition de la méthode qui appelle l'instruction susceptible d'envoyer l'exception. Une exception non contrôlée est jetée au moment de l'exécution du code. L'exemple ci-dessous en est une illustration.

```
package com.tuto.exception;
public class Main {
            // Définir une méthode qui renvoie la longeur d'une valeur String
        private static int getLength(String str){
                return str.length(); // Cette ligne renvoie une exception si la
variable str est null
            }
            public static void main(String[] args)  {
                // Appel de la méthode getLength() avec une valeur null
                String myStr=null;
                int longeur= getLength(myStr); // L'exception est déclenchée
ici car myStr est nulle.
                System.out.println(longeur);

            }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because "str" is null
at com.tuto.exception.Main.getLength(Main.java:5)
at com.tuto.exception.Main.main(Main.java:10)

Process finished with exit code 1
```

Cet exemple renvoie une exception de type NullPointerException car la méthode length() est appelée sur une valeur nulle. Et comme NullPointerException n'est pas une exception

contrôlée, on ne peut la jeter qu'à l'exécution du code et non à la compilation. C'est pourquoi, l'instruction throws n'a pas été spécifiée ni dans la définition de méthode main(), ni dans la définition de la méthode getLength().

Rappelons toutefois qu'il n'est pas interdit de spécifier l'instruction throws pour définir la méthode qui est susceptible de générer une exception non contrôlée. Par exemple dans l'exemple ci-dessous, on pouvait bien définir la méthode main() ou la méthode getLength() en spécifiant l'instruction throws NullPointerException. Mais cela n'aurait pas été d'un grand apport car la NullPointerException n'est pas une exception bloquante pour la compilation.

10.5.2 Cas où l'exception est délibérément jetée par l'utilisateur

Il arrive très souvent que le programmeur veuille délibérément jeter une exception en se basant sur ses propres critères, et cela indépendamment des exceptions qui peuvent survenir de manières imprévues. Généralement, pour jeter une exception délibérée, on définit une ou plusieurs conditions booléennes dont les valeurs servent à jeter (ou pas) l'exception. Par exemple, nous souhaitons élaborer un programme qui applique un taux d'intérêt sur un emprunt. Mais, supposons d'avance que selon la loi en vigueur pour un emprunt le taux d'intérêt ne doit pas dépasser 30%. Nous supposons aussi que le taux pour cet emprunt ne peut pas être négatif. Alors, nous mettons en place une structure de contrôle IF… ELSE qui renvoie une exception lorsque le taux indiqué par l'utilisateur n'est pas dans la fourchette indiquée. Il s'agit dans ce cas d'une exception délibérée. On peut distinguer deux formes d'exceptions délibérées. Les exceptions natives Java délibérément jetées par l'utilisateur et les exceptions conçues par l'utilisateur lui-même. Dans cette sous-section, nous allons présenter les deux formes d'exception.

10.5.2.1 Jeter délibérément une exception native Java

L'exemple ci-dessous montre les cas où une exception native est délibérément jetée par le programmeur.

```
package com.tuto.exception;
public class Main {

        // Définir une méhode qui vérifie la valeur du taux : valeur comprise
entre 0 et 1
            public static boolean verfieTaux(double taux){
                if (taux > 0 && taux > 30) {
                        throw new IllegalArgumentException("La valeur du taux
est incorrecte");
                }
                return true;
            }
            public static void main(String[] args) {
                // Appel de la méthode VerfieTaux en spécifiant une valeur
incorrecte
                double monTaux=40.5;
```

```
                boolean tauxValide=verfieTaux(monTaux);
                System.out.println("taux vérifiée");
            }
}
```

Output :

```
Exception in thread "main" java.lang.IllegalArgumentException: La valeur du
taux est incorrecte
at com.tuto.exception.Main.verfieTaux(Main.java:7)
at com.tuto.exception.Main.main(Main.java:14)

Process finished with exit code 1
```

Dans cet exemple, nous définissons une méthode qui s'appelle verfieTaux() dont le but est de vérifier si la valeur passée en paramètre est comprise entre 0 et 30. Dans le cas contraire, nous jetons délibérément une exception de type IllegalArgumentException. Dans la méthode main(), nous définissons une variable nommée monTaux dont la valeur est 40.5. Ensuite, nous appelons la méthode verifieTaux() en lui passant la valeur de monTaux. Mais puisque la valeur du taux doit être comprise entre 0 et 40, l'exception IllegalArgumentException est jetée.

10.5.2.2 Jeter délibérément une exception conçue par l'utilisateur

Java laisse la possibilité au programmeur de développer sa propre exception et de jeter cette exception partout dans le code où il en a besoin. Dans cette section, nous allons montrer comment définir sa propre exception et comment jeter cette exception dans le code.

Concevoir une exception-utilisateur

Rappelons que toutes les exceptions héritent de la classe Exception, qui, elle-même, hérite de la classe Throwable. Une exception conçue par l'utilisateur doit donc nécessairement étendre la classe Exception ou l'une de ses classes dérivées. L'exemple ci-dessous montre la définition d'une exception utilisateur définie en étendant la classe IllegalArgumentException.

```
package com.tuto.exception;
class TauxInvalideException  extends IllegalArgumentException {

    // Constructeur de la classe TauxInvalideException
    public TauxInvalideException (String message) {
        // Appel de la classe IllegalArgumentException
        super(message);
    }
}
```

Dans cet exemple, nous définissions une classe nommée TauxInvalideException qui étend la classe IllegalArgumentException[23]. La création d'une exception utilisateur reste très simple car il suffit seulement d'appeler le constructeur de la classe d'exception héritée avec le mot-clé super.

La classe d'exception TauxInvalideException étant mainteant définie, nous pouvons la jeter délibérément dans notre code partout où il y a besoin. L'exemple ci-dessous reprend l'exemple de la vérification du taux. Lorsque la valeur du taux n'est pas comprise entre 0 et 30, on jette une exception (voir exemple).

```
package com.tuto.exception;
import com.tuto.exception.TauxInvalideException;
public class Main {

        // Définir une méhode qui vérifie la valeur du taux : valeur comprise
entre 0 et 1
            public static boolean verfieTaux(double taux){
                if (taux > 0 && taux > 30) {
                        throw new TauxInvalideException("La valeur du taux est
incorrecte");
                }
                return true;
            }
            public static void main(String[] args) {
                // Appel de la méthode VerfieTaux en spécifiant une valeur
incorrecte
                double monTaux=40.5;
                boolean tauxValide=verfieTaux(monTaux);
                System.out.println("taux vérifiée");
            }
}
```

Output :

```
Exception in thread "main" com.tuto.exception.TauxInvalideException: La valeur
du taux est incorrecte
at com.tuto.exception.Main.verfieTaux(Main.java:8)
at com.tuto.exception.Main.main(Main.java:15)

Process finished with exit code 1
```

Dans cet exemple, nous avons jeté une exception conçue par l'utilisateur (TauxInvalideException), bien que cette exception reste une classe dérivée d'une exception native Java en l'occurrence la classe IllegalArgumentException.

Signalons par ailleurs que dans les deux exemples de jets délibérés d'exceptions, la classe IllegalArgumentException est une exception non contrôlée. C'est pourquoi les définition des méthodes main() et verifieTaux() ne contiennent pas l'instruction throws. Mais si la classe d'exception utilisée avait été une exception contrôlée, il aurait été obligatoire de spécifier l'instruction throws dans la définition de chacun des méthodes ou d'utiliser les

[23] On peut étendre n'importe quelle classe héritée de la classe Throwable, qu'il s'agisse des erreurs (Classe Error) ou des exceptions (Classe Exception) et l'ensemble de leurs classes dérivées.

blocs try/catch autour des instructions susceptibles de renvoyer l'exception. Par exemple, considérons la classe d'exception utilisateur définie comme suit :

```
package com.tuto.exception;

import java.io.FileNotFoundException;

class CheminNonValideException  extends FileNotFoundException {

    // Constructeur de la classe CheminNonValideException
    public CheminNonValideException  (String message) {
        // Appel de la classe FileNotFoundException
        super(message);
    }
}
```

Cette classe d'exception CheminNonValideException étend la classe FileNotFoundException qui est une exception contrôlée.

Voyons maintenant à travers un exemple comment l'appel de cette exception se traduit en terme de structure de code par rapport aux précédents exemples de jets délibérés d'exception par l'utilisateur (Voir exemple ci-dessous).

```
package com.tuto.exception;
import com.tuto.exception.CheminNonValideException ;
public class Main {

         // Définir une méhode qui vérifie le chemin spécifié. Le chemin doit
commencer par /home
            public static boolean verifieChemin(String chemin) throws
CheminNonValideException  {
                if (! chemin.toLowerCase().startsWith("/home")) {
                        throw new CheminNonValideException ("Le chemin spécifié
est invalide");
                }
                return true;
            }
            public static void main(String[] args) throws
CheminNonValideException {
                // Appel de la méthode verifieChemin en spécifiant une valeur
incorrecte
                String chemin="/bin";
                boolean tauxValide=verifieChemin(chemin);
                System.out.println("Chemin valide");
            }
}
```

Output :

```
Exception in thread "main" com.tuto.exception.CheminNonValideException: Le
chemin spécifié est invalide
at com.tuto.exception.Main.verifieChemin(Main.java:8)
at com.tuto.exception.Main.main(Main.java:15)

Process finished with exit code 1
```

Dans cet exemple, nous définissons d'abord une méthode appelée verifieChemin() dont le but est de vérifier un chemin d'accès spécifié en paramètre. Lorsque le chemin spécifié ne commence pas par la valeur String « /home », nous jetons une exception nommée CheminNonValideException. Mais puisque cette exception hérite de la classe FileNotFoundException, qui est une exception contrôlée, le compilateur nous oblige à spécifier dans la définition de la méthode verifieChemin() l'instruction throws CheminNonValideException. De même, comme la méthode main() appelle la méthode verifieChemin(), le compilateur nous oblige également à spécifier l'instruction throws CheminNonValideException. Cet exemple illustre donc l'instanciation d'une classe d'exception utilisateur qui hérite d'une classe d'exception contrôlée et celle qui hérite d'une classe d'exception non contrôlée.

10.6 Capturer une exception : l'usage des blocs try/catch/finally

Par défaut, lorsqu'une exception est jetée, les détails sur la nature et la cause de l'exception sont fournis et l'exécution du programme s'arrête. Mais suite à la survenue de certaines exceptions, il arrive que le programmeur veuille continuer le programme en exécutant d'autres instructions. Dans ce genre de situations, il devient donc nécessaire de capturer l'exception. Capturer une exception consiste à prévoir et à exécuter des instructions prédéfinies en cas de survenue d'une exception préalablement identifiée.

Traditionnellement, une exception Java est capturée en utilisant trois blocs d'instructions définis par les mots-clés try{…}, catch(){…}, finally{…}. Ci-dessous décrit le rôle de chaque bloc.

Le bloc try{..}

Ce bloc vise à spécifier les séquences d'instructions susceptibles de jeter une exception : lecture d'un fichier, connection à une base de données, connection à un réseau, réaliser une opération arithmétique comme la division, conversion de type d'objet, etc.. Toutes les exceptions peuvent être jetées à l'intérieur d'un bloc try{…}, qu'il s'agisse des exceptions natives Java ou des exceptions conçues par l'utilisateur, que ces exceptions soient des exceptions contrôlées ou des exceptions non contrôlées.

Le bloc catch () {…} :

Le bloc catch permet au programmeur de définir des instructions à exécuter en cas de survenue d'une exception. Il peut s'agir de l'envoi d'un message personnalisé, de l'exécution d'une instruction alternative à celle qui a renvoyé l'exécution, ou toute autre instruction choisie par le développeur.

Le bloc finally(){…} :

Il permet au programmeur de spécifier d'autres instructions de natures différentes de celles déjà spécifiées dans le bloc catch(). Le bloc finally est un bloc optionnel lors de la capture d'une exception. Dans certaines situations, le bloc finally est utilisé pour libérer les ressources qui étaient retenues par le programme. Ex : fermer les flux ou les connections

ouvertes, libérer les ressources mémoires, etc… Le bloc finally n'est donc pas obligatoire dans tous les cas de capture d'exception.

L'exemple ci-dessous montre un cas simple pour capturer une exception survenue lors de la lecture d'un fichier qui peut potentiellement être inexistant.

```
package com.tuto.exception;
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args)  {

        // Définition du fichier à lire
        String fileToRead="myFile.txt";
        // Défintion du bloc try:
        try {
            FileReader fileReader = new FileReader(fileToRead);
            int line=fileReader.read(); // Lecture de la première ligne du
fichier (1er appel de la méthode read())
            System.out.println(line);
            fileReader.close();
        }
        // Définition du bloc catch
        catch(Exception e){
            System.out.println("Une erreur est survenue dans l'exécution du
programme");
        }
        finally {
            System.out.println("L'exécution du traitement est arrêtée");
        }

    }
}
```

Output :

```
Une erreur est survenue dans l'exécution du programme
L'exécution du traitement est arrêtée
```

Dans cet exemple, nous avons spécifié les trois blocs d'instructions try/catch/finally.

Dans le bloc try, nous avons défini quelques instructions susceptibles de renvoyer des exceptions. Il s'agit en l'occurrence de l'appel de la méthode read() qui active l'objet FileReader pour lire le contenu. Et en cas de survenu d'exception, nous avons spécifié dans le bloc catch, une instruction qui indique simplement que « Une erreur est survenue dans l'exécution du programme ».

S'agissant du bloc catch, comme on peut le remarquer, l'exception e que nous avons capturée est de type Exception. Il peut donc s'agir de n'importe quelle exception dérivée de la classe Exception.

Cependant, cet exemple de capture d'exception reste très standard et ne permet pas d'adapter les instructions à chaque type d'exception. Parfois, dans la gestion des exceptions, il apparait judicieux de capturer les exceptions d'intérêt dans des blocs catch dédiés afin de pouvoir adapter les instructions. En effet, il est possible de spécifier autant de blocs catch qu'on souhaite lors de la capture des exceptions. L'exemple ci-dessous spécifie deux blocs de catch afin d'adapter les messages à chaque exception capturée.

```
package com.tuto.exception;
import java.io.*;
import java.util.concurrent.ExecutionException;

public class Main {
    public static void main(String[] args)  {

            // Définition du fichier à lire
        String fileToRead="myFile.txt";
             // Défintion du bloc try:
        try {
            FileReader fileReader = new FileReader(fileToRead);
            int line=fileReader.read(); // Lecture de la première ligne du
fichier (1er appel de la méthode read())
            System.out.println(line);
            fileReader.close();
        }
        // Définition des blocs catch
        catch(FileNotFoundException e1){
            System.out.println("Le fichier à lire n'existe pas");
            System.out.println(e1.getMessage());
        }
        catch( IOException e2){
            System.out.println("Une erreur est survenue dans l'exécution du
programme");
            System.out.println(e2.getMessage());
        }
    }
}
```

Output :

```
Le fichier à lire n'existe pas
myFile.txt (Le fichier spécifié est introuvable)
```

Dans l'exemple ci-dessus, nous avons indiqué deux blocs catch. Le premier est destiné à spécifier les instructions à exécuter en cas de survenu d'une exception de type FileNotFoundException. Le second est destiné à spécifier les instructions à exécuter en cas de survenue de toute autre exception de type IOException. A noter que l'exception FileNotFoundException est elle-même une exception de type IOException. Le fait de la capturer dans un bloc catch spécifique vise simplement à adapter les instructions pouvant lui correspondre.

Notons également qu'il est possible de regrouper plusieurs exceptions dans un même bloc catch et de leur faire correspondre une même séquence d'instructions. L'exemple ci-dessous illustre le regroupement des exceptions suivant les blocs catch.

```
package com.tuto.exception;
import java.io.*;
public class Main {
    public static void main(String[] args)  {

            // Définition du fichier à lire
        String fileToRead="myFile.txt";
             // Défintion du bloc try:
        try {
            FileReader fileReader = new FileReader(fileToRead);
            int line=fileReader.read(); // Lecture de la première ligne du
fichier (1er appel de la méthode read())
            System.out.println(line);
            fileReader.close();
        }
        // Définition du bloc catch avec plusieurs Exception
        catch(IOException | IllegalArgumentException | NullPointerException e){
            System.out.println("Une erreur est survenue dans l'exécution du
programme");
            System.out.println(e.getMessage());
        }
    }
}
```

Output :

```
Une erreur est survenue dans l'exécution du programme
myFile.txt (Le fichier spécifié est introuvable)
```

Dans l'exemple ci-dessus, nous avons spécifié dans un même bloc catch, trois execptions IOException, IllegalArgumentException et NullPointerException. Comme on peut le constater, lorsqu'on spécifie plusieurs exceptions dans un même bloc catch, ces exceptions sont alors séparées par l'opérateur logique « ou » symbolisé par « | ».

11 GESTION DES LOGS

11.1 Généralités

Les logs (les journaux) sont des moyens permettant de tracer les évènements significatifs qui surviennent durant l'exécution d'un programme. Le logging consiste à ajouter des blocs d'instructions dans le code afin de fournir à l'utilisateur des informations pertinentes sur les différentes étapes d'exécution du programme. Il permet notamment d'envoyer des messages, de tracer les erreurs et les exceptions, de fournir des informations fonctionnelles comme la valeur d'une variable, mais aussi des informations techniques comme l'horodatage des traitements, les durées d'exécution des traitements, des informations sur les utilisateurs ayant lancé la requête, les adresses IP, etc. Ces différentes informations peuvent par la suite être exploitées afin de monitorer le traitement ou être utilisées à d'autres fins utiles.

Tout comme la gestion des exceptions, la gestion des logs est indispensable dans le processus de développement d'un programme dans la mesure où elle facilite grandement le suivi et la maintenance des applications informatiques.

11.2 Les principaux frameworks de logging en Java

Il existe de nombreux frameworks de logging permettant de gérer les logs dans le langage Java. Les plus connus et les plus utilisés restent : java.util.logging (JUL), Log4j2, LogBack, SLF4j et Apache Common Logging (ACL). Ci-dessous un aperçu rapide sur chaque framework.

11.2.1 Le framework java.util.logging (JUL)

Java.util.logging (JUL) est le framework natif de logging en Java. C'est une API qui a été intégrée à Java depuis la version 1.4 du JDK. L'avantage du framework JUL est sa simplicité d'utilisation. Car il nécessite moins de configuration. Cependant cette API offre moins de fonctionnalités et de flexibilités par rapport à d'autres frameworks comme Log4j2 ou LogBack notamment.

11.2.2 Le framework Log4j2

Log4j2 est un framework open-source de logging qui a succédé à Log4j le framework historique arrivé en fin de vie en 2015. Log4j2 est un framework très flexible et permet d'avoir un contrôle total sur le comportement de logging de votre application, d'orienter les logs vers différentes sorties. Grâce à sa simplicité d'usage, ses fonctionnalités et sa flexibilité, Log4j2 s'impose comme un framework de référence pour le logging Java.

11.2.3 Le framework LogBack

Tout comme Log4j2, le framework LogBack est l'un des successeurs du framework historique Log4j. C'est un framework qui offre les mêmes flexibilités que Log4j2 et bénéficie presque de la même popularité d'usage.

11.2.4 Le framework SLF4J

Le framework SLF4J (Simple Logging Facade for Java) est un framework qui offre une classe d'abstraction permettant de logger avec d'autres frameworks de loggings comme JUL, LogBack, Log4j2, etc… L'usage de SLF4J est souvent recommandé lorsque votre application sera utilisée par des systèmes tiers qui n'ont pas les mêmes systèmes logging. SLF4J permet d'assurer la portabilité de votre code entre différents systèmes de logging. Il permet de découpler le code et le logging. Par exemple, lorsque votre code doit être exécuté dans un environnement utilisant le logging Log4J2, il suffit simplement de modifier les configurations de logging pour pointer vers ce système sans modifier votre code. De même, si le code doit être exécuté dans un environnement utilisant le logging JUL, il suffit simplement de modifier la configuration pour pointer vers ce système de logging. SLF4J offre donc un cadre unifié pour gérer dynamiquement les différents systèmes de logging sans avoir à modifier le code source.

11.2.5 Apache Common Logging

Apache Common Logging (anciennement connu sous le nom de Jakarta Common Logging JCL) est un framework d'abstraction tout comme le framework SLF4J et qui vise à harmoniser les autres frameworks sur un socle commun, tout en éliminant les éventuelles dépendances applicatives pouvant être liées à un framework spécifique. Le JCL se positionne comme un alternatif à SLF4J dans une application faisant appel à plusieurs frameworks de logging.

Dans ce document, nous allons passer en revue quelques-uns des frameworks ci-dessus présentés en montrant leur mode d'utilisation.

11.3 Les principaux composants d'un framework de logging

Un framework de logging fait intervenir trois principaux composants que sont le Logger, l'Appender et le Formatter. Chaque composant est représenté par une classe dans le framework. Notons qu'en plus de ces trois composants, la plupart des framework fournissent une fonctionnalité de filtre à travers une composant additionnel qu'est le Filter. Ci-dessous les détails sur chacun des composants.

11.3.1 Le Logger

C'est la composante de base de toute framework de logging. Il sert à instancier la classe dont les méthodes seront invoquées pour envoyer les logs. Chaque framework dispose de sa propre classe Logger. Nous reviendrons plus tard en détails sur l'instanciation et l'usage de la classe Logger pour certains frameworks.

11.3.2 Le layout (Formatter)

Le layout (encore appelé formatter) est le composant qui permet de formater (mettre en forme) les lignes de logs générés par l'application. Les différents paramètres définissant le formatage à adopter sont souvent spécifiés dans un fichier de configuration. Mais ils peuvent aussi être spécifiés directement dans le code après l'instanciation de la classe Logger. Cependant la bonne pratique reste l'utilisation d'un fichier de configuration car il permet de modifier les paramètres de logging sans avoir à retoucher le code source.

11.3.3 L'Appender

L'appender est le composant qui permet d'orienter les logs générés vers différentes sorties : console, fichier, base de données, email, server http,...).

11.3.4 Le Filter

Comme son nom l'indique, la fonctionnalité Filter permet de faire passer toutes les lignes de log à tamis et de ne retenir que celles qui répondent aux critères spécifiés. Par exemples, retenir toutes les lignes de log de niveau INFO, WARN, ERROR, etc. Le Filter n'est pas une composante obligatoire dans le framework de logging. Mais il peut s'avérer utile dans de nombreuses situations notamment celles où l'utilisateur souhaite un traitement plus détaillé des logs selon un ou plusieurs critères préalabalement définis.

11.4 Les niveaux de logging : Level

Le niveau (LEVEL) indique le degré de gravité de la ligne de log généré pour chaque évènement loggué. Il va d'une simple trace à une erreur plus grave entraînant l'arrêt de l'exécution de l'application. Ci-dessous les détails de quelques niveaux standards de logging Java.

TRACE : Un log de niveau TRACE permet de suivre l'exécution de l'application au niveau le plus fin. Le niveau trace n'est souvent utile qu'aux développeurs de l'application.

DEBUG : Moins détaillé que TRACE, ce niveau permet aux développeurs et aux maintenanciers de l'application de suivre de manière plus fine l'exécution d'un programme. Son rôle est d'aider à détecter l'origine des problèmes en cas de survenue d'une erreur dans une application complexe, permettant de suivre l'exécution au niveau le plus fin.

INFO : Les logs de niveau INFO vise à fournir à l'utilisateur des informations sur le déroulement des opérations de traitement prévus dans le programme. Les logs de niveau INFO sont généralement de nature technico-fonctionnelle. Ex : Création de connection réussie, chargement des données terminées, nombre de lignes traitées, etc…

WARN : Les logs de niveau WARN (WARNING) sont des lignes de logs permettant d'avertir l'utilisateur qu'une opération prévue dans la séquence des instructions ne s'est pas déroulée correctement. Contrairement aux niveaux TRACE, DEBUG et INFO, les logs de type WARNING comportent un certain degré de gravité qui doivent attirer l'attention de l'utilisateur. En effet, même si tous les logs de niveau WARN n'ont pas de conséquence immédiate sur le processus d'exécution du programme, leur survenue peut changer la nature de certains résultats escomptés. Par exemple, pour définir la valeur d'une variable, on peut être amené à lancer une requête https sur un serveur. Et lorsque le serveur ne répond pas, au lieu d'arrêter le traitement, on peut choisir d'attribuer une valeur par défaut à la variable. Le fait que l'envoi de la requête https ait échoué, cela doit être loggué comme un WARN car nous avons une valeur de rechange pour la variable. Mais il va de soi que la valeur de la variable aurait pu être différente de la valeur par défaut si la requête https avait réussi. Aussi, les logs de niveau WARN peuvent être la cause d'erreurs plus graves, plus tard dans le reste du programme. Par exemple, lorsque plusieurs évènements de type WARN surviennent successivement, cela peut entraîner la survenue d'un évènement de type ERREUR. C'est pourquoi, les utilisateurs sont invités à considérer les logs de niveau WARN avec beaucoup plus de rigueur.

ERROR : Les logs de niveaux ERROR informent sur des évènements plus graves et qui compromettent la poursuite de l'exécution du programme. Les logs de niveau ERROR sont par exemples utilisés pour des évènements comme la lecture de fichier inexistant, problème de connection à une base de données JDBC, une exception survenue suite à une opération arithmétique de division par zéro, ou toute erreur et exception capturée lors de l'exécution du programme.

FATAL : Le niveau de FATAL est un descriptif propre au framework Log4j2. Ce niveau vise à renvoyer les évènements qui exigent un arrêt net du programme. Du point de vue Log4j2, ce niveau est le plus grave des niveaux de logging. Il est utilisé par exemple pour des évènements comme l'insuffisance de mémoire, problème de droit d'un user sur un FileSystem, un répertoire ou un fichier, etc…

Remarquons que chaque framework de logging peut définir un niveau de logging spécifique pour capturer un aspect particulier des évènements. Par exemple le framework JUL ajoute les niveaux suivants.

CONFIG : niveau de log permettant d'indiquer l'évènement survenu est une configuration.

FINEST, FINER et FINE : niveaux de log permettant de donner les détails les plus fins sur l'exécution du programme. Ces trois niveaux correspondent aux niveaux TRACE et DEBUG du package Log4j2.

SEVERE : niveaux permettant de capturer les évènements graves lors de l'exécution du programme à l'image du niveau ERROR du framework Log4j2.

11.5 Templte de configuration logging : le fichier .properties et .xml

Comme nous l'avons déjà montré, tout framework de logging comporte trois principaux composants que sont la classe Logger, le formatter (layout) et l'Appender. On peut également y ajouter une quatrième composante, bien qu'optionnelle. Il s'agit du composant Filter. A noter que seule la classe Logger est obligatoirement instanciée dans le code source du programme Java. Bien entendu, les autres composants (Formatter, Appender et Filter) peuvent également être appelés et instanciés dans le code source du programme, car il existe des classes correspondant à chacun des composants. Toutefois, il est de pratique courante de spécifier le Formatter et l'Appender dans un fichier de configuration et de charger automatiquement cette configuration au moment de l'exécution du code. L'avantage d'utiliser un fichier de configuration est qu'on peut modifier les paramètres à tout instant sans toucher au code source. De ce fait, il n'est pas nécessaire de livrer un nouveau package.

La plupart des frameworks permettent de spécifier les Formatters et les Appenders en choisissant entre deux types de fichiers : un fichier .properties ou un fichier .xml. Pour rappel, un fichier .properties est un fichier dans lequel on déclare une variable (appelée propertie) à laquelle on assigne une valeur en utilisant l'opérateur =. Par exemple, le fichier .properties peut contenir une ligne spécifiée comme suit : myVar=myValue. Quant à un fichier de configuration de type xml, les variables (properties) sont spécifiées sous formes de balises. Par exemple le fichier de configuration .xml peut contenir la variable myVar définie comme suit : < name="myVar" value="myValue">. Voici par exemples comment se présente un fichier .properties et un fichier .xml pour le framework Log4j2[24].

11.5.1 Template de fichier de configuration .properties

```
# Nom de la configuration
name=PropertiesConfig
# Choix des noms des appenders (choix libre et non obligatoire: ex : console,
myConsole,file, MyFile etc..)
appenders = console, file
# Détails de l'Appender nommé console
appender.console.type = Console
appender.console.name = LogToConsole
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n
# Détails de l'Appender nommé file
appender.file.type = File
appender.file.name = LogToFile
# Fichier stockant les logs
appender.file.fileName=logs/file.log
```

[24] Ici, il s'agit simplement d'un aperçu illustrant la présentation du contenu d'un fichier de configuration .properties et .xml. Nous présenterons plus tard en détails sur des cas concrets de fichiers de configuration de logging dans les sections qui vont suivre.

```
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1}
- %msg%n
# Appel du console appender
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
rootLogger.appenderRef.file.ref = LogToFile
```

11.5.2 Template de fichier de configuration .xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="DEBUG">
    <!-- Met le niveau de log à DEBUG -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
            <!-- Définit un format pour les lignes de log pour cet Appender-->
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36}
- %msg%n" />
        </Console>
        <!-- Définit un autre Appender nommé LogToFile -->
        <File name="LogToFile" fileName="logs/app.log">
            <PatternLayout>
                <!-- Définit un format pour les lignes de log pour cet
Appender-->
                <Pattern>%d %p %c{1.} [%t] %m%n</Pattern>
            </PatternLayout>
        </File>
    </Appenders>
    <Loggers>
        <!-- Définit une configuration de log pour les classes du package
com.tuto-->
        <Logger name="com.tuto" level="debug" additivity="false">
            <!-- Et assocoe les Appender LogToFile et LogToConsole au loggger
du package com.tuto-->
            <AppenderRef ref="LogToFile" />
            <AppenderRef ref="LogToConsole" />
        </Logger>
    </Loggers>
</Configuration>
```

11.6 Logging avec le framework java.util.logging (JUL)

Comme déjà indiqué, java.util.logging (JUL) est le framework natif de logging Java. Cette section a pour but de montrer les principales caractéristiques de ce framework ainsi que ses modes d'utilisation. Mais pour pouvoir illustrer les différents aspects du logging avec ce framework, nous allons partir d'un template de code source définit ci-dessous.

11.6.1 Code source d'illustration : Code source CS03

Ce code source sera utilisé pour illustrer les différents composants du framework JUL.

Code source : CS03

```
package com.tuto.logging;

import java.util.logging.Logger;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;


public class Main {
    private static final Logger LOGGER =
Logger.getLogger(Main.class.getName());

    public static void main(String[] args)  {


        LOGGER.info("Début d'exécution de la méthode main");
        try {
            LOGGER.fine(" Début de création de l'objet BufferedReader ");
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            LOGGER.fine(" Fin de création de l'objet BufferedReader ");

            LOGGER.fine(" Début de récupération de l'entrée utilisateur ");
            System.out.print("Saisissez votre nom, svp : "); // Ex : Mister
Kelly
            String nom = br.readLine();
            LOGGER.fine(" Fin de récupération de l'entrée utilisateur ");

            LOGGER.fine(" Début envoi salutation ");
            System.out.println("Hello " + nom); br.close();
            LOGGER.fine(" Fin envoi salutation ");
        }

        catch (IOException e) {
            LOGGER.severe("Un problème est survenue lors de la lecture de la
valeur entrée par l'utilisateur");
        }

        LOGGER.info("Fin d'exécution de la méthode main");
    }
}
```

L'objectif de ce code source CS03 est de logguer quelques évènements avec plusieurs niveaux de log en utilisant le framework JUL. Pour cela, nous avons mis en place plusieurs instructions simples visant à afficher une salutation pour une personne dont le nom est saisi à l'écran par l'utilisateur. La récupération du nom saisi par l'utilisateur se fait en plusieurs étapes. Dans un premier temps, nous créons un objet de type BufferReader dont l'argument d'instanciation est un objet InputSreamReader() qui lui-même prend en paramètre la classe System.in qui invite l'utilisateur à entrer une valeur à l'écran. C'est après l'instanciation de l'objet BufferReader que nous appelons la méthode readLine() qui a pour but de transformer la valeur entrée par l'utilisateur en une valeur String. Enfin nous affichons le nom entré par l'utilisateur sous forme de salutation en ajoutant le mot « Hello ».

Dans l'exemple ci-dessous, la première étape de logging est d'importer la classe Logger. Cela se fait à travers l'instruction import java.util.logging.Logger. Ensuite, pour mettre en œuvre le logging on instancie la classe Logger avec l'instruction :

```
private static final Logger LOGGER = Logger.getLogger(Main.class.getName());
```

Cette instanciation est faite comme un attribut de la classe dont les évènements sont loggués. C'est pourquoi l'objet LOGGER est défini à l'extérieur de toutes les méthodes de la classe.

Après l'instanciation de l'objet LOGGER, on peut maintenant appeler les différentes méthodes correspondant aux différents niveaux de log.

Dans le code ci-dessus, comme on peut le remarquer, à chaque étape de l'exécution de ce code, nous affichons une ligne de log correspondant à une instruction. Nous utilisons trois niveaux de logs correspondant chacun à une instruction spécifique dans le code. Les trois niveaux de log utilisés sont : FINE, INFO, SEVERE. Le niveau FINE est utilisé pour fournir un détail plus fin sur les étapes d'exécution du programme. Le niveau INFO est utilisé pour fournir les informations sur les évènements les plus marquants dans l'exécution du traitement et le niveau SEVERE est utilisé pour informer sur la survenue d'information plus grave nécessitant parfois l'arrêt de l'exécution du programme.

Les différents niveaux de log définis pour le framework JUL sont CONFIG, FINEST, FINER, FINE, INFO et SEVERE. En plus de ces niveaux classiques, il existe deux niveaux spéciaux que sont ALL et OFF qui permettent respectivement d'activer et désactiver tous les niveaux de log précédemment indiqués.

11.6.2 Envoi de logs dans la console

Le code source CS03 montre comment instancier la classe Logger du framework et envoyer des lignes de logs dans la console de plusieurs niveaux logs. A noter que les logs du framework JUL sont envoyés par défaut sur la console. Mais ce comportement peut être modifié pour envoyer les logs sur d'autres terminaux comme les fichiers, etc..

NB : Avant d'exécuter le code CS03, ouvrons d'abord le fichier logging.properties. Ce fichier est généralement situé dans le répertoire ${JAVA_HOME}/jre/lib/logging.properties. Il peut aussi être situé dans le dossier conf du repertoire d'installation du JDK. Par exemple,

pour notre cas, il s'agit du répertoire C:\Program Files\Java\jdk-20\conf. En ouvrant le fichier logging.properties, on constate que, par défaut, que le niveau de logging est fixé à INFO. Cela est visible avec la propertie .level=INFO. Ce paramètre signifie que par défaut, tous les logs de niveau moins grave INFO ne seront pas affichés. Pour modifier ce comportement, il suffit de changer la valeur de la ligne .level= INFO et mettre .level= ALL. Le niveau ALL permet d'afficher tous les niveaux de logs quel que soit leur degré de gravité. Pour le framework JUL, cela va de FINEST (niveau le plus bas) à SEVERE (niveau le plus élevé). Noter aussi qu'on peut parcourir le fichier logging.properties et remplacer tous les niveaux INFO par ALL pour tous les handlers présents.

Pour s'assurer que les logs sont orientés uniquement sur la console, il faut vérifier que la propertie handlers ait la valeur : java.util.logging.ConsoleHandler. Au final, voici comment devrait se présenter le contenu du fichier logging.properties.

C:\Program Files\Java\jdk-20\conf\logging.properties

```
handlers= java.util.logging.ConsoleHandler
.level= ALL
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

Après ce paramétrage et en exécutant le code CS03, on obtient la sortie suivante sur la console.

Output :

```
mai 09, 2023 10:18:49  com.tuto.logging.Main main
INFO: Début d'exécution de la méthode main
mai 09, 2023 10:18:49  com.tuto.logging.Main main
FINE:  Début de création de l'objet BufferedReader
mai 09, 2023 10:18:49  com.tuto.logging.Main main
FINE:  Fin de création de l'objet BufferedReader
mai 09, 2023 10:18:49  com.tuto.logging.Main main
FINE:  Début de récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
Hello Kevin
mai 09, 2023 10:18:59  com.tuto.logging.Main main
FINE:  Fin de récupération de l'entrée utilisateur
mai 09, 2023 10:18:59  com.tuto.logging.Main main
FINE:  Début envoi salutation
mai 09, 2023 10:18:59  com.tuto.logging.Main main
FINE:  Fin envoi salutation
mai 09, 2023 10:18:59  com.tuto.logging.Main main
INFO: Fin d'exécution de la méthode main
```

11.6.3 Envoi des logs dans la console et dans un fichier

11.6.3.1 Logging avec le formatage par défaut

Pour envoyer les logs dans un fichier, on doit spécifier l'appender FileHandler dans le fichier logging.properties. Ensuite, on doit spécifier le chemin vers le fichier qui doit recueillir les lignes de logs. En partant du fichier logging.properties déjà configuré pour la console dans la section précédente, on peut apporter les modifications suivantes et ajouter la redirection des logs vers un fichier logs/file.log situé à la racine du projet Java.

C:\Program Files\Java\jdk-20\conf\logging.properties

```
# Définition des handlers
handlers= java.util.logging.ConsoleHandler,java.util.logging.FileHandler
# Fixer le niveau de log à ALL
.level= ALL

#Propriétés du handler pour le console
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

#Propriétés du handler pour le fichier
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern =
C:\\MY_JAVA_PROJECTS\\javaTuto\\logs\\file.log
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

Après avoir modifié le fichier logging.properties et en exécutant le code CS03, la sortie dans la console et dans le fichier file.log se présente comme suit.

Output :

```
mai 10, 2023 1:06:04 com.tuto.logging.Main main
INFO: Début d'exécution de la méthode main
mai 10, 2023 1:06:05 com.tuto.logging.Main main
FINE:  Début de création de l'objet BufferedReader
mai 10, 2023 1:06:05 com.tuto.logging.Main main
FINE:  Fin de création de l'objet BufferedReader
mai 10, 2023 1:06:05 com.tuto.logging.Main main
FINE:  Début de récupération de l'entrée utilisateur
mai 10, 2023 1:06:07 com.tuto.logging.Main main
FINE:  Fin de récupération de l'entrée utilisateur
mai 10, 2023 1:06:07 com.tuto.logging.Main main
FINE:  Début envoi salutation
mai 10, 2023 1:06:07 com.tuto.logging.Main main
FINE:  Fin envoi salutation
mai 10, 2023 1:06:07 com.tuto.logging.Main main
INFO: Fin d'exécution de la méthode main
```

Après l'exécution de ce code, ouvrir le fichier C:\MY_JAVA_PROJECTS\javaTuto\logs\file.log.1 et observer le contenu fichier. Il correspond aux mêmes lignes de logs que celles affichées dans le console.

11.6.3.2 Formatage des lignes de logs : utilisation des variables de formatage

Le formatage des lignes se base sur un certain nombre de variables internes appelées variables de formatage. Les valeurs de ces variables se présentent sous forme de codes numériques. Le framework JUL définit 6 codes distincts dont les descriptifs sont fournis ci-après.

- **1** : représente la date de génération de la ligne de log.
- **2** : permet d'indiquer la méthode ayant générée le code. Il peut également s'agir du nom du logger lui-même.
- **3** : permet d'indiquer le nom du logger. En général, ce nom est obtenu en appelant la méthode getLog() sur l'objet obtenu par l'instanciation de la classe Logger.
- **4** : ce code permer d'indiquer le niveau du log spécifié : FINE, INFO, WARN, SEVERE, etc.
- **5** : ce code renvoie le corps de message de log spécifié.
- **6** : ce code permet de renvoyer l'exception capturée par ligne de log. Ce code est souvent utilisé pour le niveau de log comme WARN et SEVERE

Pour utiliser un code de formatage spécifique, on doit indiquer la valeur entre le symboles % et $. Par exemple pour afficher la date de génération du log, on indique %1$. Pour afficher le niveau de gravité du log, on indique %4$,etc.

Ces codes numériques sont souvent utilisés en combinaison avec d'autres paramètres génériques spécifiés plutôt sous formes lettres alphabétiques. Il s'agit en particulier des codes %u, %g, %t,%s et %n. Ci-dessous un descriptif de chacune de ces variables internes.

- **%u** : cette variable est un numéro d'identification unique permettant de résoudre les conflits entre plusieurs process Java simultanés. Grâce à la variable %u, les logs générés par chaque process seront identifiables par un numéro unique. Cette variable est surtout utile lorsque plusieurs process Java écrivent dans le même fichier.
- **%g** : cette variable permet de générer un numéro pour chaque rotation du fichier dans lequel sont écrits les lignes de logs. En effet, chaque fois que le fichier atteint sa limite en termes de taille ou en termes de fréquence de rotation, un nouveau fichier est généré pour accueillir les nouvelles lignes de log.
- **%t** : variable permettant de spécifier le répertoire temporaire du système dans lequel les fichiers de logs sont stockés.
- **%s** : permet d'insérer un espace dans le formatage de la ligne de log. On peut utiliser cette variable pour insérer un espace entre deux informations. Par exemple entre les valeurs indiquées par les différents codes de formatage précédemment spécifiés.
- **%n** : permet d'insérer un retour à la ligne dans la ligne de log affiché.

L'exemple ci-dessous montre l'utilisation de quelques variables de formatage spécifiées dans le fichier logging.properties

C:\Program Files\Java\jdk-20\conf\logging.properties

```
# Définition des handlers
handlers= java.util.logging.ConsoleHandler,java.util.logging.FileHandler
# Fixer le niveau de log à ALL
.level= ALL

# Format du message
java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n


#Propriétés du handler pour le console
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

#Propriétés du handler pour le fichier
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern =
C:\\MY_JAVA_PROJECTS\\javaTuto\\logs\\file.log
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

Dans cet exemple, nous utilisons les variables de formatage pour définir le format du message envoyé par java.util.logging.SimpleFormatter. En effet, le format utilisé ici est la combinaison des codes 4, 5, 1 et des variables %s, %t et %n. La spécification du format est donc =%4$s: %5$s [%1$tc]%n.

En utilisant ce formatage et en réexécutant le code CS03, nous obtenons un fichier de log dont le contenu se présente comme suit :

```
INFO: Début d'exécution de la méthode main [jeu. mai 11 09:32:46 CEST 2023]
FINE:  Début de création de l'objet BufferedReader  [jeu. mai 11 09:32:46 CEST
2023]
FINE:  Fin de création de l'objet BufferedReader  [jeu. mai 11 09:32:46 CEST
2023]
FINE:  Début de récupération de l'entrée utilisateur  [jeu. mai 11 09:32:46
CEST 2023]
FINE:  Fin de récupération de l'entrée utilisateur  [jeu. mai 11 09:32:52 CEST
2023]
FINE:  Début envoi salutation  [jeu. mai 11 09:32:52 CEST 2023]
FINE:  Fin envoi salutation  [jeu. mai 11 09:32:52 CEST 2023]
INFO: Fin d'exécution de la méthode main [jeu. mai 11 09:32:52 CEST 2023]
```

11.7 Logging avec le framework Log4j2

Cette section est consacrée au logging en utilisant le framework Log4j2. Contrairement à JUL qui est un framework natif Java, Log4j2 est une librairie externe dont l'utilisation nécessite d'abord une installation dans le classpath du programme en tant que dépendance.

11.7.1 Chargement de la librairie Log4j2

La librairie Log4j2 se présente sous forme d'archive téléchargeable et installable dans le classpath du programme. Java offre plusieurs façons d'installer les dépendances et les librairies externes. On peut procéder soit par une installation manuelle, soit par une installation automatisée à travers l'utilisation des outils de gestion de dépendances comme Maven, Ivy et Gradle. Nous reviendrons plus tard sur l'utilisation de Maven pour la gestion des dépendances dans un projet Java[25]. Dans la présente section, nous montrons la méthode d'installation manuelle de la dépendance Log4j2.

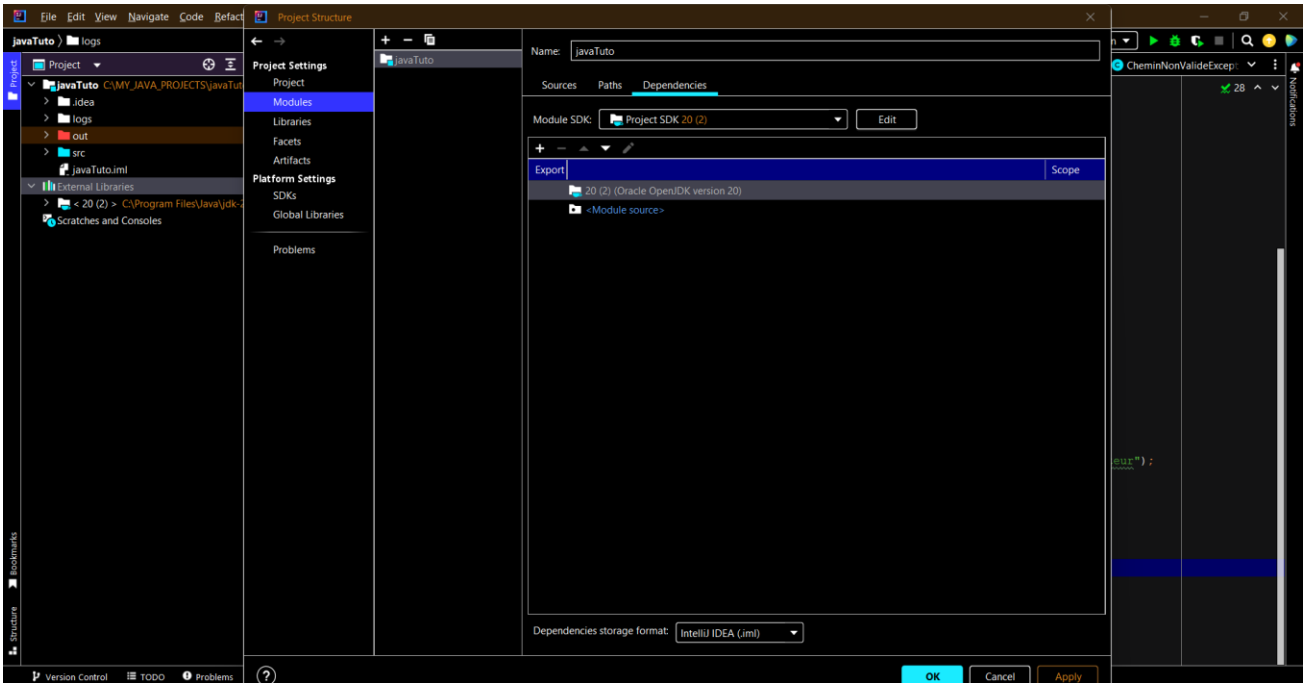
Pour télécharger et installer manuellement Log4j2, suivre les étapes suivantes :

- se rendre sur la page https://logging.apache.org/log4j/2.x/download.html. Et Choisir un package parmi les différentes formes d'archives disponibles. Par exemple, choisir l'archive zip. Ex : apache-log4j-2.20.0-bin.zip
- Télécharger cette archive, extraire et déposer les fichiers jars qu'elle contient dans le dossier lib de votre installation Java ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\lib. Le plusieurs fichiers .jar. Nous avons besoin de deux fichiers jar spécifiques que sont log4j-core-2.xx.y.jar et log4j-api-2.xx.y.jar. Ex : log4j-core-2.20.0.jar et log4j-api-2.20.0.jar.
- Nous allons ajouter ces deux fichiers dans le classpath de notre application. Cet ajout peut être directement fait à partir des menus de votre IDE. Par exemple, pour Intellij IDEA, l'ajout des fichiers jar se fait comme suit :

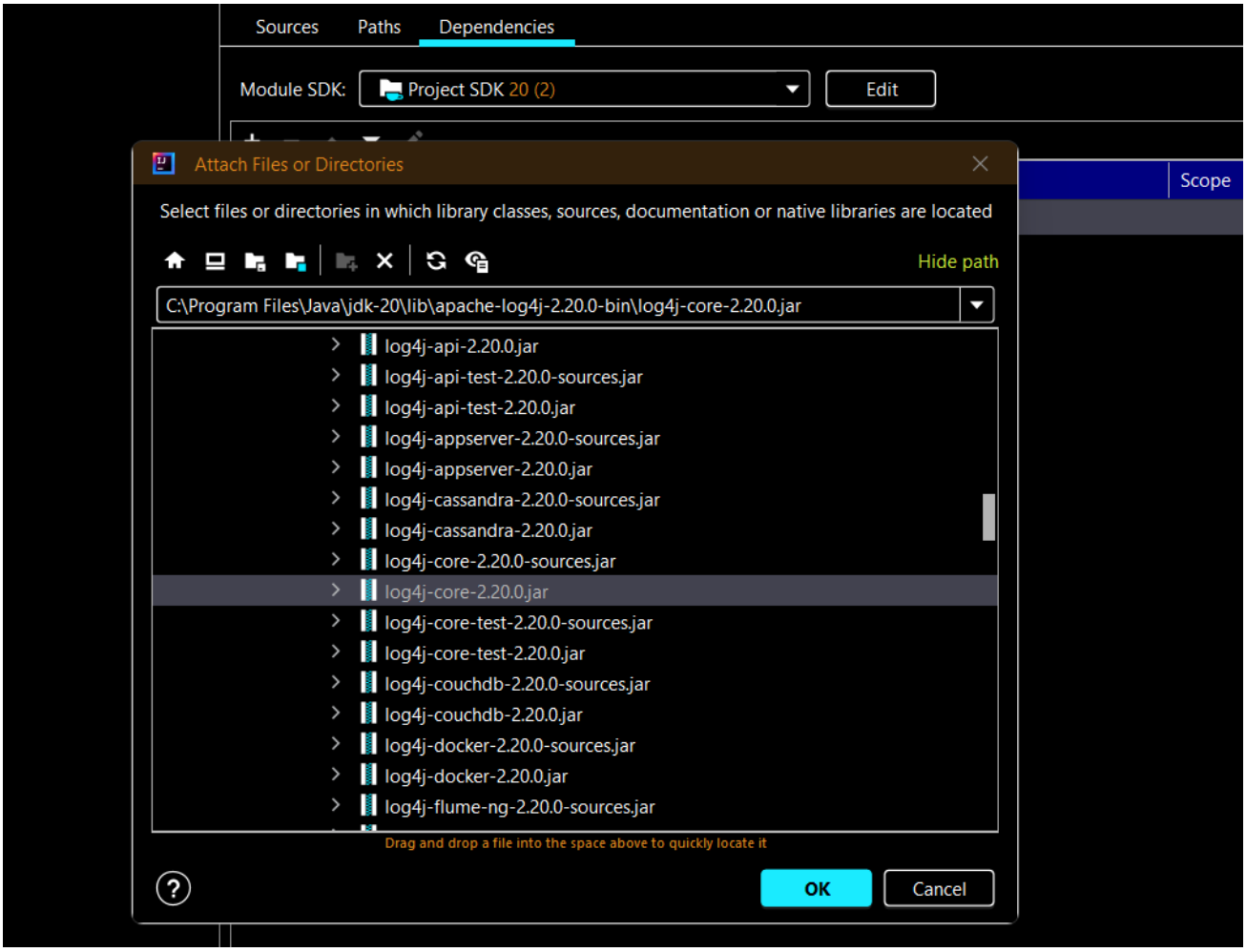
1- Dans la barre des menus, cliquer sur File >Project Structure >Project Settings> Modules > Dependencies.

[25] Pour ajouter la dépendance Log4j2 dans un projet Maven, ajouter ces spécifications dans le pom.xml

```
<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-api</artifactId>
   <version>2.20.0</version>
</dependency>
<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-core</artifactId>
   <version>2.20.0</version>
</dependency>
```



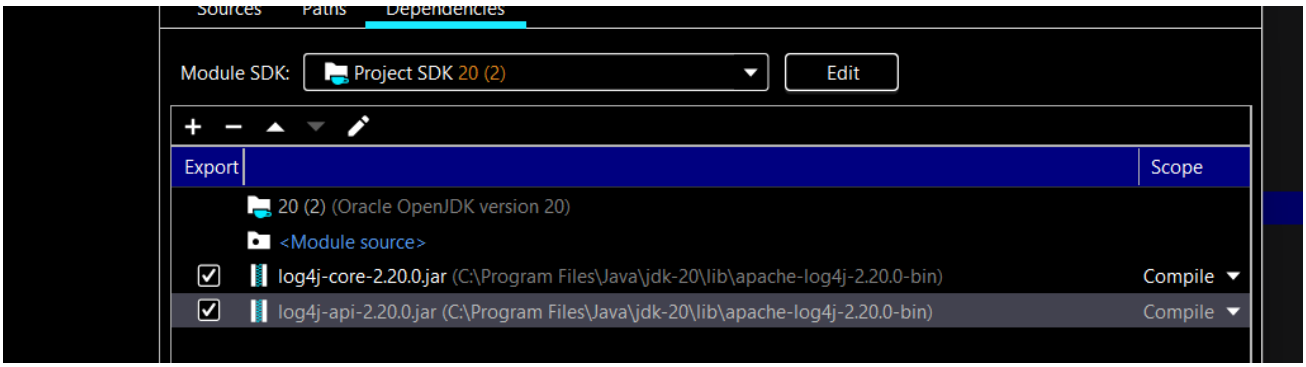
2- Cliquer le signe + (Add). Et choisir JARs or Directories. Et aller chercher le fichier log4j-core-2.xx.y.jar disponible dans le dossier lib dans lequel vous avez précédemment extrait les deux fichiers jars.



Cocher la case pour pouvoir ajouter le fichier aux dépendances. Devant la librairie sélectionnée, dans scope, dans la liste déroulante choisir Compile. Cliquer sur Ok pour valider.

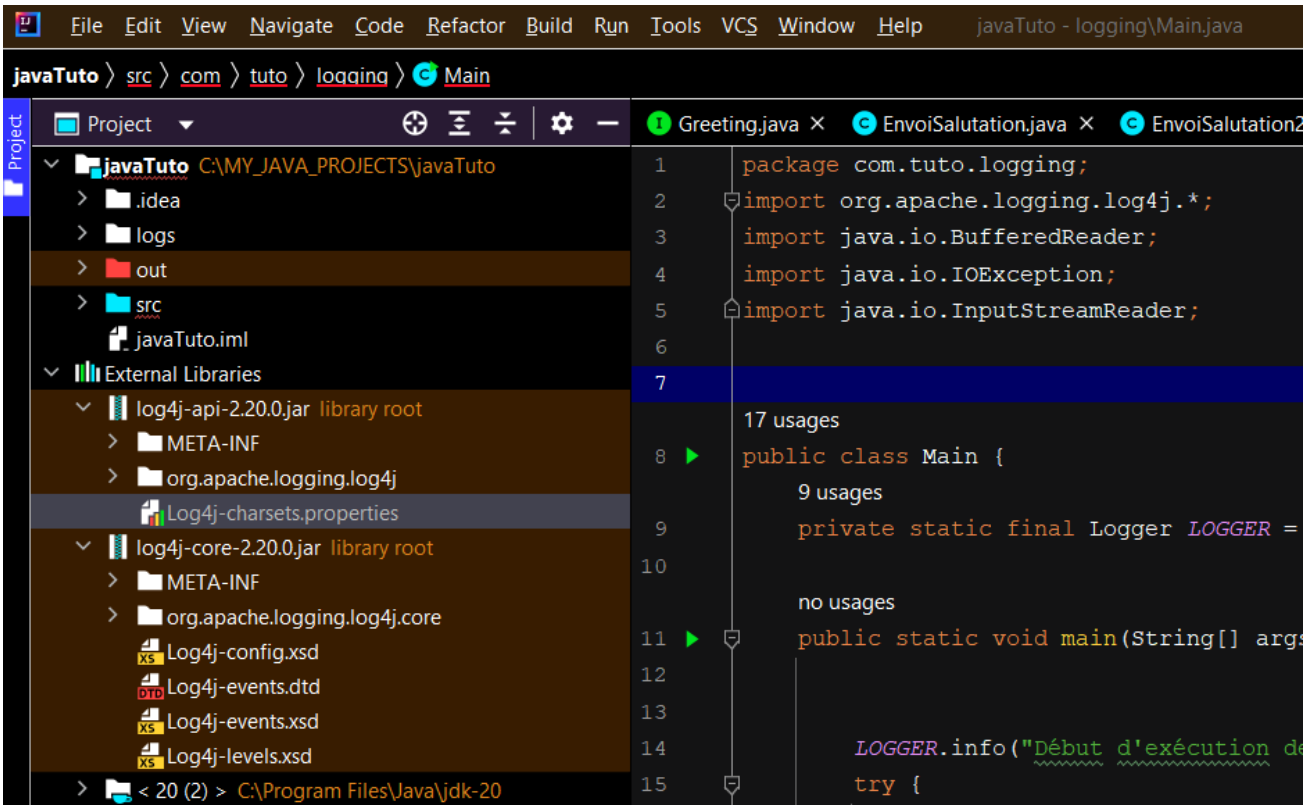
Cliquer une deuxième fois sur le symbole + (Add) et suivre la même démarche pour ajouter le fichier jar log4j-api-2.xx.y.jar disponible dans le même dossier lib.

Après l'ajout des deux fichiers jars, on a aura les deux lignes de dépendances comme suit :



3- Cliquer sur ok pour valider.

Les classes Logging sont désormais disponibles dans le classpath de votre application. Dans Intellij, les deux jars ainsi que leur contenu sont visibles à gauche dans External librairies. Cela montre que les dépendances Log4j2 sont bien installées. Voir la capture d'écran ci-dessous.



Comme on peut le remarquer, les deux fichiers jar contiennent à la fois des classes de logging mais aussi quelques templates de fichiers de configuration qui peuvent s'avérer utiles dans de nombreux cas.

11.7.2 Les principaux Appenders du framework Log4j2

Les Appenders sont les terminaux vers lesquels on peut orienter les logs. En plus de la console et les fichiers, Log4j2 offre plusieurs Appenders. Cette section a pour but de présenter, sans plus de détails, les principales classes Appenders Log4j2.

- **ConsoleAppender** : permet d'orienter les logs vers la console
- **FileAppender** : écrit les logs dans un fichier.
- **RollingFileAppender** : écrit les logs dans un fichier avec rotation c'est-à-dire en créant un nouveau fichier lorsque le fichier courant est rempli suivant un critère.
- **SMTPAppender** : envoie les logs vers une adresse email
- **KafkaAppender** : envoie les logs dans une file Kakfa
- **FlumeAppender** : envoie les logs vers Flume
- **CassandraAppender** : envoie les logs vers Cassandra

- **JDBCAppender** : envoie les logs dans une base de données en utilisant un driver JDBC.
- **HTTPAppender** : envoie les logs vers un endpoint HTTP

…

…

La page suivante présente les différents Appenders du framework Log4j2 :

https://logging.apache.org/log4j/2.x/manual/appenders.html

11.7.3 Code source d'illustration : Code source CS04

Pour illustrer le logging avec le framework Log4j2, nous reprenons le code source CS03 déjà utilisé dans la section de présentation du framework JUL. Nous modifions simplement la classe Logger en utilisant le package org.apache.logging.log4j à la place du package java.util.logging. Le code source ci-dessous montre les modifications apportées à la version initiale utilisée avec le framework JUL.

Code source : CS04

```
package com.tuto.logging;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;


public class Main {
    private static final Logger LOGGER =
LogManager.getLogger(Main.class.getName());

    public static void main(String[] args)  {


        LOGGER.info("Début d'exécution de la méthode main");
        try {
            LOGGER.debug(" Début de création de l'objet BufferedReader ");
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            LOGGER.debug(" Fin de création de l'objet BufferedReader ");

            LOGGER.debug(" Début de récupération de l'entrée utilisateur ");
            System.out.print("Saisissez votre nom, svp : "); // Ex : Kevin
            String nom = br.readLine();
            LOGGER.debug(" Fin de récupération de l'entrée utilisateur ");

            LOGGER.debug(" Début envoi salutation ");
            System.out.println("Hello " + nom); br.close();
            LOGGER.debug(" Fin envoi salutation ");
        }

        catch (IOException e) {
```

```
            LOGGER.error("Un problème est survenue lors de la lecture de la
valeur entrée par l'utilisateur");
        }

        LOGGER.info("Fin d'exécution de la méthode main");
    }
}
```

Pour pouvoir utiliser Log4j2, nous avons d'abord importé deux classes : LogManager et Logger. La classe LogManager permet d'instancier un objet de type Logger. L'objet de type Logger est créé avec l'instruction :

```
private static final Logger LOGGER =
LogManager.getLogger(Main.class.getName());
```

Cette instanciation est faite comme un attribut de la classe dont les évènements sont loggués. C'est pourquoi l'objet LOGGER est défini à l'extérieur de toutes les méthodes de la classe.

Après l'instanciation de l'objet LOGGER, on peut maintenant appeler les différentes méthodes correspondant aux différents niveaux de log.

Dans le code source CS04, comme on peut le remarquer, à chaque étape de l'exécution de ce code, nous affichons une ligne de log correspondant à une instruction. Nous utilisons trois niveaux de logs correspondant chacun à une instruction spécifique dans le code. Les trois niveaux de log utilisés sont : DEBUG, INFO, ERROR. Le niveau DEBUG permet de logguer de manière fine les instructions définies dans le programme. Le niveau INFO est utilisé pour fournir les informations sur les évènements les plus marquants dans l'exécution du traitement et le niveau ERROR est utilisé pour informer sur la survenue d'information plus grave nécessitant l'arrêt de l'exécution du programme.

Les différents niveaux de log définis pour le framework Log4j2 sont TRACE, DEBUG, INFO, WARN, ERROR et FATAL. En plus de ces niveaux classiques, il existe deux niveaux spéciaux que sont ALL et OFF qui permettent respectivement d'activer et de désactiver tous les niveaux de log précédemment indiquées.

11.7.4 Configuration du logging avec le fichier log4j2.properties

Pour pourvoir orienter les logs vers une sortie cible donnée (console, fichier, etc..) on doit d'abord définir les paramètres dans un fichier de configuration. Ce fichier peut être de type .xml ou de type .properties. Dans cette section, nous considérons le cas du fichier log4j2.properties.

11.7.4.1 Création du fichier log4j2.properties

Le fichier log4j2.properties peut être créé dans le dossier resources situé à l'intérieur du dossier src. Voici ci-dessous un exemple basique de fichier log4j2.properties.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.properties

```
status = warn
# Définir le console appender pour orienter les logs vers la console
appender.console.type = Console
appender.console.name = LogToConsole
# Appel du console appender
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
```

Le contenu de ce fichier est la définition la plus minimale pour orienter les lignes de logs dans la console, sans aucune option de formatage. Il s'agit en effet du format SimpleLayout qui est l'option de formatage la plus simple pour le logging Log4j2[26].

11.7.4.2 Appel du fichier log4j2.properties

Pour pouvoir utiliser le fichier log4j2.properties ci-dessus lors de l'exécution du code, il faut définir la propertie -Dlog4j.configurationFile dans les VM options Java. La propertie -Dlog4j.configurationFile doit pointer sur le log4j2.properties.

Pour définir la propertie -Dlog4j.configurationFile dans Intellij, cliquer dans le menu Run > Edit configuration. Cliquer >Modify options > Add VM options. Et ajouter la définition suivante :

```
-
Dlog4j.configurationFile=C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\log4j2
.properties
```

Attention : Parfois même si ce paramètre est défini dans les VM options, il arrive que le JRE fasse encore appel au fichier logging.properties défini par défaut pour votre installation Java. A noter que le fichier logging.properties est le fichier utilisé par le framework JUL pour logguer les évènements. Dans une telle situation, il est préférable de renommer le fichier logging.properties en par exemple logging_JUL_.properties[27]. Dès lors comme le fichier par défaut n'est plus visible par le JRE, il utilisera le fichier .properties spécifié dans les VM Options.

En exécutant le code source CS04 après avoir spécifié la VM Option Dlog4j.configurationFile, on obtient les lignes de log suivantes dans la console.

```
Début d'exécution de la méthode main
Début de création de l'objet BufferedReader
Fin de création de l'objet BufferedReader
Début de récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
Fin de récupération de l'entrée utilisateur
```

[26] Pour voir les différentes options de formatage proposées par log4j2, consulter cette page : https://logging.apache.org/log4j/2.x/manual/layouts.html

[27] Le fichier logging.properties se situe généralement dans le répertoire ${JAVA_HOME}/jre/lib ou dans le dossier conf de l'installation de votre JDK. Pour notre cas, il est situé à l'emplacement C:\Program Files\Java\jdk-20\conf\logging.properties

```
Début envoi salutation
Hello Kevin
Fin envoi salutation
Fin d'exécution de la méthode main
```

11.7.4.3 Formatage des lignes de logs : utilisation des variables de formatage

Comme on peut le constater, les lignes de logs initialement affichés en utilisant la configuration minimale dans le fichier log4j2.properties ne montrent que le corps de message sans aucun détail supplémentaire. Elles n'affichent aucune option supplémentaire comme la date, le niveau, etc.. C'est le comportement par défaut de l'option de formatage SimpleLayout. Mais un tel mode de logging n'est pas toujours exploitable en condition de production. Des détails supplémentaires sont nécessaires pour pouvoir exploiter les logs : date de génération, niveau de log, classe ayant généré la ligne de log, etc…

Cependant le framework Log4j2 offre plusieurs autres options pour formater les lignes de logs. Dans ce document, nous présentons le cas spécifique du SimpleLayout[28]. L'exemple ci-dessous montre l'utilisation du PatternLayout à la place du PatternLayout pour définir le fichier Log4j2.properties.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.properties

```
status = warn
# Définir le console appender pour orienter les logs vers la console
appender.console.type = Console
appender.console.name = LogToConsole
# Définir de l'option de formatage
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n
# Appel du console appender
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
```

Le formatage des logs est défini avec la propertie *appender.console.layout.type*. Ici nous choisissons la valeur PatternLayout qui permet de formater le message suivant un pattern (motif) bien défini. Ce pattern est spécifié avec la propertie *appender.console.layout.pattern*. Ici nous avons choisi la valeur `[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n`.

Cette valeur est constituée d'un certain nombre de variables de formatage. On note par exemple la présence de %level qui permet d'indiquer le niveau de log, %d qui permet d'indiquer la date, %t qui permet d'indiquer le nom du thread ayant généré la ligne de log, %c qui indique la classe à partir de laquelle le Logger a été défini, %msg qui indique le corps de message du log, etc. Voici ci-dessous quelques variables de formatage Log4j2 :

[28] Pour voir les différentes options de formatage proposées par log4j2, consulter cette page : https://logging.apache.org/log4j/2.x/manual/layouts.html

- **%d** : permet d'indiquer la date d'émission de la ligne de log. Ex : %d{dd MMM yyyy HH:MM:ss }
- **%level** : le niveau du log. Les valeurs possibles renvoyées sont : TRACE, DEBUG, INFO, WARN, ERROR et FATAL
- **%m** (ou %msg) : Renvoie le corps de message du log.
- **%logger** : permet d'indiquer le nom de la classe qui a émis le message
- **%C** : équivalenet à %logger, permet d'indiquer le nom de la classe qui a émis le message.
- **%c** : équivalent à %logger et %C, permet d'indiquer le nom de la catégorie ou du logger qui a émis le message
- **%r** : Renvoie le nombre de millisecondes écoulées entre le lancement de l'application et l'émission du message
- **%t** : indique le nom du thread qui a émis la ligne de log.
- **%n** : renvoie un saut de ligne dépendant de la plate-forme
- …
- …

Alignement et troncature des variables de formatage

Il est également possible d'agir sur l'alignement de la valeur renvoyée par une variable de formatage. Il est même possible de tronquer la valeur renvoyée si cette valeur s'avère trop longue. Les opérateurs ci-dessous montre quelques mises en forme de la valeur renvoyée par une variable de formatage. Les variables de formatage sont représentées par v (v représente ici les variables de formatage de base). Le nombre minimal de caractères qui forment la valeur n. Par exemples :

- **%nv** : permet un alignement à droite de la valeur renvoyée par une variable. Ex : %5level fait un alignement à droite du niveau du message 5. A noter que des espaces blancs sont ajoutés si le nombre de caractères de la valeur v est inférieure à n
- **%-nv** : permet un alignement à gauche. Ex : %30m : fait un alignement à gauche du corp du message. A noter également que des espaces blancs sont ajoutés si le nombre de caractères de m est inférieur à n
- **%.nv** : permet de tronquer le nombre de caractères si le nombre de caractères de la valeur v est supérieur à n caractères. Ex : %.10m limite le nombre de caractères du message à 10.
- **%-n.nv** : C'est une combinaison d'opération. Cette spécification permet de faire un alignement à gauche, fait une troncature de la valeur à n caractères ou le complète par des espaces blancs si le nombre de caractères est inférieur est à n.

Reéxecutons le code source CS04 en en utilisant un fichier log4J2.properties définit comme suit :

```
status = warn
# Définir le console appender pour orienter les logs vers la console
```

```
appender.console.type = Console
appender.console.name = LogToConsole
# Définir de l'option de formatage
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n
# Appel des appenders
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
```

Cette configuration produit dans la console les lignes de log suivantes :

```
[INFO ] 2023-05-13 09:39:39.956 [main] Main - Début d'exécution de la méthode
main
[DEBUG] 2023-05-13 09:39:39.959 [main] Main -  Début de création de l'objet
BufferedReader
[DEBUG] 2023-05-13 09:39:39.959 [main] Main -  Fin de création de l'objet
BufferedReader
[DEBUG] 2023-05-13 09:39:39.960 [main] Main -  Début de récupération de
l'entrée utilisateur
Saisissez votre nom, svp : Kevin
[DEBUG] 2023-05-13 09:39:45.493 [main] Main -  Fin de récupération de l'entrée
utilisateur
[DEBUG] 2023-05-13 09:39:45.494 [main] Main -  Début envoi salutation
Hello Kevin
[DEBUG] 2023-05-13 09:39:45.494 [main] Main -  Fin envoi salutation
[INFO ] 2023-05-13 09:39:45.494 [main] Main - Fin d'exécution de la méthode
main
```

11.7.4.4 Envoi des logs dans un seul fichier : FileAppender

Le framework Log4j2 offre des classes pour envoyer les logs vers plusieurs sorties : console, fichier, base de données, email, etc. Les exemples présentés jusque-là envoient les logs dans la console. A présent, nous allons envoyer les logs vers un fichier. Log4j2 permet d'envoyer les logs vers plusieurs types de fichiers : fichiers plats, fichiers xml, fichiers html. La page suivante montre les différents Appenders du framework Log4j2: https://logging.apache.org/log4j/2.x/manual/appenders.html

La configuration ci-dessous permet d'envoyer les logs aussi bien dans un fichier plat nommé logs/file.log mais aussi dans la console.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.properties

```
# Nom de la configuration
name=PropertiesConfig
# Choix des Appenders
# choix des noms des appenders. choix libre et non obligatoires. Ex : console,
myConsole, file, myFile..)
appenders = console, file

# Détails de l'Appender console
appender.console.type = Console
appender.console.name = LogToConsole
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
```

```
%c{1} - %msg%n
# Détails de l'Appender file
appender.file.type = File
appender.file.name = LogToFile
# Fichier stockant les logs
appender.file.fileName=logs/file.log
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1}
- %msg%n
# Appel des appenders
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
rootLogger.appenderRef.file.ref = LogToFile
```

En utilisant le fichier log4j2.properties, la classe FileAppender est traduite par la propertie appender.file.type = File. Dans cet exemple, le nom du fichier de log est défini avec appender.file.fileName=logs/file.log. Ce fichier se situe à la racine de notre projet Java C:\MY_JAVA_PROJECTS\javaTuto. Mais il peut se situer à n'importe quel emplacement accessible depuis l'environnent d'exécution du code. L'appel du FileAppender est fait avec la ligne rootLogger.appenderRef.file.ref.

L'exécution du code source CS04 avec cette configuration produit les lignes de logs suivantes :

```
[INFO ] 2023-05-13 16:12:28.330 [main] Main - Début d'exécution de la méthode
main
[DEBUG] 2023-05-13 16:12:28.333 [main] Main -  Début de création de l'objet
BufferedReader
[DEBUG] 2023-05-13 16:12:28.334 [main] Main -  Fin de création de l'objet
BufferedReader
[DEBUG] 2023-05-13 16:12:28.334 [main] Main -  Début de récupération de
l'entrée utilisateur
[DEBUG] 2023-05-13 16:12:38.685 [main] Main -  Fin de récupération de l'entrée
utilisateur
[DEBUG] 2023-05-13 16:12:38.685 [main] Main -  Début envoi salutation
[DEBUG] 2023-05-13 16:12:38.686 [main] Main -  Fin envoi salutation
[INFO ] 2023-05-13 16:12:38.686 [main] Main - Fin d'exécution de la méthode
main
```

Remarque importante: Il faut noter que le FileAppender stocke toutes les lignes de logs dans un seul fichier. Ainsi, toutes les fois que le programme Java est exécuté, les logs sont dirigés vers le même fichier. Ce qui aboutit à faire grandir indéfiniment le fichier. Du fait de ce comportement, le FileAppender doit être utilisé à bon escient. Cet Appender reste tout de même utilisable dans les situations où les fichiers de log sont purgés à intervalle réguliers après le passage d'un traitement qui les exploite.

11.7.4.5 Envoi des logs dans un fichier avec rotation : RollingFile

Comme nous l'avons déjà évoqué plus haut, le FileAppender envoie toutes les lignes de logs vers un seul fichier, quelque que soit le nombre de fois où le programme Java est exécuté. Ceci est une limite du FileAppender car ce comportement ne convient pas dans toutes les

situations. Le RollingFile permet de répondre aux lacunes du FileAppender en faisant une rotation des fichiers, c'est-à-dire en créant un nouveau fichier de logs chaque fois qu'un critère de rotation est vérifié. Par exemple, un nouveau fichier de log est créé lorsque le fichier de log courant atteint une certaine taille. Dans cette section, nous allons montrer comment utiliser le RollingFile à partir de la configuration du fichier log4j2.properties.

L'exemple ci-dessous montre comment spécifier RollingFile Appender dans un fichier log4j2.properties.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.properties

```
# Détails de l'Appender console
appender.console.type = Console
appender.console.name = LogToConsole
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n

# Détails de l'Appender Rolling
appender.rolling.type = RollingFile
appender.rolling.name = LogToRollingFile
appender.rolling.fileName= logs/file.log
appender.rolling.filePattern= logs/file_%d{yyyyMMdd}.log.gz
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n

# RollingFileAppender rotation policy
appender.rolling.policies.type = Policies
appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
appender.rolling.policies.size.size = 50MB
appender.rolling.policies.time.type = TimeBasedTriggeringPolicy
appender.rolling.policies.time.interval = 1
appender.rolling.policies.time.modulate = true
appender.rolling.strategy.type = DefaultRolloverStrategy

# Appel des appenders
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
rootLogger.appenderRef.rolling.ref = LogToRollingFile
```

La définition de ce fichier de properties pour spécifier le RollingAppender appelle un certain nombre de commentaires. D'abord, comme on peut le remarquer, le RollingAppender est déclaré et défini avec la propertie appender.rolling.type = RollingFile. Nous avons choisi d'attribuer le nom LogToRollingFile à l'appender. Le fichier de log courant est nommé logs/file.log tandis que les fichiers générés par rotation sont nommés suivant le pattern logs/file_%d{yyyyMMdd}.log.gz. Ce pattern signifie qu'à chaque rotation, un nouveau fichier suffixé avec la date courante et l'extension log.gz est généré. Le format utilisé pour la date est très important lors de la génération des fichiers. Par exemple, avec la valeur %d{yyyyMMdd} un nouveau fichier est généré chaque jour même si la taille maximale spécifiée pour le fichier de log courant n'est pas encore atteinte. Avec la valeur %d{yyyyMMddHH}, un fichier sera généré chaque heure. Pour les grosses applications comme les applications gérant le big data (Google, Meta, Tweeter, etc...) qui génèrent

d'énormes volumes de logs, on peut même spécifier des valeurs comme %d{yyyyMMddHHmm}, etc.

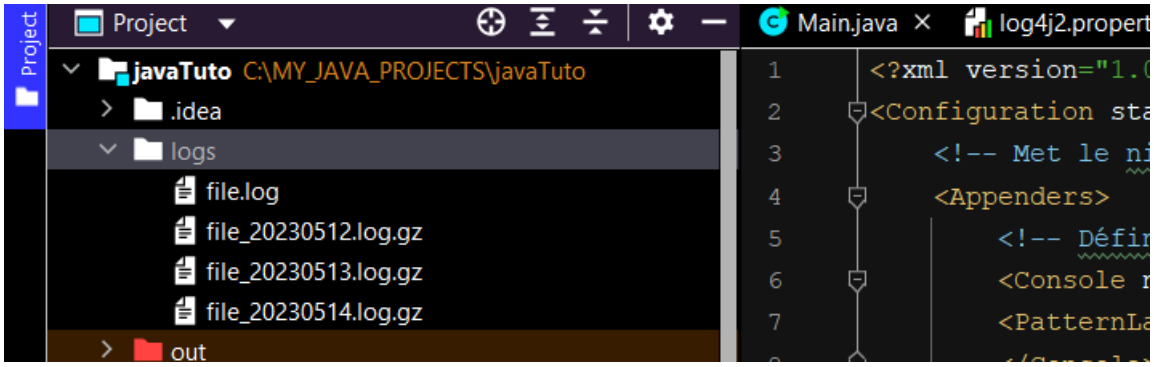
S'agissant de la taille maximale du fichier de log courant logs/file.log, elle est définie avec la propertie appender.rolling.policies.size.size. Dans l'exemple nous avons fixé la valeur à 50MB. Cela signifie qu'une nouvelle rotation doit être faite chaque fois que le fichier de log courant atteint 50MB. Ce critère fait partie d'un ensemble de policies associé à l'appender. Les policies sont déclarées avec la propertie appender.rolling.policies.type = Policies. Dans cet exemple, nous avons choisi deux policies. La policy SizeBasedTriggeringPolicy qui permet de déclencher la rotation des fichiers en se basant sur la taille maximale fixée pour le fichier courant de log. Ici, la taille est fixée à 50MB avec la propertie size. Ensuite, nous choisissons la policy TimeBasedTriggeringPolicy qui permet de déclencher la rotation des fichiers, non pas en se basant sur la taille du fichier courant de log mais plutôt en suivant un intervalle de temps défini en nombre de jours avec la propertie appender.rolling.policies.time.interval. Ici, nous choisissons la valeur 1 qui signifie qu'une nouvelle rotation de fichier sera déclenchée chaque jour, et cela indépendamment de la policy déjà définie avec SizeBasedTriggeringPolicy. Toutefois, il faut noter que c'est la spécification des deux policies ensemble en plus de la propertie appender.rolling.filePattern qui permet de gérer avec souplesse la rotation des fichiers.

En résumé, la propertie appender.rolling.filePattern avec la variable %d{ } permet de déclencher une rotation en générant un nouveau fichier à chaque nouvelle valeur de %d{ }. La policy SizeBasedTriggeringPolicy permet de déclencher la rotation en se basant sur une taille maximale spécifiée pour le fichier courant. Enfin la policy TimeBasedTriggeringPolicy permet de déclencher la rotation de fichier suivant un intervalle de temps spécifié en nombre de jours.

En exécutant le code source CS04, le fichier courant de log contient les lignes qui se présentent comme suit :

```
[INFO ] 2023-05-15 16:12:28.330 [main] Main - Début d'exécution de la méthode
main
[DEBUG] 2023-05-15 16:12:28.333 [main] Main -  Début de création de l'objet
BufferedReader
[DEBUG] 2023-05-15 16:12:28.334 [main] Main -  Fin de création de l'objet
BufferedReader
[DEBUG] 2023-05-15 16:12:28.334 [main] Main -  Début de récupération de
l'entrée utilisateur
[DEBUG] 2023-05-15 16:12:38.685 [main] Main -  Fin de récupération de l'entrée
utilisateur
[DEBUG] 2023-05-15 16:12:38.685 [main] Main -  Début envoi salutation
[DEBUG] 2023-05-15 16:12:38.686 [main] Main -  Fin envoi salutation
[INFO ] 2023-05-15 16:12:38.686 [main] Main - Fin d'exécution de la méthode
main
```

Et en exécutant plusieurs jours de suite le code source CS04 avec la configuration RollingFile définie, les fichiers de log seront générés en rotation (voir la capture d'écran pour un lancement quatre jours de suite).



Le fichier file.log contient les lignes de logs générées pour la date courante %d{yyyyMMdd} tandis que les fichiers file_20230514.log.gz, file_20230513.log.gz et file_20230513.log.gz contiennent les lignes de log des jours précédents. Ces fichiers sont générés chaque jour par rotation du fichier file.log de la veille avant l'exécution du programme à la date courante.

11.7.5 Configuration du logging avec le fichier log4j2.xml

Comme évoqué dans les sections précédentes, on peut utiliser un fichier de configuration log4j2.xml à la place du fichier log4j2.properties. Cette section a pour but de présenter l'usage du fichier log4j2.xml à la place du fichier log4j2.properties.

11.7.5.1 Création du fichier log4j2.xml

Le fichier log4j2.xml peut être créé dans le dossier resources à l'intérieur du dossier src. Voici ci-dessous un exemple basique de fichier log4j2.xml.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        </Console>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
        </Root>
    </Loggers>
</Configuration>
```

Cette définition du fichier de log4j2.xml est la configuration la plus minimale pour orienter les lignes de logs dans la console sans aucune option supplémentaire. Etant donné que nous n'avons défini aucune une option de formatage, la classe Logger utilisera l'option de

formatage par défaut qui est le SimpleLayout. Le SimpleLayout envoie seulement les corps des messages sans aucun détail supplémentaire comme la date, le niveau, etc…

11.7.5.2 Appel du fichier log4j2.xml

Tout comme pour le fichier log4j2.properties, pour pouvoir utiliser les configurations définies dans le fichier log4j2.xml celui-ci doit être appelé lors de l'exécution du programme Java en définissant une VM Option supplémentaire spécifiée comme suit :

```
-
Dlog4j.configurationFile=C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\log4j2
.xml
```

Pour définir cette VM Option dans Intellij, cliquer dans le menu Run > Edit configuration. Cliquer >Modify options > Add VM options. Et copier-coller la valeur indiquée ci-dessus.

Attention : Parfois même si ce paramètre est bien défini dans les VM options, il arrive que le JRE fasse toujours appel au fichier logging.properties défini par défaut pour votre installation Java en l'occurrence le fichier utilisé par le framework JUL. Dans une telle situation, il est préférable de renommer le fichier logging.properties en par exemple logging_JUL_.properties[29]. Dès lors comme le fichier par défaut n'est plus visible par le JRE, il chargera le fichier properties spécifié dans les VM Options

En exécutant le code source CS04 après avoir spécifié la VM Option, on obtient les lignes de log suivantes dans la console.

```
Début d'exécution de la méthode main
Début de création de l'objet BufferedReader
Fin de création de l'objet BufferedReader
Début de récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
Fin de récupération de l'entrée utilisateur
Début envoi salutation
Hello Kevin
Fin envoi salutation
Fin d'exécution de la méthode main
```

11.7.5.3 Formatage des lignes de logs : utilisation des variables de formatage

Les lignes de logs initialement affichées en utilisant la configuration minimale dans le fichier log4j2.xml ne montrent que le corps de message sans aucun détail supplémentaire. Celui-ci est le comportement par défaut de l'option de formatage SimpleLayout. Mais ce mode de logging n'est pas toujours exploitable. Des détails supplémentaires sont nécessaires pour pouvoir exploiter les logs : date de génération, niveau de log, classe ayant

[29] Le fichier logging.properties se situe généralement dans le répertoire ${JAVA_HOME}/jre/lib ou dans le dossier conf de l'installation de votre JDK. Pour notre cas, il est situé à l'emplacement C:\Program Files\Java\jdk-20\conf\logging.properties

généré la ligne de log, etc… Cependant le framework Log4j2 offre plusieurs options pour formater les lignes de logs. Dans ce document, nous présentons le cas spécifique du PatternLayout[30]. L'exemple ci-dessous montre l'utilisation du PatternLayout à la place du SimpleLayout pour définir le fichier log4j2.xml.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C -
%msg%n" />
        </Console>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
        </Root>
    </Loggers>
</Configuration>
```

Le formatage des logs est défini avec la balise <PatternLayout /> qui permet de formater le message suivant un pattern (un motif). Ce pattern est spécifié avec la propertie *pattern*. Ici nous avons choisi la valeur `%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C - %msg%n`

Cette valeur est constituée d'un certain nombre de variables de formatage. On note par exemple la présence de %d qui permet d'indiquer la date, %t qui permet d'indiquer le nom du thread ayant généré la ligne de log, %level qui permet d'indiquer le niveau de log, %logger qui indique la classe à partir de laquelle le Logger a été défini, %msg le corps de message du log, etc. Voici ci-dessous quelques variables de formatage Log4j2 :

- **%d** : permet d'indiquer la date d'émission de la ligne de log. Ex : %d{dd MMM yyyy HH:MM:ss }
- **%level** : le niveau du log. Les valeurs possibles renvoyées sont : TRACE, DEBUG, INFO, WARN, ERROR et FATAL
- **%m** (ou %msg) : renvoie le corps de message du log.
- **%logger** : permet d'indiquer le nom de la classe qui a émis le message
- **%C** : équivalenet à %logger, permet d'indiquer le nom de la classe qui a émis le message.
- **%c** : équivalent à %logger et %C, permet d'indiquer le nom de la catégorie ou du logger qui a émis le message

[30] Pour voir les différentes options de formatage proposées par log4j2, consulter cette page : https://logging.apache.org/log4j/2.x/manual/layouts.html

- **%r** : renvoie le nombre de millisecondes écoulées entre le lancement de l'application et l'émission du message
- **%t** : indique le nom du thread qui a émis la ligne de log.
- **%n** : renvoie un saut de ligne dépendant de la plate-forme
- …
- …

Alignement et troncature des variables de formatage

Il est également possible d'agir sur l'alignement de la valeur renvoyée par une variable de formatage. Il est même possible de tronquer la valeur renvoyée si cette valeur s'avère trop longue. Les opérateurs ci-dessous montrent quelques mises en forme de la valeur renvoyée par une variable de formatage. Les variables de formatage sont représentées par v (v représente ici les variables de formatage de base). Le nombre minimal de caractères qui forment la valeur n. Par exemples :

- **%nv** : permet un alignement à droite de la valeur renvoyée par une variable. Ex : %5level fait un alignement à droite du niveau du message 5. A noter que des espaces blancs sont ajoutés si le nombre de caractères de la valeur v est inférieure à n
- **%-nv** : permet un alignement à gauche. Ex : %30m : fait un alignement à gauche du corps du message. A noter également que des espaces blancs sont ajoutés si le nombre de caractères de m est inférieur à n
- **%.nv** : permet de tronquer le nombre de caractères si le nombre de caractères de la valeur v est supérieur à n caractères. Ex : %.10m limite le nombre de caractères du message à 10.
- **%-n.nv** : c'est une combinaison d'opérations. Cette spécification permet de faire un alignement à gauche, fait une troncature de la valeur à n caractères ou le complète par des espaces blancs si le nombre de caractères est inférieur est à n.

Réexécutions le code source CS04 en en utilisant un fichier log4j2.xml définit comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C -
%msg%n" />
        </Console>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
        </Root>
    </Loggers>
</Configuration>
```

Cette configuration produit dans la console les lignes de log suivantes :

```
2023-05-14 17:13:10.586 [main] INFO  com.tuto.logging.Main - Début d'exécution
de la méthode main
2023-05-14 17:13:10.589 [main] DEBUG com.tuto.logging.Main -  Début de création
de l'objet BufferedReader
2023-05-14 17:13:10.590 [main] DEBUG com.tuto.logging.Main -  Fin de création
de l'objet BufferedReader
2023-05-14 17:13:10.590 [main] DEBUG com.tuto.logging.Main -  Début de
récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
2023-05-14 17:13:14.386 [main] DEBUG com.tuto.logging.Main -  Fin de
récupération de l'entrée utilisateur
2023-05-14 17:13:14.386 [main] DEBUG com.tuto.logging.Main -  Début envoi
salutation
Hello Kevin
2023-05-14 17:13:14.387 [main] DEBUG com.tuto.logging.Main -  Fin envoi
salutation
2023-05-14 17:13:14.387 [main] INFO  com.tuto.logging.Main - Fin d'exécution de
la méthode main
```

11.7.5.4 Envoi des logs dans un seul fichier : FileAppender

Le framework Log4j2 offre des classes pour envoyer les logs vers plusieurs sorties : console, fichier, base de données, email, etc. Les exemples présentés jusque-là envoient les logs dans la console. A présent, nous allons envoyer les logs vers un fichier. Log4j2 permet d'envoyer les logs vers plusieurs types de fichiers : fichiers plats, fichiers xml, fichiers html. La page suivante montre les différents Appenders du framework Log4j2: https://logging.apache.org/log4j/2.x/manual/appenders.html

La configuration ci-dessous permet d'envoyer les logs aussi bien dans un fichier plat nommé logs/file.log mais aussi dans la console.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C -
%msg%n" />
        </Console>
        <!-- Définit un Appender nommé LogToFile -->
        <File name="LogToFile" fileName="logs/file.log">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%C - %msg%n" />
        </File>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
            <AppenderRef ref="LogToFile" />
        </Root>
    </Loggers>
</Configuration>
```

En utilisant le fichier log4j2.xml, la classe FileAppender est représentée par la balise <File>..<File/>. A l'intérieur de cette balise nous définissons les properties name et fileName qui permettent d'indiquer respectivement le nom de l'Appender et le chemin du fichier vers lequel les lignes de log seront orientées. Au premier lancement du programme, si ce chemin n'existe pas, il sera automatiquement créé. Dans cet exemple, le fichier se situe à la racine du projet Java C:\MY_JAVA_PROJECTS\javaTuto. Mais il peut se situer à n'importe quel emplacement accessible depuis l'environnent d'exécution du code. L'appel du FileAppender est fait avec la balise <AppenderRef ref="LogToFile" /> défini au sein de la balise <Root>… <Root/> qui elle-même est définie au sein de la balise <Loggers>…<Loggers />

L'exécution du code source CS04 avec cette configuration produit les lignes de logs suivantes :

```
2023-05-14 17:33:49.708 [main] INFO  com.tuto.logging.Main - Début d'exécution
de la méthode main
2023-05-14 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Début de création
de l'objet BufferedReader
2023-05-14 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Fin de création
de l'objet BufferedReader
2023-05-14 17:33:49.713 [main] DEBUG com.tuto.logging.Main -  Début de
récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
2023-05-14 17:33:54.067 [main] DEBUG com.tuto.logging.Main -  Fin de
récupération de l'entrée utilisateur
2023-05-14 17:33:54.068 [main] DEBUG com.tuto.logging.Main -  Début envoi
salutation
Hello Kevin
2023-05-14 17:33:54.069 [main] DEBUG com.tuto.logging.Main -  Fin envoi
salutation
2023-05-14 17:33:54.070 [main] INFO  com.tuto.logging.Main - Fin d'exécution de
la méthode main
```

Remarque importante: Il faut noter que le FileAppender stocke toutes les lignes de logs dans un seul fichier. Ainsi, toutes les fois que le programme Java est exécuté, les logs sont dirigés vers le même fichier faisant ainsi que le fichier grandit indéfiniment. Du fait de ce comportement, le FileAppender doit être utilisé à bon escient. Cet Appender reste tout de même utilisable dans les situations où les fichiers de log sont purgés à intervalle régulier après le passage d'un traitement qui les exploite.

11.7.5.5 Envoi des logs dans un fichier avec rotation : RollingFile

Comme nous l'avons déjà évoqué plus haut, le FileAppender envoie toutes les lignes de logs vers un seul fichier, quelque que soit le nombre de fois où le programme Java est exécuté. Ceci est une limite du FileAppender car ce comportement ne convient pas dans toutes les situations. Le RollingFile permet de répondre aux lacunes du FileAppender en faisant une rotation des fichiers, c'est-à-dire en créant un nouveau fichier de logs chaque fois qu'un critère de rotation est vérifié. Par exemple, un nouveau fichier de log est créé lorsque le

fichier actuel atteint une certaine taille. Dans cette section, nous allons montrer comment utiliser le RollingFile à partir de la configuration du fichier log4j2.xml.

L'exemple ci-dessous montre comment spécifier RollingFile Appender dans un fichier log4j2.xml.

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C -
%msg%n" />
        </Console>
        <!-- Définit un Appender nommé LogToRollingFile -->
        <RollingFile name="LogToRollingFile" fileName="logs/file.log"
filePattern= "logs/file_%d{yyyyMMdd}.log.gz">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%C - %msg%n" />
            <!-- Définition des critères de rotation -->
            <Policies>
                <!-- Taille maximale du fichier courant fixé à 50MB -->
                <SizeBasedTriggeringPolicy size="50MB" />
                <!-- Rotation chaque 1 jour -->
                <TimeBasedTriggeringPolicy interval="1" modulate="true" />
            </Policies>
            <DefaultRolloverStrategy >
            </DefaultRolloverStrategy>
        </RollingFile>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
            <AppenderRef ref="LogToRollingFile" />
        </Root>
    </Loggers>
</Configuration>
```

La définition de ce fichier de configuration pour spécifier le RollingAppender appelle un certain nombre de commentaires. D'abord, comme on peut le remarquer, le RollingAppender est déclaré dans la balise <RollingFile>...<RollingFile/>. Nous avons choisi d'attribuer le nom LogToRollingFile à l'appender. Le fichier de log courant est nommé logs/file.log tandis que les fichiers générés par rotation sont nommés suivant le pattern logs/file_%d{yyyyMMdd}.log.gz. Ce pattern signifie qu'à chaque rotation, un nouveau fichier suffixé avec la date courante et l'extension log.gz est généré. Le format utilisé pour la date est très important lors de la génération des fichiers. Par exemple, avec la valeur %d{yyyyMMdd} un nouveau fichier est généré chaque jour même si la taille maximale spécifiée pour le fichier de log courant n'est pas encore atteinte. Avec la valeur %d{yyyyMMddHH}, un fichier sera généré chaque heure. Pour les grosses applications comme les applications gérant le big data (Google, Meta, Tweeter, etc...) qui génèrent

d'énormes volumes de logs, on peut même spécifier des valeurs comme %d{yyyyMMddHHmm}, etc.

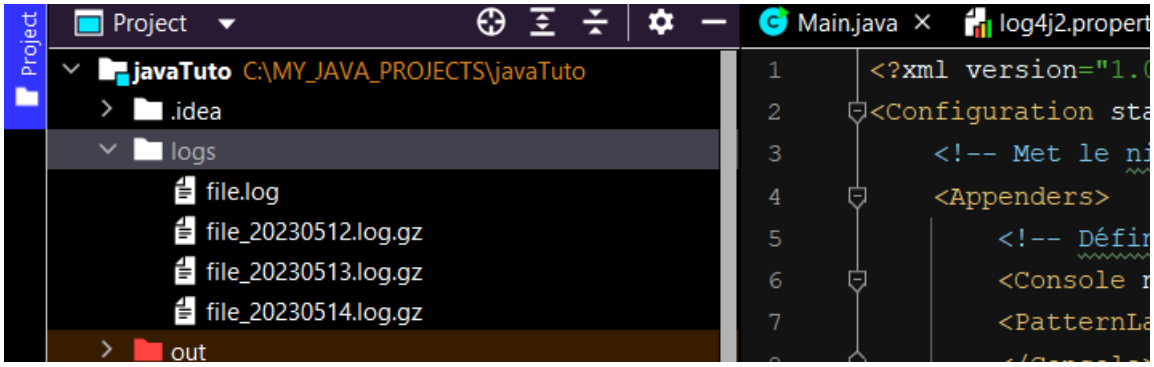
La balise des policies permet de spécifier les différents critères servant de déclencheur à la rotation des fichiers. Ici deux policies sont spécifiées : <SizeBasedTriggeringPolicy/> et <TimeBasedTriggeringPolicy/>. La policy SizeBasedTriggeringPolicy permet de déclencher la rotation des fichiers en se basant sur la taille maximale fixée pour le fichier courant de log. Ici, la taille est fixée à 50MB avec la propertie size. La policy TimeBasedTriggeringPolicy permet de déclencher la rotation des fichiers, non pas sur la base de la taille du fichier courant de log mais plutôt en se basant sur un intervalle de temps défini en nombre de jours défini avec la propertie time.interval. Ici, nous choisissons la valeur 1 qui signifie qu'une nouvelle rotation des fichiers sera déclenchée chaque jour. Et cela indépendamment de la policy déjà définie avec SizeBasedTriggeringPolicy. Toutefois, il faut noter que c'est la spécification des deux policies ensemble en plus de la propertie filePattern qui permet de gérer avec souplesse la rotation des fichiers.

En résumé, la propertie filePattern avec la variable %d{ } permet de déclencher une rotation en générant un nouveau fichier à chaque nouvelle valeur de %d{ }. La policy SizeBasedTriggeringPolicy permet de déclencher la rotation en se basant sur une taille maximale spécifiée pour le fichier courant. Enfin la policy TimeBasedTriggeringPolicy permet de déclencher la rotation des fichiers suivant un intervalle de temps spécifié en nombre de jours.

En exécutant le code source CS04, le fichier courant de log contient les lignes qui se présentent comme suit :

```
2023-05-15 17:33:49.708 [main] INFO  com.tuto.logging.Main - Début d'exécution
de la méthode main
2023-05-15 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Début de création
de l'objet BufferedReader
2023-05-15 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Fin de création
de l'objet BufferedReader
2023-05-15 17:33:49.713 [main] DEBUG com.tuto.logging.Main -  Début de
récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
2023-05-15 17:33:54.067 [main] DEBUG com.tuto.logging.Main -  Fin de
récupération de l'entrée utilisateur
2023-05-15 17:33:54.068 [main] DEBUG com.tuto.logging.Main -  Début envoi
salutation
Hello Kevin
2023-05-15 17:33:54.069 [main] DEBUG com.tuto.logging.Main -  Fin envoi
salutation
2023-05-15 17:33:54.070 [main] INFO  com.tuto.logging.Main - Fin d'exécution de
la méthode main
```

Et en exécutant plusieurs jours de suite le code source CS04, les fichiers de log seront générés en rotation (voir une capture d'écran avec un lancement quatre jours de suite).



Le fichier file.log contient les lignes de logs générés pour la date courante %d{yyyyMMdd} tandis que les fichiers file_20230514.log.gz, file_20230513.log.gz et file_20230513.log.gz contiennent les lignes de log des jours précédents. Ces fichiers sont générés chaque jour par rotation du fichier file.log de la veille avant l'exécution du programme à la date courante.

11.8 Logging avec le framework SLF4J

Le framework SLF4J (Simple Logging Facade for Java) est une couche d'abstraction qui permet de choisir entre plusieurs frameworks de logging lors du déploiement de votre application : JUL (java.util.logging), Log4j2, logBack, etc... A noter que SLF4J n'est pas un framework de logging proprement dit. Son rôle est de faciliter la gestion des logs en découplant l'écriture du code du choix du système de logging. En effet, grâce à SLF4J, on peut écrire le code sans avoir à se soucier du système de logging qui sera utilisé par la suite. Le choix du framework de logging peut être fait plus tard lors du déploiement de l'application. Ce qui apporte beaucoup plus de souplesse dans le processus de développement applicatif. De même, pour une application déjà déployée, SLF4J permet de migrer d'un système de logging à un autre sans aucun impact sur le code source.

Comme son nom l'indique, SLF4J offre une façade unique et harmonisée permettant de communiquer avec plusieurs systèmes de logging. Pour voir l'utilité potentielle de SLF4J, supposons par exemple que vous ayez développé une application web et que vous comptez déployer cette application sur deux serveurs web différents : Tomcat et Nginx. Or chaque Serveur web utilise sa propre API de logging. Par exemple Tomcat utilise, par défaut, le framework JUL. Supposons que le serveur Nginx ait été configuré pour logguer avec du Log4j2. Comment garantir que votre application puisse être déployé sur chacun des deux serveurs et continuer à logguer correctement les évènements. En effet, si la classe Logger dans votre code est celle fournie par le framework JUL, votre application ne logguera pas correctement sur le serveur Nginx. De même si la classe Logger que vous utilisez dans votre

code provient de Log4j2, votre application ne logguera pas correctement sur le serveur Tomcat. Pour assurer l'interopérabilité de votre logging entre les deux serveurs, la solution est d'utiliser la classe Logger fournit par le framework SLF4J. Ainsi, lors du déploiement de votre application, il est possible de choisir le framework de logging adapté à votre environnement.

Dans cette section, nous allons montrer comment utiliser le framework SLF4J pour logguer vers d'autres frameworks de logging, en particulier JUL et Log4j2.

11.8.1 Chargement de la librairie externe SLF4J

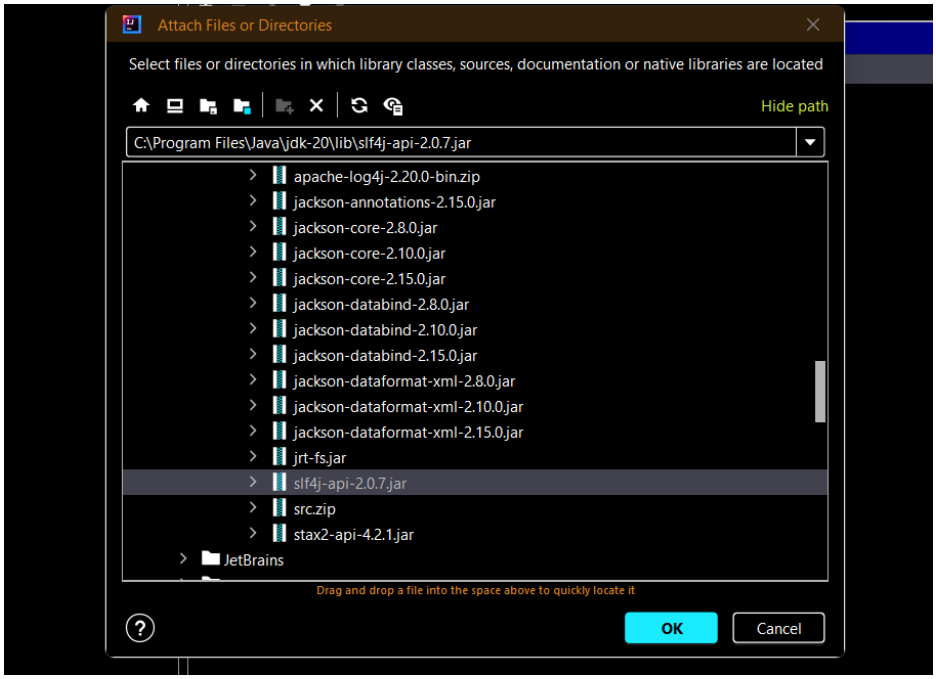
La librairie SLF4J se présente sous forme d'archive téléchargeable et installable dans le classpath du programme. Java offre plusieurs façons d'installer les dépendances et les librairies externes. On peut procéder soit par une installation manuelle, soit par une installation automatisée à travers l'utilisation des outils de gestion de dépendances comme Maven, Ivy et Gradle. Nous reviendrons plus tard sur l'utilisation de Maven pour la gestion des dépendences dans un projet Java[31]. Dans la présente section, nous montrons la méthode d'installation manuelle de la dépendence SLF4J.

Pour télécharger et installer manuellement SLF4J, suivre les étapes suivantes :

- Télécharger le fichier jar correspondant à l'api SLF4J depuis un repository de gestion de dépendence. Ex : https://repo1.maven.org/maven2/org/slf4j/slf4j-api/2.0.7/slf4j-api-2.0.7.jar
- Déposer ce fichier jar dans le dossier lib de votre installation Java ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\lib.
- Nous allons ajouter ce fichier jar dans le classpath de notre application. L'ajout peut être directement fait à partir de votre IDE. Par exemple, pour Intellij IDEA, l'ajout des fichiers jar se fait comme suit :
  1. Dans la barre des menus, cliquer sur File >Project Structure >Project Settings> Modules > Dependencies.
  2. Cliquer sur le signe + (Add). Et choisir JARs or Directories. Et aller chercher le fichier slf4j-api-2.0.7.jar disponible dans le dossier lib. Voir capture d'écran ci-dessous.

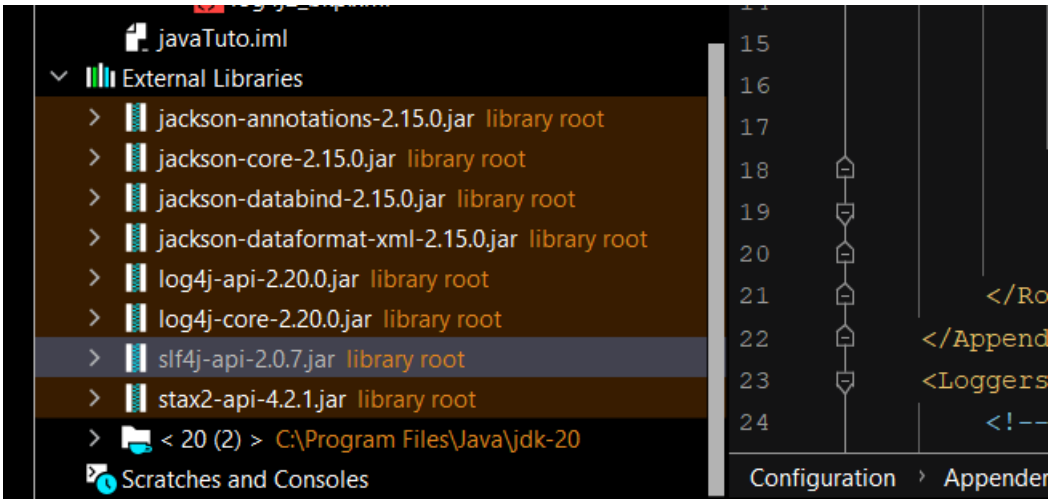
[31] Pour ajouter la dépendance slf4j dans un projet Maven, ajouter ces spécifications dans le pom.xml

```
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-api</artifactId>
   <version>2.0.7</version>
</dependency>
```



3. Cliquer sur ok pour sélectionner et cocher la case pour pouvoir ajouter le fichier aux dépendances. Devant la librairie sélectionnée, dans scope, dans la liste déroulante choisir Compile. Cliquer sur Ok pour valider.
4. Cliquer sur ok pour valider.

A la suite de cette procédure d'installation, le fichier jar ainsi que son contenu sont visibles à gauche dans External librairies. Ce qui montre que les dépendances SLF4J sont bien installées. Voir la capture d'écran ci-dessous.



11.8.2 Code source d'illustration : Code source CS05

Pour illustrer le logging avec le framework SLF4J, considérons le code source CS05 présenté ci-dessous. Ce code source correspond au code source CS03 et CS04 déjà utilisé dans les sections de présentation du framework JUL et Log4j2. Nous apportons, cependant quelques modifications. D'abord nous modifions la classe Logger en utilisant le package org.slf4j à la place à la place du package java.util.logging (pour JUL) ou du package org.apache.logging.log4j (pour Log4j2). Le code source ci-dessous montre les modifications apportées par rapport aux versions initiales utilisées avec le framework JUL ou avec le framework Log4j2.

Code source : CS05

```
package com.tuto.logging;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;


public class Main {
    private static final Logger LOGGER =
LoggerFactory.getLogger(Main.class.getName());

    public static void main(String[] args)  {


        LOGGER.info("Début d'exécution de la méthode main");
        try {
            LOGGER.debug(" Début de création de l'objet BufferedReader ");
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            LOGGER.debug(" Fin de création de l'objet BufferedReader ");

            LOGGER.debug(" Début de récupération de l'entrée utilisateur ");
            System.out.print("Saisissez votre nom, svp : "); // Ex : Kevin
            String nom = br.readLine();
            LOGGER.debug(" Fin de récupération de l'entrée utilisateur ");

            LOGGER.debug(" Début envoi salutation ");
            System.out.println("Hello " + nom); br.close();
            LOGGER.debug(" Fin envoi salutation ");
        }

        catch (IOException e) {
            LOGGER.error("Un problème est survenue lors de la lecture de la
valeur entrée par l'utilisateur");
        }

        LOGGER.info("Fin d'exécution de la méthode main");
    }
}
```

Comme on peut le remarquer, pour pouvoir utiliser SLF4J, nous avons d'abord importé deux classes : LoggerFactory et Logger. La classe LoggerFactory permet d'instancier un objet de Logger. L'objet de type Logger est créé avec l'instruction :

```
private static final Logger LOGGER = LoggerFactory.getLogger(Main.class.getName());
```

Cette instanciation est faite comme un attribut de la classe dont les évènements sont loggués. C'est pourquoi l'objet LOGGER est défini à l'extérieur de toutes les méthodes de la classe.

Après l'instanciation de l'objet LOGGER, on peut maintenant appeler les différentes méthodes correspondant aux différents niveaux de log.

Dans le code ci-dessus, comme on peut le remarquer, à chaque étape de l'exécution, nous affichons une ligne de log correspondant à une instruction spécifique. Nous utilisons trois niveaux de logs correspondant chacun à une instruction spécifique dans le code. Les trois niveaux de log utilisés sont : DEBUG, INFO, ERROR. Le niveau DEBUG permet de logguer de manière fine les instructions définies dans le programme. Le niveau INFO est utilisé pour fournir les informations sur les évènements les plus marquants dans l'exécution du traitement et le niveau ERROR est utilisé pour informer sur la survenue d'information plus grave nécessitant parfois l'arrêt de l'exécution du programme.

Les différents niveaux de log définis pour le framework SLF4J sont TRACE, DEBUG, INFO, WARN et ERROR. Même si le framework SLF4J reste très proche de Log4j2, il n'existe donc pas de correspondance parfaite entre les niveaux de logging pour les deux framework. Par exemple le niveau FATAL qui est le niveau le plus grave dans Log4j2 n'existe pas dans SLF4J. La différence de niveau reste également très marquante avec le framework JUL. Pour rappel, les différents niveaux de log définis pour le framework JUL sont CONFIG, FINEST, FINER, FINE, INFO et SEVERE. En pratique, les niveaux de log de SLF4J seront mis en correspondance avec le niveau de log le plus proche du framework cible. Par exemple, le niveau SEVERE de JUL correspond au niveau ERROR de SLFJ4 et Log4j2.

Remarque importante :

Même si la dépendance SLF4J est déjà installé (voir sous-section précédente), cela ne suffit pas encore pour logguer les lignes de logs spécifiées dans le code source CS05. Par exemple, en exécutant ce code, on obtient sur la console les sorties suivantes :

```
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#noProviders for further details.
Saisissez votre nom, svp : Kevin
Hello Kevin
```

Ces lignes indiquent qu'aucun provider (fournisseur) n'est encore spécifié pour logguer. Le provider SLF4J correspond au système de logging choisi pour générer les lignes de logs. Il peut s'agir de JUL, de Log4j2, de LogBack ou de tout autre framework de logging compatibles avec SLF4J. Dans les sections suivantes, nous allons présenter les cas où le provider est JUL ou Log4j2.

11.8.3 Générer les logs SLF4J avec le provider JUL

Comme nous l'avons déjà indiqué en introduction de la section le framework SLF4J n'est pas un framework de logging en tant que tel. C'est une couche d'abstraction de logging qui permet de choisir le système de logging et de changer de système de logging sans impacter le code source. SLF4J ne génère donc pas de ligne de log à proprement parler. Il se base toujours sur un autre framework de logging pour pouvoir logguer. Ce système cible est alors appelé provider. Dans cette sous-section, nous illustrons le cas où le framework JUL (java.util.logging) est utilisé comme provider pour SLF4J.

11.8.3.1 Installation de la librairie de liaison entre SLFJ4 et JUL : slf4j-jdk14

Pour pouvoir utiliser JUL comme provider de SLF4J, nous devons d'abord installer la librairie slf4j-jdk14 dans le classpath. Les étapes d'installation de cette librairie sont décrites ci-dessous.

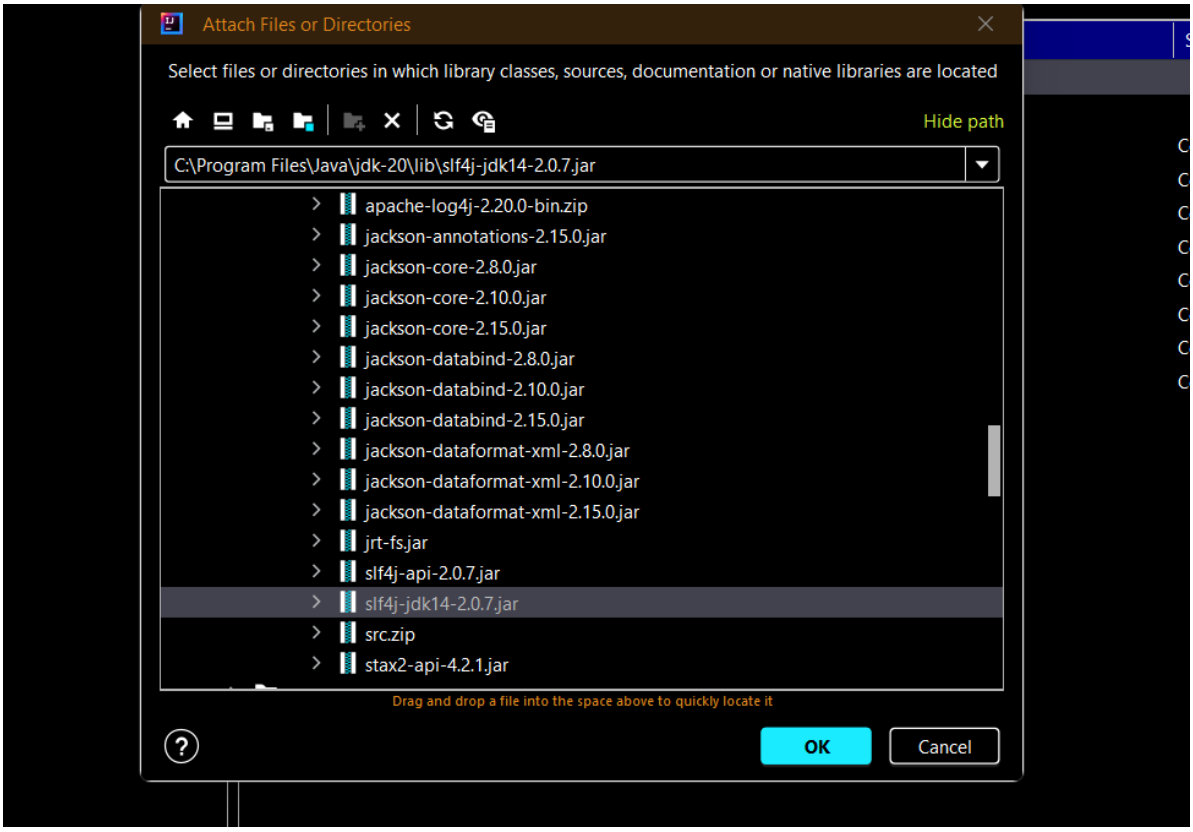
Comme la plupart des librairies externes Java, slf4j-jdk14 peut être installée dans le classpath, soit manuellement, soit automatiquement avec les outils de gestion de dépendances externes comme Maven, Gradle, Ivy, etc. Nous reviendrons plus tard sur l'utilisation de Maven pour la gestion des dépendances dans un projet Java[32]. Ici, nous montrons le cas de l'installation manuelle.

- S'assurer d'abord que d'autres librairies de liaisons de SLF4J avec d'autres frameworks n'existant déjà pas dans le classpath. Par exemple, s'assurer que le jar log4j-slf4j2-impl-2.xx.y.jar qui est la librairie de liaison de SLF4J avec le framework Log4j2 n'existe pas dans le classepath. Si d'autres librairies de liaisons existent, lors de l'exécution SLF4J vous recevrez des warnings du type : *SLF4J: Class path contains multiple SLF4J providers*. Et SLF4J choisit un provider parmi ceux disponibles. Pour cela, il vous renvoie un message du type : *SLF4J: Actual provider is of type [org.apache.logging.slf4j.SLF4JServiceProvider@25f38edc]*. Toutefois ce choix ne correspondra pas toujours à celui que vous souhaitez au départ. Pour éliminer ce type de désagrément, il est donc préférable, si cela n'a pas d'inconvénients majeurs, de supprimer du classpth toutes les libraires de liaison ne correspondant pas à votre souhait. Ces suppressions se font généralement par une simple exclusion du jar soit des librairies externes, soit du pom.xml s'il s'agit d'un projet Maven. Par exemple, on peut supprimer le jar log4j-slf4j2-impl-2.xx.y.jar s'il existe déjà dans le classpath.
- Télécharger le fichier jar correspondant à l'api slf4j-jdk14 depuis un repository de gestion de dépendence. Ex : https://repo1.maven.org/maven2/org/slf4j/slf4j-jdk14/2.0.7/slf4j-jdk14-2.0.7.jar

[32] Pour ajouter la dépendance slf4j-jdk14 dans un projet Maven, ajouter ces spécifications dans le pom.xml

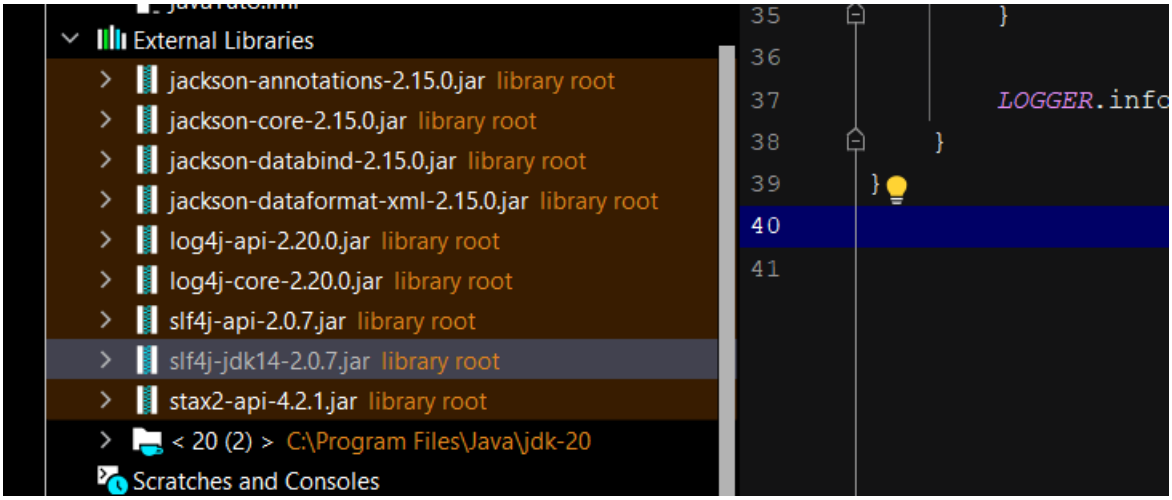
```
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-jdk14</artifactId>
   <version>2.0.7</version>
   <scope>test</scope>
</dependency>
```

- Déposer ce fichier jar dans le dossier lib de votre installation Java ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\lib.
- Nous allons ajouter ce fichier jar dans le classpath de notre application. L'ajout peut être directement fait à partir de l'IDE. Par exemple, pour Intellij IDEA, l'ajout du fichier jar se fait comme suit :
    1. Dans la barre des menus, cliquer sur File >Project Structure >Project Settings> Modules > Dependencies.
    2. Cliquer sur le signe + (Add). Et choisir JARs or Directories. Et aller chercher le fichier slf4j-jdk14-2.0.7.jar disponible dans le dossier lib. Voir capture d'écran ci-dessous.



    3. Cliquer sur ok pour sélectionner et cocher la case pour pouvoir ajouter le fichier aux dépendances. Devant la librairie sélectionnée, dans scope, dans la liste déroulante choisir Compile. Cliquer sur Ok pour valider.
    4. Cliquer sur ok pour valider.

A la suite de cette procédure d'installation, le fichier jars ainsi que son contenu sont visibles à gauche dans External librairies. Ce qui montre que les dépendances SLF4J sont bien installées. Voir la capture d'écran ci-dessous.



11.8.3.2 Envoi des logs SLF4J par JUL : configuration du fichier logging.properties

Après l'installation de la librairie slf4j-jdk14 telle que nous venons de la présenter, la deuxième étape pour générer les logs via JUL est la configuration du fichier logging.properties qui est le fichier de configuration par défaut du framework JUL. Nous avons déjà montré dans les sections précédentes comment configurer le fichier logging.properties afin d'envoyer les logs vers la console et/ou vers un fichier (au besoin, revoir la section exclusivement dédiée au logging avec le framework JUL).

Dans le cas présent, nous souhaitons, en effet, envoyer les logs à la fois dans la console et dans un fichier plat nommé file.log. Ce fichier sera situé dans un dossier nommé logs positionné à la racine de notre projet Java. Pour ce faire nous devons configurer en conséquence le fichier logging.properties.

Le fichier logging.properties se situe habituellement dans le répertoire ${JAVA_HOME}/jre/lib ou dans le dossier conf du répertoire d'installation de votre JDK. Pour notre cas, le fichier est situé dans le dossier C:\Program Files\Java\jdk-20\conf.

Ouvir le fichier logging.propertie et coller la configuration ci-dessous.

C:\Program Files\Java\jdk-20\conf\logging.properties

```
# Définition des handlers
handlers= java.util.logging.ConsoleHandler,java.util.logging.FileHandler
# Fixer le niveau de log à ALL
.level= ALL

#Propriétés du handler pour le console
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

#Propriétés du handler pour le fichier
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern =
C:\\MY_JAVA_PROJECTS\\javaTuto\\logs\\file.log
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

Il s'agit ici d'une configuration utilisant deux appenders : ConsoleHandler (pour le logging sur la console) et FileHandler(pour le logging dans un fichier). Pour chacun des appenders, les messages sont formatés en utilisant la classe SimpleFormatter. Par ailleurs, le niveau de logging est fixé à ALL au niveau global. Ce qui signifie que tous les niveaux de log seront affichés par défaut. Pour davantage de détails sur la configuration du fichier logging.properties, voir la section consacrée au logging avec le framework JUL.

Après avoir configuré le fichier logging.properties, reexécuter le code CS05, la sortie dans la console ne se présente plus comme lors de l'exécution sans fournir le provider JUL. De plus les logs apparaissent dans le fichier logs/file.log. Ci-dessous les lignes qui apparaissent sur la console et dans le fichier.

Output :

```
mai 15, 2023 12:54:40 PM com.tuto.logging.Main main
INFO: Début d'exécution de la méthode main
mai 15, 2023 12:54:40 PM com.tuto.logging.Main main
FINE:  Début de création de l'objet BufferedReader
mai 15, 2023 12:54:40 PM com.tuto.logging.Main main
FINE:  Fin de création de l'objet BufferedReader
mai 15, 2023 12:54:40 PM com.tuto.logging.Main main
FINE:  Début de récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
Hello Kevin
mai 15, 2023 12:54:49 PM com.tuto.logging.Main main
FINE:  Fin de récupération de l'entrée utilisateur
mai 15, 2023 12:54:49 PM com.tuto.logging.Main main
FINE:  Début envoi salutation
mai 15, 2023 12:54:49 PM com.tuto.logging.Main main
FINE:  Fin envoi salutation
mai 15, 2023 12:54:49 PM com.tuto.logging.Main main
INFO: Fin d'exécution de la méthode main
```

11.8.4 Générer les logs SLF4J avec le provider Log4J2

Encore une fois, nous rappelons que le framework SLF4J n'est pas un framework de logging en tant que tel. C'est une couche d'abstraction qui permet de choisir le système de logging et de changer de système de logging sans impacter le code source. SLF4J ne génère donc pas de ligne de log à proprement parler. Il se base toujours sur un autre framework de logging pour logguer. Ce système est alors appelé provider. Dans cette sous-section, nous illustrons le cas où le framework Log4j2 est utilisé comme provider pour SLF4J.

11.8.4.1 Installation de la librairie de liaison entre SLFJ4 et Log4j2 : log4j-slf4j-impl

Pour pouvoir utiliser Log4j2 comme provider de SLF4J, nous devons d'abord installer dans le classpath la librairie log4j-slf4j-impl. Les étapes d'installation de cette librairie sont décrites ci-dessous.

Comme la plupart des librairies externes Java, log4j-slf4j-impl peut être installée, soit manuellement, soit automatiquement avec les outils de gestion de dépendances comme

Maven, Gradle, Ivy, etc. Nous reviendrons plus tard sur l'utilisation de Maven pour la gestion des dépendances dans un projet Java[33]. Ici, nous montrons le cas de l'installation manuelle.

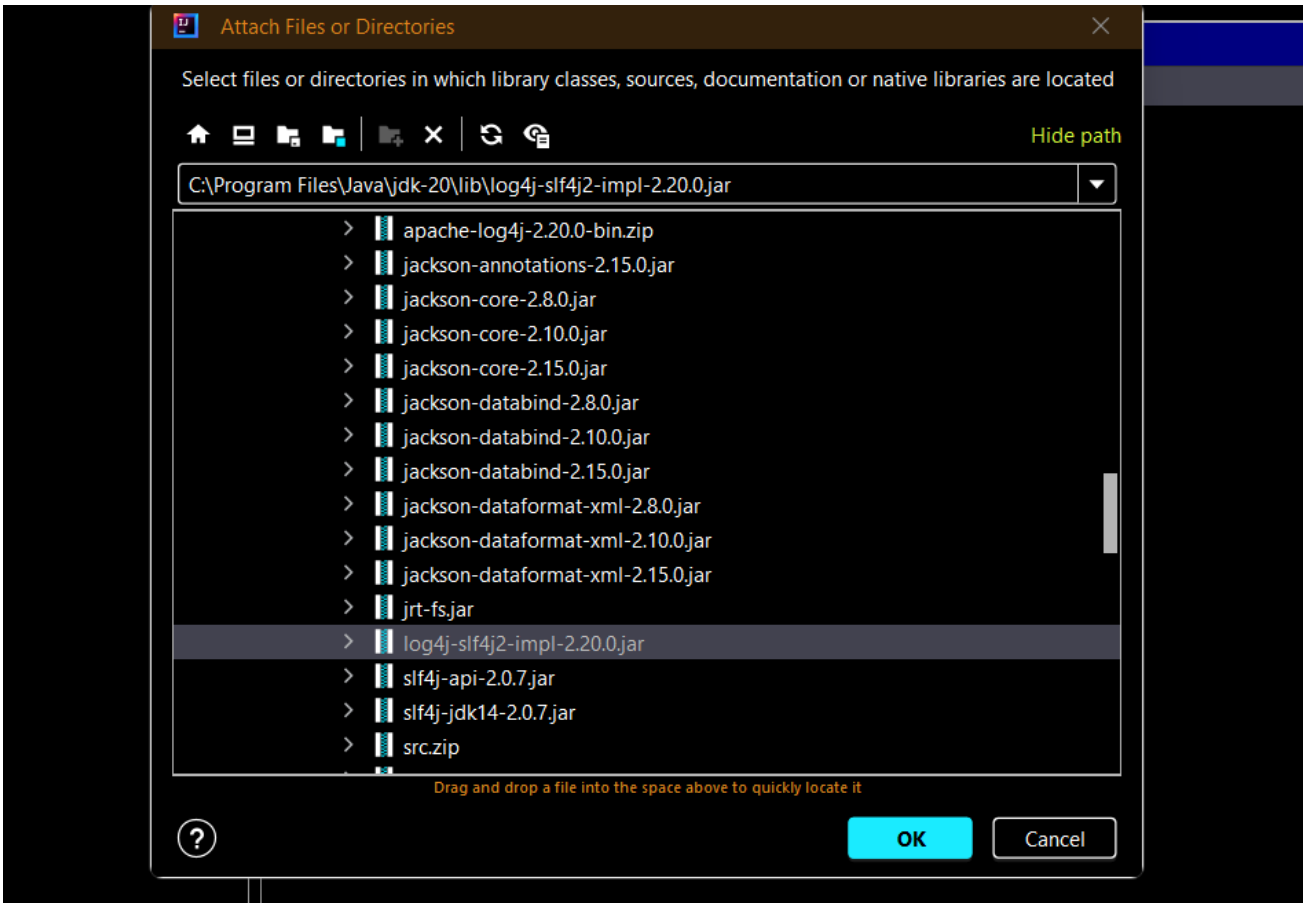
- Vérifier d'abord que le framework Log4j2 est bien installé et disponible sur votre environnement. Pour vérifier que Log4j2 est bien installé, les dépendances log4j-core-2.xx.y.jar et log4j-api-2.xx.y.jar doivent être disponibles dans le classpath. Ces jars sont ajoutés au classpath soit manuellement en les téléchargeant et en les ajoutant dans les External librairies de votre IDE, soit automatiquement à partir d'un outil de gestion de dépendance comme Maven. Si ces jars ne sont pas disponibles dans votre classpath, veuillez suivre les étapes d'installation de Log4j2 décrites plus haut dans la section exclusivement dédiée au logging avec Log4j2.
- S'assurer aussi que d'autres librairies de liaisons de SLF4J avec d'autres frameworks n'existent pas déjà dans le classpath. Par exemple, vérifier que le jar slf4j-jdk14-2.x.y.jar qui est la librairie de liaison avec le framework JUL n'existe pas dans le classepath. Si d'autres librairies de liaisons existent, lors de l'exécution SLF4J vous renvoie des warnings du type : *SLF4J: Class path contains multiple SLF4J providers*. Et SLF4J choisit un provider parmi ceux disponibles. Pour cela, il vous renvoie un message du type : *SLF4J: Actual provider is of type [org.slf4j.jul.JULServiceProvider@1a86f2f1]*. Toutefois ce choix ne correspondra pas toujours à celui que vous souhaitez au départ. Pour éliminer ce type de désagrément, il est donc préférable, si cela n'a pas d'inconvénients majeurs, de supprimer du classpth toutes les libraires de liaison ne correspondant pas à votre souhait. Ces suppressions se font généralement par une simple exclusion du jar soit des librairies externes de l'IDE, soit du pom.xml s'il s'agit d'un projet Maven. Par exemple, on peut supprimer le jar slf4j-jdk14-2.x.y.jar s'il existe déjà dans le classpath.
- Télécharger le fichier jar correspondant à l'api log4j-slf4j2-impl depuis un repository de gestion de dépendances. Ex :

  https://repo1.maven.org/maven2/org/apache/logging/log4j/log4j-slf4j2-impl/2.20.0/log4j-slf4j2-impl-2.20.0.jar
- Déposer ce fichier jar dans le dossier lib de votre installation Java ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\lib.
- Nous allons ajouter ce fichier jar dans le classpath de l'application. L'ajout peut être directement fait à partir de l'IDE. Par exemple, pour Intellij IDEA, l'ajout du fichier jar se fait comme suit :
  1. Dans la barre des menus, cliquer sur File >Project Structure >Project Settings> Modules > Dependencies.

[33] Pour ajouter la dépendance log4j-slf4j2-impl dans un projet Maven, ajouter ces spécifications dans le pom.xml

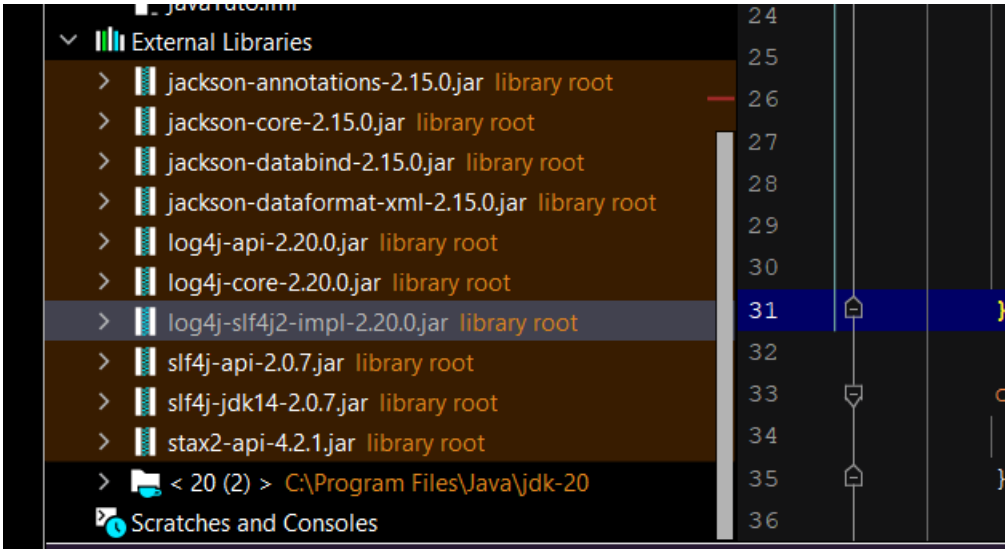
```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j2-impl</artifactId>
  <version>2.20.0</version>
  <scope>test</scope>
</dependency>
```

2. Cliquer sur le signe + (Add). Et choisir JARs or Directories. Et aller chercher le fichier log4j-slf4j2-impl-2.20.0.jar disponible dans le dossier lib. Voir capture d'écran ci-dessous.



3. Cliquer sur ok pour sélectionner et cocher la case pour pouvoir ajouter le fichier aux dépendances. Devant la librairie sélectionnée, dans scope, dans la liste déroulante choisir Compile. Cliquer sur Ok pour valider.
4. Cliquer sur ok pour valider.

A la suite de cette procédure d'installation, le fichier jar ainsi que son contenu sont visibles à gauche dans External librairies. Ce qui montre que les dépendances SLF4J sont bien installées. Voir la capture d'écran ci-dessous.



11.8.4.2 Envoi des logs SLF4J par Log4j2 : configuration du fichier log4j2.properties ou du fichier log4j2.xml

Après l'installation de la librairie log4j-slf4j-impl telle que nous venons de la présenter, la deuxième étape pour générer les logs par Log4j2 est la configuration soit du fichier log4j2.properties soit du fichier log4j2.xml. Comme l'avons déjà montré dans les précédentes sections, le logging avec Log4j2 peut être configuré en utilisant soit le fichier log4j2.properties soit le fichier log4j2.xml. Et pour choisir un fichier spécifique lors de l'exécution du programme, il suffit de créer une VM Option -Dlog4j.configurationFile dont la valeur est le chemin vers le fichier choisi. Dans les sections précédentes, nous avons déjà montré comment configurer le fichier log4j2.properties et fichier log4j2.xml afin d'envoyer les logs vers la console et/ou vers un fichier (au besoin, revoir la section dédiée au logging avec le framework Log4j2).

Dans le cas présent, nous souhaitons, en effet, envoyer les logs à la fois dans la console et dans un fichier plat nommé file.log. Ce fichier sera situé dans un dossier nommé logs positionné à la racine de notre projet Java. Pour ce faire nous devons configurer le fichier log4j2.properties ou le fichier log4j2.xml. Pour information, ces fichiers sont habituellement positionnés dans le dossier resources dans le répertoire src contenant le code source.

Ici, nous allons présenter successivement le cas où les configurations sont chargées partir du fichier log4j2.properties et le cas où les configurations sont chargées à partir du fichier log4j2.xml.

Remarque importante : Parfois malgré la définition de la VM Option pour charger les configurations à partir du fichier log4j2.properties ou du fichier log4j2.xml, il arrive que le JRE pointe toujours sur le fichier logging.properties, qui est le fichier de configuration de JUL, donc directement rattaché au JDK. Alors, si vous rencontrez les problèmes de chargement des fichiers de configuration Log4j2, il est préférable de renommer le fichier

logging.properties en par exemple logging_JUL_.properties[34]. Dès lors comme le fichier par défaut n'est plus visible par le JRE, il chargera le fichier de configuration spécifié dans les VM Options.

Cas où la configuration est chargée à partir du fichier log4j2.properties

La spécification ci-dessous montre la configuration du fichier log4j2.properties

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.properties

```
# Détails de l'Appender console
appender.console.type = Console
appender.console.name = LogToConsole
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n

# Détails de l'Appender Rolling
appender.rolling.type = RollingFile
appender.rolling.name = LogToRollingFile
appender.rolling.fileName= logs/file.log
appender.rolling.filePattern= logs/file_%d{yyyyMMdd}.log.gz
appender.rolling.layout.type = PatternLayout
appender.rolling.layout.pattern = [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t]
%c{1} - %msg%n

# RollingFileAppender rotation policy
appender.rolling.policies.type = Policies
appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
appender.rolling.policies.size.size = 50MB
appender.rolling.policies.time.type = TimeBasedTriggeringPolicy
appender.rolling.policies.time.interval = 1
appender.rolling.policies.time.modulate = true
appender.rolling.strategy.type = DefaultRolloverStrategy

# Appel des appenders
rootLogger.level = trace
rootLogger.appenderRef.stdout.ref = LogToConsole
rootLogger.appenderRef.rolling.ref = LogToRollingFile
```

Cette configuration permet d'orienter les logs à la fois vers la console (ConsoleAppender) et vers un fichier dont le chemin est logs/file.log. Les fichiers de logs sont générés avec rotation (RollingFileAppender). Au besoin, revoir la section exclusivement dédiée au logging avec Log4j2 pour plus de détails sur ce type de gestion des fichiers de log.

Tout comme nous le verrons pour le fichier log4j2.xml, pour pouvoir utiliser les configurations définies dans le fichier log4j2.properties celui-ci doit être appelé lors de l'exécution du programme en définissant une VM Option supplémentaire -Dlog4j.configurationFile. Cette option est spécifiée comme suit :

```
-
Dlog4j.configurationFile=C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\log4j2
.properties
```

[34] Le fichier logging.properties se situe généralement dans le répertoire ${JAVA_HOME}/jre/lib ou dans le dossier conf de l'installation de votre JDK. Pour notre cas, il est situé à l'emplacement C:\Program Files\Java\jdk-20\conf\logging.properties

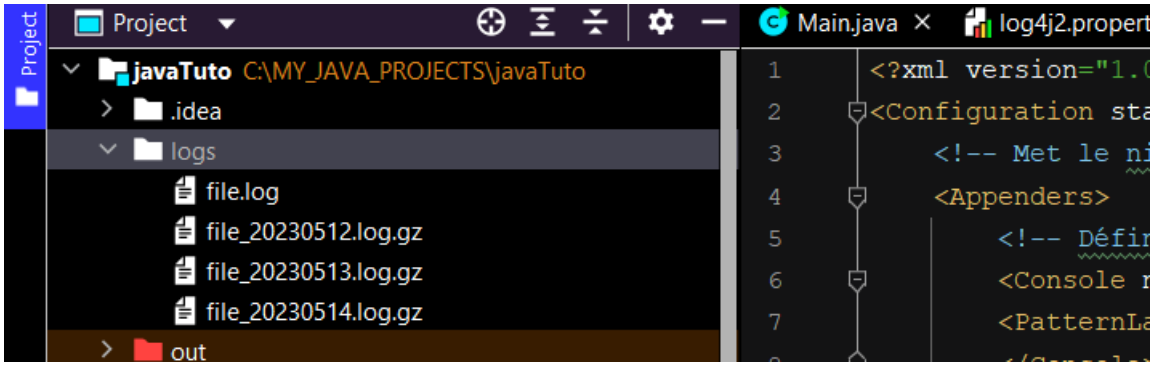
Pour définir cette VM Option dans Intellij, cliquer dans le menu Run > Edit configuration. Cliquer >Modify options > Add VM options. Et copier-coller la valeur indiquée ci-dessus.

Et pour éviter d'éventuelles interférences avec JUL, ne pas oublier de renommer le fichier logging.properties en logging_JUL.properties. Ce fichier se trouve dans le répertoire de votre JRE ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\conf\logging.properties.

Après toutes ces configurations en lançant le code source CS05, on obtient la sortie suivante dans la console et dans le fichier logs/file.log.

```
[INFO ] 2023-05-15 16:12:28.330 [main] Main - Début d'exécution de la méthode
main
[DEBUG] 2023-05-15 16:12:28.333 [main] Main -  Début de création de l'objet
BufferedReader
[DEBUG] 2023-05-15 16:12:28.334 [main] Main -  Fin de création de l'objet
BufferedReader
[DEBUG] 2023-05-15 16:12:28.334 [main] Main -  Début de récupération de
l'entrée utilisateur
[DEBUG] 2023-05-15 16:12:38.685 [main] Main -  Fin de récupération de l'entrée
utilisateur
[DEBUG] 2023-05-15 16:12:38.685 [main] Main -  Début envoi salutation
[DEBUG] 2023-05-15 16:12:38.686 [main] Main -  Fin envoi salutation
[INFO ] 2023-05-15 16:12:38.686 [main] Main - Fin d'exécution de la méthode
main
```

Et en exécutant plusieurs jours de suite le code source CS05, les fichiers de log seront générés en rotation. Voir une capture d'écran pour un lancement de quatre jours de suite.



Le fichier file.log contient les lignes de logs générées pour la date courante %d{yyyyMMdd} tandis que les fichiers file_20230514.log.gz, file_20230513.log.gz et file_20230513.log.gz contiennent les lignes de log des jours précédents. Ces fichiers sont générés chaque jour par rotation du fichier file.log de la veille avant l'exécution du programme à la date courante.

Cas où la configuration est chargée à partir du fichier log4j2.xml

La spécification ci-dessous montre la configuration du fichier log4j2.xml

C:\MY_JAVA_PROJECTS\javaTuto\src\resources\log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <!-- Met le niveau de log à WARN pour la configuration -->
    <Appenders>
        <!-- Définit un Appender nommé LogToCOnsole -->
        <Console name="LogToConsole" target="SYSTEM_OUT">
        <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %C -
%msg%n" />
        </Console>
        <!-- Définit un Appender nommé LogToRollingFile -->
        <RollingFile name="LogToRollingFile" fileName="logs/file.log"
filePattern= "logs/file_%d{yyyyMMdd}.log.gz">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%C - %msg%n" />
            <!-- Définition des critères de rotation -->
            <Policies>
                <!-- Taille maximale du fichier courant fixé à 50MB -->
                <SizeBasedTriggeringPolicy size="50MB" />
                <!-- Rotation chaque 1 jour -->
                <TimeBasedTriggeringPolicy interval="1" modulate="true" />
            </Policies>
            <DefaultRolloverStrategy >
            </DefaultRolloverStrategy>
        </RollingFile>
    </Appenders>
    <Loggers>
        <!-- Appel des Appenders-->
        <Root level="debug" additivity="false">
            <AppenderRef ref="LogToConsole" />
            <AppenderRef ref="LogToRollingFile" />
        </Root>
    </Loggers>
</Configuration>
```

Cette configuration permet d'orienter les logs à la fois vers la console (ConsoleAppender) et vers un fichier dont le chemin est logs/file.log. Les fichiers de logs sont générés avec rotation (RollingFileAppender). Au besoin, revoir la section exclusivement dédiée au logging avec Log4j2 pour plus de détails sur ce type de gestion des fichiers de log.

Tout comme nous l'avons vu pour le fichier log4j2.properties, pour pouvoir utiliser les configurations définies dans le fichier log4j2.xml, celui-ci doit être appelé lors de l'exécution du programme en définissant une VM Option supplémentaire -Dlog4j.configurationFile. Cette option est spécifiée comme suit :

```
-
Dlog4j.configurationFile=C:\\MY_JAVA_PROJECTS\\javaTuto\\src\\resources\\log4j2
.xml
```

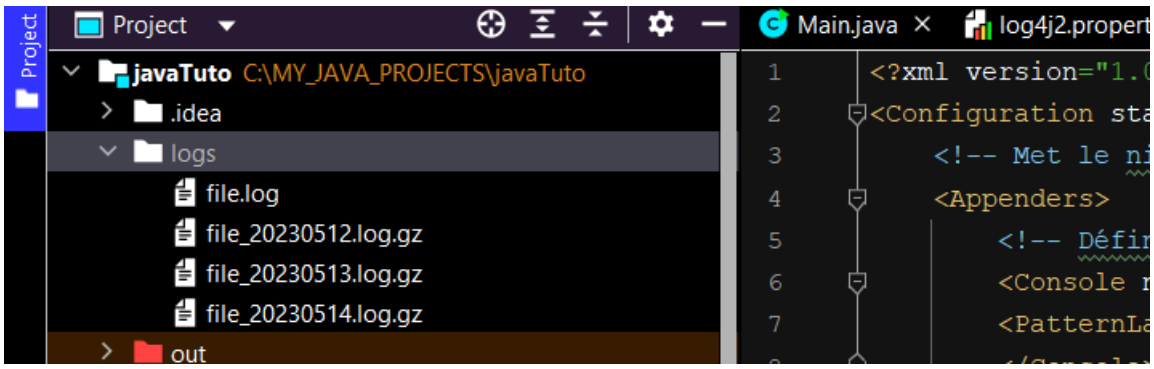
Pour définir cette VM Option dans Intellij, cliquer dans le menu Run > Edit configuration. Cliquer >Modify options > Add VM options. Et copier-coller la valeur indiquée ci-dessus.

Et pour éviter d'éventuelles interférences avec JUL, ne pas oublier de renommer le fichier logging.properties en logging_JUL.properties. Ce fichier se trouve dans le répertoire de votre JRE ou de votre JDK. Ex : C:\Program Files\Java\jdk-20\conf\logging.properties.

Après toutes ces configurations en lançant le code source CS05, on obtient la sortie suivante dans la console et dans le fichier logs/file.log.

```
2023-05-15 17:33:49.708 [main] INFO  com.tuto.logging.Main - Début d'exécution
de la méthode main
2023-05-15 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Début de création
de l'objet BufferedReader
2023-05-15 17:33:49.711 [main] DEBUG com.tuto.logging.Main -  Fin de création
de l'objet BufferedReader
2023-05-15 17:33:49.713 [main] DEBUG com.tuto.logging.Main -  Début de
récupération de l'entrée utilisateur
Saisissez votre nom, svp : Kevin
2023-05-15 17:33:54.067 [main] DEBUG com.tuto.logging.Main -  Fin de
récupération de l'entrée utilisateur
2023-05-15 17:33:54.068 [main] DEBUG com.tuto.logging.Main -  Début envoi
salutation
Hello Kevin
2023-05-15 17:33:54.069 [main] DEBUG com.tuto.logging.Main -  Fin envoi
salutation
2023-05-15 17:33:54.070 [main] INFO  com.tuto.logging.Main - Fin d'exécution de
la méthode main
```

Et en exécutant plusieurs jours de suite le code source CS05, les fichiers de log seront générés en rotation. Voir une capture d'écran pour un lancement de quatre jours de suite.



Le fichier file.log contient les lignes de logs générées pour la date courante %d{yyyyMMdd} tandis que les fichiers file_20230514.log.gz, file_20230513.log.gz et file_20230513.log.gz contiennent les lignes de log des jours précédents. Ces fichiers sont générés chaque jour par rotation du fichier file.log de la veille avant l'exécution du programme à la date courante.

12 LES ANNOTATIONS

12.1 Généralités

Une annotation est un marqueur permettant d'associer des métadonnées à des éléments d'un programme de sorte que la JVM leur réserve un traitement spécifique lors de la compilation ou de l'exécution du code.

Depuis Java 8, tous les éléments de code d'un programme peuvent être annotés qu'il s'agisse d'un package, d'une classe, d'une méthode, d'un champ ou d'une variable locale. Les annotations peuvent jouer plusieurs rôles dans un programme Java. Elles peuvent permettre au compilateur de procéder à certaines vérifications lors de la compilation du code et de renvoyer des warnings ou des erreurs. Par exemple, lorsqu'une interface a été déclarée comme une interface fonctionnelle et annotée comme telle, lors de l'usage de cette interface, le compilateur vérifie bien qu'il s'agit d'une interface fonctionnelle, sinon il renvoie une erreur de compilation. Les annotations servent aussi à d'autres usages : documenter un code, générer automatiquement du code ou des fichiers de configuration, injecter des dépendances dans un programme, valider un code, etc..

Les frameworks Java utilisant le plus les annotations sont Junit (framework dédié aux tests unitaires), Hibernate ORM (framework de persistance des objets en base de données), Spring MVC (framework de développement de web services), FindBugs (framework d'analyse statique de bytecode pour détecter des bugs) ou JAXB (framework pour créer des classes Java à partir de schémas XML et inversement créer des schémas XML à partir des classes Java).

Ce chapitre est consacré à la présentation et à l'utilisation standard des annotations dans un programme Java.

12.2 Annoter un élément de code Java

Pour annoter un élément de code Java, il suffit de spécifier sur la ligne précédant sa déclaration le nom de l'annotation qui lui est associé. Ci-dessous la syntaxe générale de déclaration d'un élément de code annoté.

Syntaxe d'annotation d'une classe

```
@myAnnotation
public class MyClass {
    ...
}
```

Syntaxe d'annotation d'une méthode

```
@myAnnotation

public void myMethod( parametres){
```

```
    instructions
};
```

Syntaxe d'annotation d'un champ ou d'une variable locale

```
@myAnnotation
String myVariable;
```

Comme on peut le remarquer dans les différentes syntaxes ci-dessus, une annotation est déclarée avec le symbole @ suivi du nom de l'annotation. Habituellement, l'annotation est spécifiée sur la ligne précédant la déclaration de l'élément de code. Mais cela n'est pas obligatoire. Il est possible de spécifier sur la même ligne l'annotation et la déclaration de l'élément de code. Voir syntaxe ci-dessous.

```
@myAnnotation public void myMethod( parametres){
    instructions
};
```

Par ailleurs, il est possible d'associer plusieurs annotations à un même élément de code. La syntaxe ci-dessous illustre cette situation.

```
@myAnnotation1
@myAnnotation2
@myAnnotation3
public void myMethod( parametres){
    instructions
};
```

12.3 Quelques annotations standards Java (annotations built-in)

Depuis la version 5, plusieurs annotations sont disponibles dans Java. Cette section a pour but de présenter quelques-unes d'entre elles et leur mode d'usage.

12.3.1 @Override

L'annotation @Override est utilisée sur une méthode de classe dérivée pour indiquer qu'il s'agisse d'une redéfinition d'une méthode déjà existante dans une classe abstraite[35]. Pour illustrer l'usage de l'annotation @Override supposons une classe abstraite A définie comme suit :

```
package com.tuto.annotation;

public abstract class A {
    public abstract int calculSomme (int x, int y) ;
}
```

[35] Voir dans les chapitres précédentes les détails sur les classes abstraites.

Cette classe abstraite nommée A prévoit une méthode calculSomme() dont le but est de calculer la somme de deux entiers x et y. Comme on peut le remarquer, cette méthode n'est pas encore implémentée ; d'où l'usage du mot-clé abstract devant sa déclaration de la classe.

Supposons maintenant qu'on crée une classe concrète B dont le but est d'étendre la classe abstraite A et d'implémenter sa méthode calculSomme(). La classe B peut alors être définie comme suit :

```
package com.tuto.annotation;

public class B extends A{

    @Override
    public int calculSomme (int x, int y){
        return x+y;
    } ;
}
```

12.3.2 @Deprecated

Dans les programmes, librairies et applications Java, l'annotation @Deprecated est une annotation de documentation utilisée pour indiquer à l'utilisateur que l'élément de code en question est déprécié et qu'il est conseillé de l'abandonner au profit d'un nouvel élément de code de même type mais plus évolué. Très généralement l'annotation @Deprecated est utilisée sur les méthodes d'une classe. Pour montrer l'usage de l'annotation @Deprecated, prenons le cas d'une classe A définie comme suit :

```
package com.tuto.annotation;

public class A {

    public double calculSomme(double x, double y) {
        return x + y;
    }
}
```

La classe définit une méthode nommée calculSomme() dont le but est d'additionner deux nombres x et y.

Supposons maintenant que l'on veuille généraliser la méthode calculSomme() en faisant la somme de n'importe quel type numérique. La classe A peut être redéfinie comme suit :

```
package com.tuto.annotation;

public class A {

    @Deprecated
    public double calculSomme(double x, double y) {
        return x + y;
    }

    public double calculSomme(Object x, Object y) {
        double num_x=0;
        double num_y=0;
```

```
        try{
           num_x= (double)x;
           num_y=(double)y;
        }
        catch(Exception e){
            System.out.println("Problème de cast de la valeur de x ou de y");
            System.exit(1);
        };
        return num_x + num_y;
    }
}
```

Dans cette nouvelle définition de la classe A, nous avons deux définitions de la méthode calculSomme(). La première est l'ancienne version qui fait la somme de deux nombres de type double pris en paramètres. La deuxième définition est plus générale et plus générique. Il prend en argument n'importe quels objets x et y, les caste d'abord en type double avant de faire la somme. Mais la somme des deux objets n'est effectuée que lorsque l'opération de cast est un succès, sinon le traitement renvoie une erreur.

Mais dans cette redéfinition de la classe A, nous avons gardé les deux définitions de la méthode calculSomme() car les deux restent valables. En effet, l'utilisateur a la possibilité de faire d'abord ses propres cast en double sur les arguments x et y avant de les passer à la méthode calculSomme() ou il peut simplement passer en vrac les deux objets x et y. Et la méthode calculSomme() se charge de faire le cast et la somme. La deuxième définition de la méthode semble donc plus évoluée que la première. Et comme nous souhaitons que les utilisateurs utilisent plutôt la version évoluée de la méthode, nous leur envoyons une information en leur indiquant que la première définition est dépréciée. D'où l'usage de l'annotation @Deprecated devant la première définition de la méthode calculSomme().

12.3.3 @SuppressWarnings

L'annotation @SuppressWarnings est utilisée sur un élément de code (classe ou méthode) pour indiquer au compilateur de ne pas renvoyer de warnings lors de la compilation de l'élément de code concerné. Cette annotation permet en quelque sorte d'indiquer au compilateur de ne pas s'inquiéter d'éventuelles risques que peuvent comporter les instructions définies dans l'élément de code.

Pour illustrer l'usage de l'annotation @SuppressWarnings prenons l'exemple d'une classe A dont les membres sont : un champs nommé elements, une méthode nommée ajouteElement et une méthode nommée getElements. La classe A est définie comme suit :

```
package com.tuto.annotation;

import java.util.ArrayList;
import java.util.List;
public class A {
    private List elements =new ArrayList();

    public void ajouteElement(String e) {
        this.elements.add(e);
    }
```

```
    public List getElements() {
        return this.elements;
    }
}
```

Comme on peut le remarquer, le champ element est un objet de type ArrayList. Il est initialisé en appelant l'opérateur new sur la classe ArrayList. Mais dans cette instanciation, les types des éléments qui vont constituer l'ArrayList ne sont pas déclarés. On dit que le type des éléments n'est pas contrôlé à la déclaration. Ce qui signifie, qu'à priori, lors de la première opération d'ajout d'élément, cette liste est prête à accueillir les éléments de n'importe quel type : primitif ou type classe. Une telle définition comporte donc des risques et le compilateur se charge de nous le rappeler. Ce risque apparaît surtout au niveau de la méthode ajouteElement() dont le but est d'ajouter des éléments au champ elements. Ici, la méthode ajouteElement() ajoute des éléments de type int. Ce qui signifie implicitement que le type des éléments du champ elements dépend du type des éléments ajoutés par la méthode ajouteElement().

Cependant lorsque nous sommes conscients du risque que comporte la déclaration du champ elements, et surtout de l'absence de conséquence néfaste liée à cette déclaration, nous pouvons rassurer le compilateur en associant l'annotation @SuppressWarnings à la méthode ajouteElement(). Ainsi, la classe A se présentera comme suit :

```
package com.tuto.annotation;

import java.util.ArrayList;
import java.util.List;
public class A {
    private List elements =new ArrayList();
    @SuppressWarnings({})
    public void ajouteElement(int e) {
        this.elements.add(e);
    }
    public List getElements() {
        return this.elements;
    }
}
```

12.3.4 @SafeVarargs

L'annotation @SafeVarargs est une annotation qui, comme l'annotation @SuppressWarnings, vise à rassurer le compilateur sur certaines instructions qui présentent visiblement des risques. L'annotation @SafeVarargs s'applique spécifiquement aux méthodes dont le nombre d'arguments, c'est-à-dire de type Varargs. Par exemple, considérons une classe A contenant une méthode nommée calculSomme() qui vise à calculer la somme d'un nombre variable d'entiers pris en paramètre. La signature de cette méthode peut être de type Varargs telle qu'indiquée ci-dessous :

```
package com.tuto.myannotations;

public class A {
```

```
    public final int calculSomme (int...valueList){
        int total = 0;
        for (int i : valueList)
            total += i;
        return total;
    } ;
}
```

Les arguments de la méthode calculSomme() sont de type Varargs car à chaque appel de la méthode, on peut spécifier un nombre variable d'arguments.

Mais le fait de spécifier une méthode dont le nombre d'arguments est variable n'est pas rassurant pour le compilateur. Ainsi pour rassurer le compilateur, on peut faire appel à l'annotation @SafeVarargs pour indiquer que la méthode ne comporte pas de risque. Ainsi, avec la classe A définie ci-dessous, on peut appeler l'annotation @ SafeVarargs comme suit :

```
package com.tuto.myannotations;

public class A {
    @SafeVarargs
    public final int calculSomme(int... valueList){
        int total = 0;
        for (int i : valueList)
            total += i;
        return total;
    } ;
}
```

12.3.5 @FunctionalInterface

L'annotation @FunctionalInterface est utilisée pour déclarer une interface ne comportant qu'une seule méthode. Pour information, une interface est une classe dont toutes les méthodes sont déclarées mais dont aucune n'est implémentée. Et une interface abstraite est une interface qui ne comporte qu'une seule méthode. L'utilisation des interfaces fonctionnelles est parfois très utile car elle rend possible la programmation fonctionnelle travers l'usage des expressions lambda[36]. L'exemple ci-dessous montre la déclaration d'une interface fonctionnelle nommée A ainsi que la spécification de l'annotation @FunctionalInterface.

```
package com.tuto.annotation;

@FunctionalInterface
public interface A {
    public int calculSomme (int x, int y);
}
```

L'usage de l'annotation @FunctionalInterface dans la définition de la classe A permet au compilateur de vérifier qu'il s'agit bien d'une interface fonctionnelle.

[36] Voir les chapitres précédents pour plus de détails sur les expressions lambda.

Le fait que la classe A soit déclarée avec l'annotation @FunctionalInterface est utile à plusieurs égards. En particulier, le fait de savoir qu'une interface est fonctionnelle permet d'implémenter la méthode unique abstraite sans avoir à implémenter une classe concrète. En effet, il suffit simplement d'instancier une classe anonyme et d'utiliser les expressions lambda pour implémenter la méthode. L'exemple ci-après permet de créer un objet de A et d'implémenter la méthode calculSomme() et l'appeler sans instancier une classe concrète.

```
package com.tuto.annotation;

public class Main {
    public static void main(String[] args)  {
        A op = (int x, int y )-> {return x+y;};
        int somme = op.calculSomme(5, 7);
        System.out.println( somme);
    }
}
```

Output

```
12
```

12.4 Les annotations paramétrés

Les annotations paramétrées sont des annotations auxquelles on associe un ensemble d'attributs spécifiés sous formes de clé-valeur. La plupart des annotations que nous avons présentées jusque-là sont des annotations simples c'est-à-dire spécifiées uniquement avec la syntaxe @NomAnnotation sans aucune information supplémentaire. On les appelle annotation non paramétrées. Mais Java offre aussi la possibilité de spécifier une annotation en lui associant des attributs qui sont plutôt des champs auxquels on assigne des valeurs. Ce sont des annotations paramétrées. Cette section a pour but d'étudier les annotations paramétrées.

12.4.1 Syntaxe d'appel d'une annotation paramétrée

La syntaxe ci-dessous présente le mode d'appel d'une annotation paramétrée dans le code.

```
@NomAnnotation (param1=valeur1, param2=valeur2,...,paramN=valeurN)
```

Dans sa forme générale, les paramètres d'une annotation sont spécifiés en indiquant le nom du paramètre en lui assignant une valeur. Cette valeur peut être de n'importe quel type : primitif, String ou tableaux. Ci-dessous quelques formes de spécification des paramètres d'une annotation.

```
@NomAnnotation // 'Annotation sans paramètres
@NomAnnotation(param1 = valeur1, param2 = valeur2) // Annotation avec deux
paramètres
@NomAnnotation(param1 = valeur1) // Annotation avec un paramètre
@NomAnnotation(valeur1) // Annotation avec un paramètre
```

```
@NomAnnotation(param1 = {e1, e2, e3}) // Annotation avec un seul paramètre dont la valeur est de type tableau avec trois éléments
@NomAnnotation ({e1}) // Annotation avec un seul paramètre dont la valeur est de type tableau avec 1 seul élément
```

On remarque à partir de ces différents exemples que lorsque l'annotation a un seul paramètre, il n'est pas obligatoire d'indiquer le nom du paramètre. Il suffit simplement d'indiquer la valeur. C'est le cas par exemple de l'annotation @NomAnnotation(param1 = valeur1) qui peut être spécifié simplement comme @NomAnnotation(valeur1).

12.4.2 Appel d'une annotation paramétrée

Les exemples ci-dessous montrent quelques exemples d'appels d'une annotation paramétrée.

```
@MyAnnotation1 (value=10)
public void myMethod1(){
    instructions
}

@MyAnnotation2 (param1=10, param2="valueString")
public void myMethod2(){
    instructions
}

@MyAnnotation3 (name="Jean", score={2, 4,8,1})
public void myMethod3(){
    instructions
}

@MyAnnotation4 (id="dgfh23",city="New-York")
public void myMethod4(){
    instructions
}
```

Comme on peut le remarquer, @MyAnnotation1 a un paramètre nommé value auquel on attribue la valeur 10. @MyAnnotation2 a deux paramètres nommés param1 et param2 qui prennent respectivement les valeurs 10 et valueString. L'annotation @MyAnnotation3 a également deux paramètres name et score. Le paramètre name prend la valeur Jean et le paramètre score est de type tableau qui prend la valeur {2, 4,8,1}. Ces différents exemples montrent que chaque annotation peut avoir ses propres paramètres. Ces paramètres peuvent prendre n'importe quel nom et les valeurs qui leur sont associées peuvent être de n'importe quel type.

Il faut noter que les paramètres dont les valeurs sont spécifiées lors de l'appel de l'annotation ne sont pas déclarées par hasard. Les paramètres ont été préalablement déclarées lors de la création de l'annotation (voir plus bas la section dédiée à la création d'une annotation). L'appel d'une annotation est similaire à l'appel du constructeur d'une classe. C'est lors de l'appel du constructeur que les valeurs des champs sont définies. Il en est de même pour une annotation paramétrée. C'est en appelant l'annotation et en spécifiant les valeurs des paramètres que ces valeurs sont assignées aux attributs.

12.4.3 Créer sa propre annotation : usage du mot clé @interface

En plus de nombreuses annotations built-in disponibles, Java laisse la possibilité à l'utilisateur de spécifier ses propres annotations et de les personnaliser en ajoutant autant de paramètres qu'il souhaite. Cette sous-section vise à montrer comment créer et utiliser ses propres annotations.

Note : Il est important de faire remarquer que la création d'une annotation présente plusieurs similarités avec la création qu'une classe Java. D'abord la règle de nommage d'une annotation est la même qu'une classe : le nom commence par une lettre majuscule. Et si le nom est composé de plusieurs mots, chaque mot commence par une lettre en majuscule. De plus, tout comme une classe Java, une annotation est créée dans un fichier source dédié avec l'extension .java. Ce fichier est déposé dans un package. Tous les éléments de code (classe, méthodes, etc..) appartenant au même package que l'annotation définie peuvent utiliser l'annotation sans avoir besoin d'utiliser l'instruction import. Seuls les éléments de code qui n'appartiennent pas au même package qui utilisent l'instruction import pour rendre accessible l'annotation.

12.4.3.1 Créer une annotation non paramétrée

Une annotation non paramétrée est une annotation à laquelle n'est associée aucun attribut. Par exemple, les annotations built-in @Override ou @FunctionalInterface sont des annotations non paramétrées. L'exemple ci-dessous montre comment créer une annotation sans paramètres

```
package com.tuto.myannotations;
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface MyAnnotation {

}
```

Une annotation est déclarée avec le mot-clé @interface suivi du nom qu'on lui attribue. Le nom attribué à l'annotation est myAnnotation. Il s'agit ici d'une annotation sans paramètres car aucun attribut n'est défini dans le corps de l'annotation. Nous verrons par la suite la création d'une annotation avec paramètres.

Comme on peut le remarquer, la création de l'annotation MyAnnotation est précédée de l'appel d'un certain nombre d'annotation built-in. Il s'agit notamment des annotations @Retention, @Target, @Inherited et @Documented. Ci-dessous le rôle de chacune de ces annotations dans la gestion du comportement de l'annotation @MyAnnotation.

- **@Retention :** cette annotation permet d'indiquer le domaine d'application d'une annotation dans le cycle de vie du code. Le cycle de vie d'un code est caractérisé par trois étapes : l'étape de code source : à cette étape les fichiers sources sont définis avec l'extension .java ; l'étape de compilation en bytecodes : à cette étape les fichiers sources sont définis avec l'extension .class et l'étape de runtime : qui correspond à l'étape d'exécution des bytecode dans la JVM.

  L'annotation @Retention permet d'indiquer à quelle étape est utilisée l'annotation que nous créons. Suivant les trois étapes du cycle de vie du code, il existe alors trois valeurs possible pour l'annotation @Retention :

  - @Retention (RetentionPolicy.SOURCE) : indique l'annotation est utilisée uniquement à l'étape de code source
  - @Retention (RetentionPolicy.CLASS) : l'annotation est utilisée à l'étape de compilation du code.
  - @Retention (RetentionPolicy.RUNTIME) : l'annotation est utilisée à l'étape d'exécution du code.
- **@Target :** l'annotation @Target permet d'indiquer à quel élément de code l'annotation créée sera associée : package, classe, méthodes, champs, variables locales, etc… A la différence @Retention qui indique l'étape d'application de l'annotation dans le cycle de vide du code, @Target permet d'indiquer les éléments spécifiques du code. Différentes valeurs sont possibles :
  - @Retention(ElementType.PACKAGE) : indique que l'annotation est appliquée à tout le package.
  - @Retention(ElementType.TYPE) : l'annotation est appliquée aux classes, interfaces et Enum.
  - @Retention(ElementType.CONSTRUCTOR) : s'applique à des constructeurs de classe
  - @Retention(ElementType.METHOD) : s'applique à des méthodes de classe
  - @Retention(ElementType.FIELD) : s'appliquer à des champs classe.
  - @Retention(ElementType.LOCAL_VARIABLE) : s'applique à des variables locales
  - @Retention(ElementType.TYPE_PARAMETER) : S'applique à des types génériques
  - @Retention(ElementType.TYPE_USE) : s'applique à tout usage de type comme les déclarations, les cast, les types génériques, etc..
  - @Retention(ElementType.ANNOTATION_TYPE)

  A noter qu'on peut indiquer plusieurs éléments type cibles. Dans ce cas, les valeurs doivent être spécifiées sous forme de tableau. Par exemple la spécification ci-dessous permet d'indiquer que l'annotation est applicable aux champs et aux types :

  @Target({ElementType.TYPE, ElementType.FIELD })

- **@Inherited :** Cette annotation permet d'indiquer que lorsque l'annotation créée et utilisée sur un élément de code (par exemple : TYPE), alors tout élément de code qui hérite du premier élément annoté héritera également de l'annotation spécifiée. Par exemple, supposons qu'on définisse une classe A annotée avec l'annotation MyAnnotation telle que :

```
@MyAnnotation
public class A {…}
```

Si on définit une classe B qui hérite de la classe A telle que

```
public class B extends A {…}
```

Alors la classe B héritéra aussi de l'annotation MyAnnotation. Telle est l'utilité de l'usage de l'annotation @Inherited lors de la définition de MyAnnotation.

- **@Documented :** cette annotation permet de rendre accessible dans la documentation Java les caractéristiques de l'annotation créée. Grâce à @Documented, on peut accéder aux attributs et aux métadonnées de l'annotation depuis la documentation Java.

Appel de l'annotation non paramétrée :

L'annotation MyAnnotation étant créée, nous pouvons maintenant l'utiliser dans n'importe quelle classe. Ci-dessous un exemple d'appel depuis une classe hypothétique nommée myClass.

```
package com.tuto.utils;

import com.tuto.myannotations.MyAnnotation;
@MyAnnotation
public class MyClass {
    public void infos() {
      System.out.println("Cette classe utilise l'annotation");
    }
}
```

Comme le montre l'exemple, pour appeler une annotation non paramétrée, il suffit simplement de spécifier le nom de l'annotation précédé du symbole @ sans aucune information supplémentaire. Ici, MyAnnotation est appelée avec simplement @MyAnnotation.

12.4.3.2 Créer une annotation paramètrée

Pour rappel, une annotation paramétrée est une annotation à laquelle sont associées des attributs se présentant sous forme de clés-valeurs. Il appartient à l'utilisateur d'indiquer les valeurs de ces attributs lors de l'appel de l'annotation. L'exemple ci-dessous montre la création d'une annotation paramétrée.

Note : Etant donné que nous partons de l'annotation @MyAnnotation précdéemment définie sans spécifier de paramètres, toutes les remarques formulées à propos de MyAnnoation sont aussi valables dans cette section à savoir le rôle des annotations @Retention, @Target, @Inherited et @Documented (voir la sous-section précédente pour plus de détails sur l'usage de ces annotations).

```
package com.tuto.myannotations;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface MyAnnotation {
    String id ();
    String date();
    String name() default "";
    int value() default 0;
}
```

Dans cet exemple, nous créons l'annotation MyAnnotation en lui associant quatre attributs nommés respectivement id, date, name, et value. A noter que ces attributs pouvaient prendre n'importe quel nom. En effet, le choix du nom des attributs dépend de l'utilisateur sans aucune restriction.

Comme nous l'avons déjà évoqué plus haut, la création d'une annotation présente plusieurs similarités avec la création d'une classe. Dans une annotation, les attributs sont déclarés dans le même style que les champs d'une classe. Chaque attribut est déclaré avec son type. Dans l'exemple ci-dessus, les attributs id, date et name sont de type String, tandis que l'attribut value est de type int. Cependant, il existe quelques différences notables entre la déclaration des attributs d'une annotation et celle des champs d'une classe. D'abord, les attributs de l'annotation sont déclarés avec le symbole (). De même, il est possible d'attribuer des valeurs par défaut à un attribut d'une annotation. Par exemple, les attributs name et value ont été déclarés avec des valeurs par défaut. Pour associer une valeur par défaut à un attribut, il suffit de spécifier l'instruction default. Dans l'exemple, la valeur par défaut de l'attribut name est "", tandis que la valeur par défaut de l'attribut value est 0.

Appel de l'annotation paramétrée :

Pour appeler une annotation paramétrée, on indique le nom de l'annotation précédé du symbole @ et suivi par une parenthèse dans laquelle on indique chaque attribut et sa valeur. L'exemple ci-dessous montre l'appel de l'annoation MyAnnotation précédemment définie.

```
package com.tuto.utils;

import com.tuto.myannotations.MyAnnotation;
@MyAnnotation(id = "kh2396", date = "2023-05-19",name = "asset",value = 35)
public class MyClass {
    public void infos() {
      System.out.println("Cette classe appelle utilise l'annotation avec des
paramètres");
```

```
    }
}
```

Nous appelons MyAnnotation en spécifiant la valeur de chacun de ses quatre attributs id, date, name et value.

Rappelons que lors de l'appel d'une annotation paramétrée, il n'est pas obligatoire de spécifier les attributs pour lesquels les valeurs par défaut ont été définies. C'est le cas par exemple des attributs name et value pour lesquels les valeurs par défaut ont été définies (voir plus haut la définition de l'annotation MyAnnotation).

Lorsque nous souhaitons garder les valeurs par défaut des attributs name et value, l'appel de MyAnnotation peut se présenter comme suit :

```
package com.tuto.utils;

import com.tuto.myannotations.MyAnnotation;
@MyAnnotation(id = "kh2396", date = "2023-05-19")
public class MyClass {
    public void infos() {
      System.out.println("Cette classe appelle utilise l'annotation avec des
paramètres");
    }
}
```

Dans cet appel, les attributs possédant des valeurs par défaut peuvent être ignorées lors de l'appel si l'on souhaite conserver ces valeurs par défaut.

13 LES THREADS JAVA

13.1 Généralités sur les threads Java

Les threads Java sont des processus légers autonomes permettant d'exécuter plusieurs tâches en parallèle au sein de la JVM qui, lui est considéré comme un processus lourd. Les threads partagent en commun l'ensemble de ressources fournies par la JVM : bytecode à exécuter, données, fichiers ouverts, etc.

Les threads offrent des fonctionnalités multitâches inspirées des microprocesseurs. En effet, tous les microprocesseurs de dernières générations sont capables d'exécuter plusieurs programmes en simultanée, chaque programme représentant alors un processus spécifique. Historiquement, sur les ordinateurs de premières générations qui tournaient avec des monoprocesseurs, la simultanéité d'exécution des tâches était seulement apparente. A un instant donné, un seul programme utilise toutes les ressources de l'unité centrale. Et sur un intervalle de temps suffisamment courts, l'environnement switche d'un programme à un autre. Le switch peut également se faire durant l'attente d'un programme lors d'une opération input/output (saisie utilisateur, lecture ou écriture sur disque, attente de fin de transfert d'un fichier Web…). C'est la bascule d'un programme à un autre programme dans un laps de temps relativement très court qui donnait l'impression que les programmes s'exécutaient en simultanéité. Les threads Java se basent sur le même principe d'exécution multiple au sein d'un même JVM.

Il faut noter que chaque programme Java dispose obligatoirement d'un thread principal qui permet, en fait, d'exécuter le programme principal (la méthode main). Mais en déhors du thread principal, on peut créer plusieurs autres threads afin d'exécuter des tâches en parallèles pour améliorer la performance d'exécution du programme. Le but de ce chapitre est de présenter la création et l'exécution des threads à l'intérieur d'un programme Java.

13.2 Créer un thread

Il existe deux façons de créer un thread dans un programme Java: soit étendre la classe Thread déjà dispose dans le package natif java.lang ; soit implémenter l'interface Runnable également disponible dans le package java.lang. Les deux sous-sections ci-dessous présentent chacune des deux approches de création de thread.

13.2.1 Créer un thread en étendant la classe Thread

Pour créer un thread à partir de la classe Thread il suffit de définir une nouvelle classe en utilisant le mot-clé extends sur la classe Thread et de redéfinir la méthode run () déjà disponible dans la classe Thread. Et pour exécuter le thread créé, on appelle la méthode start() après l'instanciation de l'objet. L'exemple ci-dessous illustre la création et l'exécution d'un thread nommé myThread.

Définition de la classe de thread

```
package com.tuto.thread;
class MyThread extends Thread {
    public void run() {
        System.out.println("Afficher un nombre chaque seconde : de 1 à 10");
        for (int i = 1; i <= 10; i++) {
            System.out.println("Le nombre affiché est "+i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException
                    e) {
            }
        }
    }
}
```

Instanciation de la classe de thread et execution

```
package com.tuto.thread;
public class Main {
        public static void main(String[] args)  {

                // Instanciation de la du thread

                Thread myThread= new MyThread();
                // Execution du thread
                myThread.start();


        }
}
```

Output :

```
Afficher un nombre chaque seconde: de 1 à 10
Le nombre affiché est 1
Le nombre affiché est 2
Le nombre affiché est 3
Le nombre affiché est 4
Le nombre affiché est 5
Le nombre affiché est 6
Le nombre affiché est 7
Le nombre affiché est 8
Le nombre affiché est 9
Le nombre affiché est 10
```

Dans l'exemple ci-dessus, nous créons d'abord une classe nommée MyThread qui étend la classe Thread et qui redéfinit la méthode run(). L'objectif de la création de ce thread est d'afficher les nombres entre 1 et 10 en faisant une pause de 1000 millisecondes (1 seconde) entre chaque affichage. La pause est effectuée avec l'appel de la méthode sleep(). Ensuite, nous définissons une classe Main dans laquelle nous instancions la classe MyThread et nous appelons la méthode start(). En effet, un thread Java est caractérisé par deux méthodes principales que sont la méthode run() et la méthode start(). La méthode run() permet de définir les instructions qui seront exécutées à l'intérieur du Thread. Il peut s'agir

des instructions simples comme un simple println() mais aussi des instructions plus complexes comme la lecture/écriture de fichiers. Il peut aussi s'agir des instructions plus avancées comme l'exécution d'un bout de programme susceptible d'être exécuté en autonomie du reste du programme. Quant à la méthode start(), elle permet de démarrer le thread instancié et ainsi d'exécuter les instructions définies dans la méthode run(). La méthode start() est toujours appelée après l'instanciation de la classe Thread c'est-à-dire la création de l'objet représentant le thread suite à l'appel de l'opérateur new sur la classe de Thread. Dans l'exemple ci-dessous, l'objet thread est créé et exécuté dans la classe Main.

En plus des méthodes run(), start(), sleep(), la classe Thread dispose de plusieurs autres méthodes permettant de gérer les objets threads. Pour avoir plus de détails sur la classe Thread ainsi que ses différentes méthodes, consulter la page suivante : https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.html

13.2.2 Créer un Thread en implémentant l'interface Runnable

L'interface Runnable est une interface fonctionnelle c'est-à-dire une interface ne comportant qu'une seule méthode à implémenter. Il s'agit en l'occurrence de la méthode run().

Créer un Thread en implémentant l'interface Runnable consiste d'abord à créer une classe à partir de l'interface Runnable en utilisant le mot-clé implements. Cette classe implémente la méthode run() de l'interface Runnable. Et par la suite dans le reste du programme, on peut instancier la classe Thread en lui passant comme argument une instance de la classe ayant implémentée l'interface Runnable. L'exemple ci-dessous décrit la mécanique de création d'un thread à partir de l'interface Runnable.

Implémenter l'interface Runnable

```
package com.tuto.thread;
class MyRunnable implements Runnable  {
        public void run() {
                System.out.println("Afficher un nombre chaque seconde: de 1 à
10");
                for (int i = 1; i <= 10; i++) {
                        System.out.println("Le nombre affiché est "+i);
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException
                                e) {
                        }
                }
        }
}
```

Création de l'objet Thread à partir de la classe Runnable

```
package com.tuto.thread;
public class Main {
        public static void main(String[] args)  {
                // Instanciation de la du thread
                Thread myThread=new Thread(new MyRunnable());
                // Execution du thread
```

```
                myThread.start();

        }
}
```

Output :

```
Afficher un nombre chaque seconde: de 1 à 10
Le nombre affiché est 1
Le nombre affiché est 2
Le nombre affiché est 3
Le nombre affiché est 4
Le nombre affiché est 5
Le nombre affiché est 6
Le nombre affiché est 7
Le nombre affiché est 8
Le nombre affiché est 9
Le nombre affiché est 10
```

Comme on peut le constater, dans l'exemple ci-dessus, nous définissons d'abord une classe nommée MyRunnable qui implémente l'interface Runnable et sa méthode run(). Les instructions que nous avons définies dans la méthode run() ont pour but d'afficher les nombres de 1 à 10 avec une pause de 1000 millisecondes (1 seconde) entre deux affichages. Mais il est important de noter que la classe obtenue en implémentant l'interface Runnable ne peut pas être directement instanciée pour créer un objet Thread. Elle doit être instanciée et passée en argument d'une classe Thread pour enfin obtenir l'objet Thread. C'est dans cette démarche qu'est obtenue l'objet myThread qui est, en effet, un objet Thread proprement-dit, c'est-à-dire qui dispose de la méthode start() qu'on peut appeler pour exécuter les instructions définies dans la méthode run() définie dans la classe MyRunnable.

13.3 Lancer plusieurs threads

Dans la section précédente, nous avons montré comment créer et démarrer un thread. Dans cette section, nous allons montrer comment instancier et démarrer plusieurs threads dans le même programme. A titre illustratif, nous allons lancer trois threads, chacun exécutant une instruction spécifique. Le premier thread affiche cinq fois le mot de salutation « Bonjour » avec une pause de 1000 millisecondes entre deux affichages. Le second thread affiche sept fois le mot de salutation « Bonsoir » avec une pause de 1000 millisecondes entre deux affichages. Et le troisième thread affiche dix fois la salutation « Au revoir » avec une pause de 1000 millisecondes. Voir code source ci-dessous.

Créer une classe de Thread

```
package com.tuto.thread;
class MyThread extends Thread {
        // Champs permettant de spécifier le texte de salutation: bonjour,
bonsoir, Au revoir
        String salutation;
        // Champ indiquant le nombre de fois la salutation sera affichée
        int nb_salutation;
        // Constructeur de la classe
        MyThread(String salutation, int nb_salutation) {
                this.salutation= salutation;
```

```
                this.nb_salutation=nb_salutation;
        }
        public void run() {
                for(int i = 1; i <= this.nb_salutation; i++) {
                        System.out.println(this.salutation);
                        try {
                                Thread.sleep(1000); // Pause de 1000
millisecondes après chaque affichage
                        }
                        catch(InterruptedException ie) {
                                System.out.println(ie.getMessage());
                        }
                }
        }

}
```

Instancier et lancer plusieurs Threads

```
package com.tuto.thread;
public class Main {
        public static void main(String[] args)  {

        // Instancier et démarrer le Thread bonjour
        Thread bonjourThread= new MyThread("Bonjour",5);
        // Execution du thread
        bonjourThread.start();
        // Instancier et démarrer le Thread bonsoir
        Thread bonsoirThread= new MyThread("Bonsoir",7);
        // Execution du thread
        bonsoirThread.start();
        // Instancier et démarrer le Thread au-revoir
        Thread aurevoirThread= new MyThread("Aurevoir",10);
        // Execution du thread
        aurevoirThread.start();

        }
}
```

Output :

```
Bonjour
Bonsoir
Aurevoir
Bonjour
Bonsoir
Aurevoir
Aurevoir
Bonsoir
Bonjour
Aurevoir
Bonjour
Bonsoir
Bonjour
Aurevoir
Bonsoir
Aurevoir
Bonsoir
Bonsoir
Aurevoir
Aurevoir
Aurevoir
Aurevoir
```

Dans l'exemple ci-dessus, nous définissons d'abord une classe Thread nommée MyThread qui étend la classe Thread et qui redéfinit sa méthode run(). Par ailleurs, pour que la même classe Thread puisse répondre aux trois cas d'utilisation, nous avons défini deux champs nommés salutation et nb_salutation qui permettent respectivement de spécifier le mot de salutation (Bonjour, Bonsoir ou Au revoir) et le nombre de fois que ce mot va être affiché. Nous avons donc prévu un constructeur pour la classe MyThread afin de définir la valeur de ces champs lors de la création des objets threads.

Après avoir défini la classe MyThread, nous définissons une classe Main afin de pouvoir instancier la classe MyThread et lancer les différents threads souhaités. A noter que la classe MyThread peut être importée et instanciée dans n'importe quelle autre classe du programme en dehors de la classe Main.

Comme on peut le constater, nous avons instancié trois fois la classe MyThread ; chaque instanciation étant faite avec à la fois son message de salutation et le nombre de fois que le message est affiché. En l'occurrence, nous avons créé et démarré les threads bonjourThread, bonsoirThread et aurevoirThread. Chaque thread est démarré en appelant la méthode start().

L'output produit suite à l'exécution de la classe Main permet de montrer que les threads ne sont pas synchrones, c'est-à-dire que le thread suivant n'attend pas la fin de thread précédent pour exécuter ses instructions. C'est pourquoi dans l'output, les messages Bonjour, Bonsoir et Au revoir sont mélangés dans la console. Cette situation provient du fait qu'au moment où un thread est en pause avec l'appel de la méthode sleep(), la JVM profite de ce instant pour lancer l'exécution des instructions d'un autre thread, et vice-versa.

En résumé, la création et le démarrage de plusieurs threads dans un même programme permet d'exécuter quasi-simultanément les instructions de tous les threads. Cependant par défaut, les threads restent asynchrones. Mais, à noter que Java offre plusieurs autres méthodes pour gérer les threads notamment gérer la priorité des threads, connaître l'état des threads, interrompre des threads, etc… Pour plus de détails sur la gestion des threads, consulter la page : https://docs.oracle.com/javase/10/docs/api/java/lang/Thread.html

14 TESTS UNITAIRES JAVA: UTILISATION DU FRAMEWORK JUNIT

14.1 Généralités

Les tests unitaires servent à vérifier le bon fonctionnement des composants unitaires d'un programme : classes, méthodes, etc.... Le rôle des tests unitaires est de détecter et corriger d'éventuels bugs lors du développement d'une application ou détecter des régressions lors des évolutions d'un programme déjà existant.

JUnit est un framework open source regroupant un ensemble de ressources prêts-à-l'emploi permettant d'implémenter et de dérouler de tests unitaires Java. Ce chapitre vise à présenter de façon sommaire et synthétique l'usage du framework JUnit à travers des exemples concrets.

14.2 Définition d'une classe d'illustration pour les tests unitaires JUnit

Pour illustrer la mise en œuvre des tests unitaires JUnit, nous partons d'un exemple de classe nommée Calculator qui permet de réaliser 5 opérations arithmétiques : addition, soustraction, multiplication, division et puissance. La classe Calculator disposera d'une méthode nommée calculate() qui prend en paramètre le type d'opération à réaliser et les deux nombres sur lesquels l'opération sera appliquée. Par exemple, pour additionner deux nombres x et y, on appelle la méthode calcultate() paramétrée telle que calculate("addition", x, y). La classe Calculator et sa méthode calculate() étant définies, l'objectif sera par la suite de réaliser des tests unitaires pour vérifier que la méthode calculate() effectue correctement les opérations. Le bout de code ci-dessous présente la définition de la classe Calculator ainsi que sa méthode calculate().

Définition de la classe Calculator

```
package com.tuto.tu;

public class Calculator{

  public double calculate(String typeOperation,double x, double y) throws
Exception {

      switch (typeOperation.toLowerCase()) {
          case "addition":
              System.out.println("Addition de "+ x +" et de "+y);
              return x+y;
          case "soustraction":
              System.out.println("Soustraction de "+ x +" et de "+y);
              return x-y;
          case "multiplication":
              System.out.println("Multiplication de "+ x +" et de "+y);
              return x*y;
          case "division":
              System.out.println("Division de "+ x +" et de "+y);
              return x/y;
          case "puissance":
```

```
                System.out.println("Puissance de "+ x +" et de "+y);
                return Math.pow(x, y);
            default:
                System.out.println("L'opération spécifiée n'est pas prise en
charge");
                throw new Exception("Opération non reconnue par la classe
Claculator");
        }

    }

}
```

Comme on peut le constater, la méthode calculate() réalise les opérations arithmétiques dans un bloc switch() défini suivant les valeurs du paramètre typeOperation. La classe Calculator prend en charge cinq opérations : addition, soustraction, multiplication, division et puissance. Toute autre opération spécifiée en dehors de ces cinq renvoie une exception. Le bout de code ci-dessous montre un exemple d'instanciation de la classe Calculator et d'appel de sa méthode calculate().

Instanciation de la classe Calculator dans une classe Main

```
package com.tuto.tu;

public class Main {
        public static void main(String[] args) throws Exception {

                // Instanciation de la classe Calculator
                Calculator calculator =new Calculator();
                // Addition de 7 et 5
                double resAddition=calculator.calculate("addition",7,5);
                System.out.println("resAddition: "+resAddition);
                // Soustraction de 7 et 5
                double
resSoustraction=calculator.calculate("soustraction",7,5);
                System.out.println("resSoustraction "+resSoustraction);
                // multiplication de 7 et 5
                double
resMultiplication=calculator.calculate("multiplication",7,5);
                System.out.println("resMultiplication "+resMultiplication);
                // division de 7 et 5
                double resDivision=calculator.calculate("division",7,5);
                System.out.println("resDivision "+resDivision);
                // puissance de 7 et 5
                double resPuissance=calculator.calculate("puissance",7,5);
                System.out.println("resPuissance "+resPuissance);
                // Opération non reconnue par la classe Claculator. ex: modul0
                double resModulo=calculator.calculate("modulo",7,5);
                System.out.println("resModulo "+resModulo);

        }
}
```

Output

```
Addition de 7.0 et de 5.0
resAddition: 12.0
Soustraction de 7.0 et de 5.0
resSoustraction 2.0
Multiplication de 7.0 et de 5.0
resMultiplication 35.0
Division de 7.0 et de 5.0
resDivision 1.4
Puissance de 7.0 et de 5.0
resPuissance 16807.0
L'opération spécifiée n'est pas prise en charge
Exception in thread "main" java.lang.Exception: Opération non reconnue par la
classe Claculator
at com.tuto.tu.Calculator.calculate(Calculator.java:25)
at com.tuto.tu.Main.main(Main.java:30)

Process finished with exit code 1
```

Dans le code ci-dessus, nous avons défini une classe Main. Et dans la méthode main() de cette classe, nous avons instancié la classe Calculator en créant un objet nommé calculator. Ensuite, nous appelons la méthode calculate() pour réaliser différentes opérations arithmétiques et afficher leur résultat. Comme on peut le constater, toutes les opérations prédéfinies semblent se dérouler correctement à l'exception de la dernière qui est, en effet, une opération non prise en charge.

Rappelons que la mise en place de la classe Calculator vise à disposer d'une classe pouvant servir de cas d'illustration pour la mise en œuvre des tests unitaires JUnit. Ainsi, dans le reste de ce chapitre, les exemples d'illustrations se baseront sur la classe Calculator, en particulier, sur sa méthode calculate() pour illustrer différents aspects pratiques des tests Junit.

14.3 Installation de la librairie junit

Pour pouvoir utiliser le framework JUnit, il faut d'abord télécharger et installer la dépendance externe junit qui est une librairie contenant l'ensemble des utilitaires nécessaires à la mise en place de tests unitaires. A noter que la librairie junit a aussi sa propre dépendance. Il s'agit de la libraire hamcrest. Ainsi pour pouvoir dérouler les tests unitaires JUnit, il faut installer à la fois la librairie junit et la librairie hamscrest. Dans ce présent document, nous utilisons les fichiers jar junit-4.13.2.jar et hamcrest-2.2.jar. Tous les deux fichiers sont disponibles sur la plupart des repositories de gestion de dépendances : Maven, Gradle, Ivy, SBT, etc. Ici, nous téléchargeons les fichiers depuis le repository Maven. Nous reviendrons plus tard sur l'utilisation de Maven pour la gestion des dépendances dans un projet Java dans le chapitre 15 consacré à la gestion des dépendances par l'utilisation de l'outil Maven[37].

[37] Pour ajouter la dépendance junit et hamcrest dans un projet Maven, ajouter ces spécifications dans le fichier pom.xml
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.13.2</version>

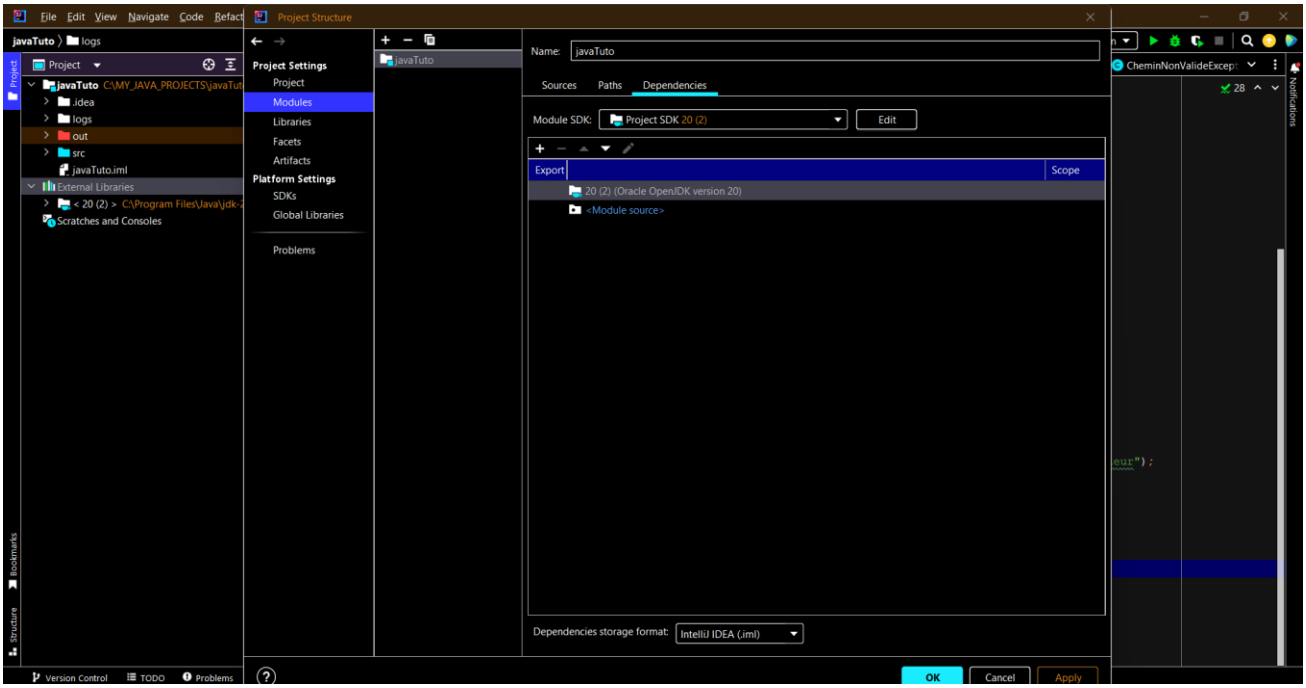
Pour installer et charger manuellement les librairies junit et hamcrest , suivre les étapes suivantes :

- Télécharger les fichiers
  https://repo1.maven.org/maven2/junit/junit/4.13.2/junit-4.13.2.jar

  https://repo1.maven.org/maven2/org/hamcrest/hamcrest/2.2/hamcrest-2.2.jar
- Déposer ces fichiers dans le dossier lib de votre installation Java ou de votre JDK. Pour notre cas, nous utilisons : C:\Program Files\Java\jdk-20\lib.

Nous allons ajouter ce fichier dans le classpath de notre programme Java. Cet ajout peut être directement fait à partir des menus de votre IDE. Par exemple, pour Intellij IDEA, l'ajout des fichiers jar se fait comme suit :

1. Dans la barre des menus, cliquer sur File >Project Structure >Project Settings> Modules > Dependencies. Voir capture d'écran ci-dessous.

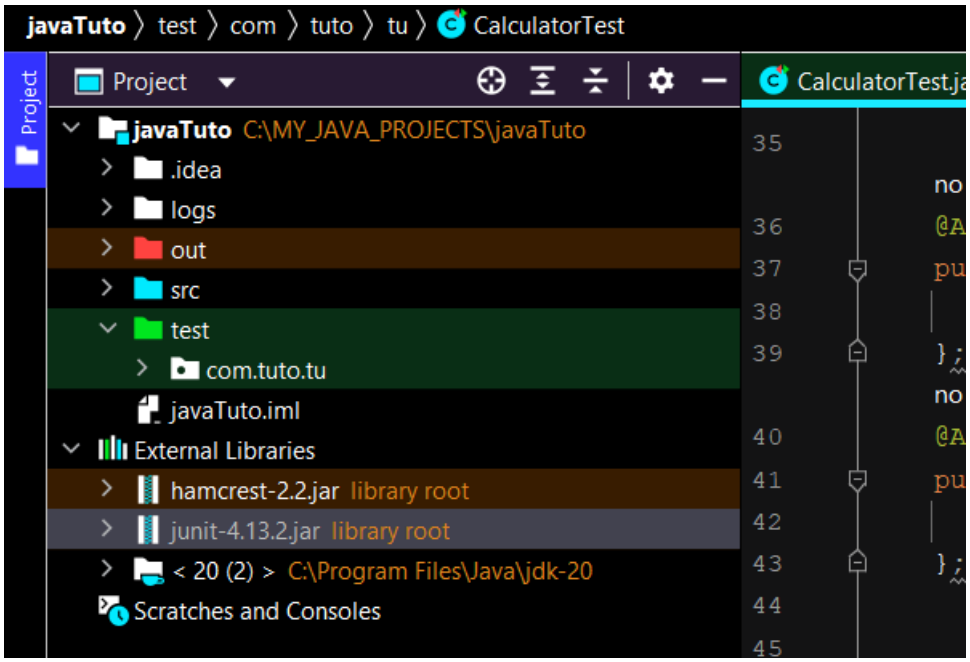


2. Cliquer sur le signe + (Add). Et choisir JARs or Directories. Et aller chercher les fichiers junit-4.13.2.jar et hamcrest-2.2.jar depuis le dossier lib de votre JDK.

```
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest</artifactId>
  <version>2.2</version>
  <scope>test</scope>
</dependency>
```

3. Cliquer sur ok pour sélectionner et cocher la case pour pouvoir ajouter les fichiers aux dépendances. Devant chaque librairie sélectionnée, dans scope, dans la liste déroulante choisir Compile. Cliquer sur Ok pour valider.
4. Cliquer sur ok pour valider.

A la suite de cette procédure d'installation, les fichiers jar ainsi que leur contenu sont visibles à gauche dans External librairies. Ce qui montre que les dépendances junit et hamscrest sont désormais bien installées. Voir la capture d'écran ci-dessous.



14.4 Structure d'une classe de test JUnit

La classe de test représente l'unité principale d'exécution des tests unitaires Junit. Tout comme une classe Main (par l'intermédiaire de la méthode main()) est nécessaire pour exécuter le programme principal, une classe de test est nécessaire pour exécuter les tests unitaires Junit. Mais à la différence de la classe Main, une classe de test ne contient pas de méthode main() à proprement parler. Le rôle de chaque méthode est défini via l'usage des annotations. Cette section vise à présenter la structure générique d'une classe de test.

D'abord, faisons remarquer d'entrée que les classes de test ainsi que l'ensemble des ressources de tests sont positionnés par défaut dans un dossier source dédié nommé ./test. Le dossier . /test est situé à la racine du projet Java à l'image des codes sources du programme principal situés dans le dossier ./src. Mais comme nous allons le voir plus tard dans les chapitres suivants notamment l'utilisation des outils de gestion de dépendances comme Maven, le dossier ./src peut être réorganisé pour contenir à la fois les codes sources

et les codes de test. Par exemple dans un projet Maven, les codes sources du programme sont situés dans l'arborescence ./src/main/java. Tandis que les codes de test unitaires sont situés dans l'arborescence ./src/test/java. A ce stade du tutoriel, nous avons positionné les codes sources et les codes de test unitaires dans deux dossiers distincts : ./src (pour les codes sources) et ./test (pour les codes de tests). Cette distinction nécessite, toutefois une petite configuration au niveau de l'IDE pour que le rôle de chaque dossier puisse être explicitement reconnu. Par exemple, dans l'IDE Intellij, pour que le dossier ./test puisque être reconnu comme le dossier source des test, il faut faire les manipulations suivantes. Clique-droit sur le dossier test. Dans le menu de contexte, on choisit « Mark Directory as » et on choisit « Test Source root ». Et s'il s'agissait de l'IDE Eclipse, la démarche serait la suivante. Cliquer-droit sur le nom de votre projet, choisir « Properties », choisir « Java Build Path », cliquer sur l'onglet « Source », cliquer sur « Add folder » et choisir le dossier test. Cliquer sur « Apply and Close ».

Après la configuration du dossier ./test comme répertoire de codes source pour les tests, nous pouvons maintenant écrire la classe de test. Le squelette de code ci-dessous présente la structure standard d'une classe principale de test.

Structure générale d'une classe de test

```
package com.tuto.tu;

import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class MyClassTest{

    @BeforeClass
    public static void setUpClass() throws Exception {
        // Méthode qui exécuter avant toute autre instruction
    }
    @Before
    public void setUp() throws Exception {
        // Méthode qui sera executée avant chaque méthode de test
    }
    @Test
    public void testMethod1() {
        // Méthode de test 1
    }

    @Test
    public void testMethod2() {
        // Méthode de test 1
    }

    @After
    public void tearDown() throws Exception {
        // Méthode qui sera executée après chaque méthode de test
    }
    @AfterClass
```

```
    public static void tearDownClass() throws Exception {
        // Méthode à exécuter après tous les tests
    }

}
```

Comme le montre ce squelette de code, la définition d'une classe de test commence toujours par l'import des dépendances JUnit. Les classes qui sont habituellement importées sont : Test, les classes d'assertion du package Assert, les classes BeforeClass, Before, After et AfterClass. Chacune de ces classes joue un rôle spécifique dans la mise en place et le déroulement du test unitaire. Nous détaillerons dans la foulée le rôle de chaque classe. Mais d'abord, concernant le nom de la classe de test, celui-ci doit être choisi de sorte à faire apparaître le mot « Test » en suffixe. Même si cette règle de nommage n'est pas une obligation, il facilite néanmoins la reconnaissance d'une classe de test par rapport à une classe de traitement standard. Dans ce squelette, la classe de test est nommée MyClassTest. Habituellement une classe de test sert à regrouper l'ensemble des tests à réaliser sur les méthodes d'une classe donnée. C'est d'ailleurs pour cette raison qu'une classe de test est habituellement créée pour chaque classe à tester. Par exemple si l'on dispose deux classes MyClass1 et MyClass2, pour tester séparément les méthodes de ces deux classes, on crée deux classes correspondant respectivement à MyClass1Test et MyClass2Test. Mais techniquement, rien n'empêche de tester les méthodes des deux classes dans la même classe de test MyClassesTest.

Maintenant, pour ce qui concerne les classes importées Test, Before, BeforeClass, After et AfterClass, ces classes servent plutôt à annoter les méthodes définies dans la classe de test de sorte à distinguer le rôle de chaque méthode lors du déroulement du test. Ci-après la description du rôle de chaque annotation.

- **@Test :** lorsque l'annotation @Test accompagne une méthode dans la classe de test, cela permet d'indiquer à JUnit que cette méthode contient les tests unitaires à exécuter. Il faut noter que dans la classe de test, au moins une méthode doit être accompagnée par l'annotation @Test pour que la classe puisse être exécutée. L'annotation @Test sert ainsi de déclencheur des tests unitaires tout comme la méthode main() sert de déclencheur pour le programme principal.

  A noter qu'un test vise toujours à vérifier une « assertion », c'est-à-dire une proposition affirmative ou négative prise à priori comme vraie. Et le résultat du test est toujours une valeur booléenne qui est égale à true lorsque la proposition est vérifiée et false lorsque la proposition n'est pas vérifiée. Lorsque la valeur renvoyée par le test est true on dit que le test passe (success). Et lorsque la valeur renvoyée par le test est false, on dit que le test ne passe pas (fail). Pour vérifier les assertions, le module Junit prévoit déjà plusieurs classes de test disponibles dans le package Assert. Les classes les plus couramment utilisées sont entre autres :

  - **assertEquals()** : qui permet de vérifier si le résultat renvoyé par la méthode à tester est égale à une valeur spécifiée par l'utilisateur (égalité des deux valeurs) ;

- **assertTrue()** : qui permet de vérifier que le résultat renvoyé par une instruction est effectivement true ;
- **assertFalse()** : qui permet de vérifier que le résultat renvoyé par une instruction de test est effectivement false
- **assertNull()** : qui permet de vérifier que la valeur renvoyée par une instruction de test est effectivement nulle
- **assertNotNull()** : qui permet de vérifier que la valeur renvoyée par une instruction de test est effectivement non nulle
- …

@BeforeClass et @Before : lorsque l'annotation @BeforeClass accompagne une méthode dans la classe de test, cela permet d'indiquer à JUnit que cette méthode doit être exécutée en début de la classe de test avant toute autre instruction dans la classe. Et lorsque l'annotation @Before accompagne une méthode dans la classe de test, cela permet d'indiquer à Junit que cette méthode doit être exécutée avant l'appel de chaque méthode de test, c'est-à-dire chaque méthode accompagnée de l'annotation @Test. Pendant que les méthodes annotées avec @Before peuvent être lancées plusieurs fois dans la même session de test, les méthodes @BeforeClass ne sont exécutées qu'une seule fois.

Les méthodes annotées avec @BeforeClass et @Before jouent des rôles différents dans une classe de test. En l'occurrence les méthodes @BeforeClass sont utilisées pour préparer l'environnement de test : initialisation de session, création et démarrage de connections aux bases de données, appel de constructeurs de la classe à tester, etc… Et les méthodes annotées avec @Before sont utilisées pour réinitialiser des variables de test avant le lancement d'une autre méthode de test. A noter que l'usage des méthodes @Before et @BeforeClass n'est pas obligatoire dans une classe de test. Par exemples, toutes les instructions concourant au test peuvent être directement spécifiées dans une seule méthode accompagnée de l'annotation @Test. Mais il n'en demeure pas moins que l'usage des méthodes @Before et @BeforeClass reste une bonne pratique qui améliore la lisibilité du code.

Lorsque qu'elles sont définies, les méthodes estampillées @Before et @BeforeClass sont toujours exécutées même si les tests définis dans les méthodes de test ne passent pas.

- **@After et @AfterClass** : Les annotations @After et @AfterClass sont définies à l'image des annotations @Before et @BeforeClass. Lorsque l'annotation @After accompagne une méthode de la classe de test, cela permet d'indiquer à JUnit que cette méthode doit être exécutée après l'exécution de chaque méthode de test, c'est-à-dire chaque méthode accompagnée par l'annotation @Test. Quant à l'annotation @AfterClass, elle permet d'indiquer à JUnit que la méthode qu'elle accompagne doit être exécutée à la fin de toutes les méthodes de test, c'est-à-dire à la fin de l'exécution complète des tests. Une méthode avec @After peut être exécutée plusieurs fois dans

une même session de test, notamment lorsque plusieurs méthodes de test sont définies. En revanche les méthodes annotées avec @AfterClass ne sont exécutées qu'une seule fois dans la session de test. A l'image des annotations @Before et @BeforeClass, les annotations @After et @AfterClass n'ont pas le même usage. L'annotation @After peut être utilisée pour définir des méthodes visant à réinitialiser une variable, à nettoyer l'environnement de test avant le lancement d'une autre méthode de test. Quant à l'annotation @AfterClass, elle peut être utilisée pour définir des méthodes qui nettoient l'environnement et ferment la session de test : fermeture des connections aux bases de données, etc… Tout comme les annotations @Before et @BeforeClass, l'usage des annotations @After et @AfterClass n'est pas obligatoire dans une classe de test. Cependant leur usage constitue une bonne pratique qui améliore la lisibilité du code des tests unitaires.

Notons que lorsqu'elles sont définies, les méthodes estampillées @After et @AfterClass sont toujours exécutées même si les tests définies dans les méthodes de test renvoent le statut « failed ».

14.5 Exemple pratique de définition d'une classe de test

Pour mettre en pratique le test unitaire JUnit, nous partons de la classe d'illustration nommée Calculator que nous avons déjà définie dans les sections précédentes (voir la section 14.2 dédiée à la présentation de la classe d'illustration de tests unitaires). La classe Calculator est une classe qui dispose d'une méthode nommée calculate() permettant de réaliser cinq opérations arithmétiques distinctes : addition, soustraction, multiplication, division et puissance. Pour réaliser une opération parmi celles indiquées, il suffit de spécifier en paramètre de la méthode calculate() le type d'opération ainsi que les deux nombres sur lesquels portent l'opération. Voir quelques exemples d'appel de la méthode calculate() dans la section consacrée à la présentation de la classe d'illustration.

L'objet de cette présente section est de vérifier à travers des tests JUnit que la méthode calculate() réalise correctement les opérations prévues et que les résultats qu'elle produit sont conformes aux attentes.

Le code ci-dessous illustre quelques cas de test de la méthode calcultate() de la classe Calculator.

Exemple typique d'un test untaire Junit

```
package com.tuto.tu;

import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.AfterClass;

import com.tuto.tu.Calculator;

public class CalculatorTest {
```

```
    static Calculator calculator;

    @BeforeClass
    public static void setUpClass() throws Exception {
        System.out.println("Instructions à exécuter avant toute autre
instruction de test");
        // Instancier la classe Calculator et définir le champs de classe
calculator
        calculator = new Calculator();
    };
    @Before
    public void setUp() throws Exception {
        System.out.println("Instructions à exécuter avant le lancement de
chaque méthode de test");
    };

    @Test
    public void testOperation() throws Exception {
        // Tester l'opération addition
        double resAddition=calculator.calculate("addition",7,5);
        assertEquals(12, (int)resAddition);
        //Tester l'opération multiplication
        double resMultiplication=calculator.calculate("multiplication",7,5);
        assertEquals(35, (int)resMultiplication);

        System.out.println("Tous les tests passent");

    };

    @After
    public void tearDown() throws Exception {
        System.out.println("Instructions à exécuter après le lancement de
chaque méthode de test");
    };
    @AfterClass
    public static void tearDownClass() throws Exception {
        System.out.println("Instructions à exécuter à la fin complète des
test");
    };


}
```

Output :

```
Instructions à exécuter avant toute autre instruction de test
Instructions à exécuter avant le lancement de chaque méthode de test
Addition de 7.0 et de 5.0
Multiplication de 7.0 et de 5.0
Tous les tests passent
Instructions à exécuter après le lancement de chaque méthode de test
Instructions à exécuter à la fin complète des test

Process finished with exit code 0
```

La classe CalculatorTest définie ci-dessus est une classe typique de test faisant figurer tous les types de méthodes que nous avons précédemment discutées suivant le type d'annotation

utilisé. En effet dans la classe CalculatorTest, nous avons défini une méthode annotée avec @BeforeClass, une méthode annotée avec @Before, une méthode annotée avec @Test, une annotée avec @After et enfin une annotée avec @AfterClass. Encore une fois, il s'agit ici d'un exemple typique présenté à titre pédagogique, car en réalité c'est seulement la méthode annotée @Test qui est obligatoire dans une classe de test. Toutes les autres annotations restent optionnelles dans la mesure où les instructions qu'elles définissent peuvent être directement spécifiées à l'intérieur de la méthode annotée avec @Test. Cependant pour des raisons de lisibilité de code et de clarté de présentation, nous avons distinctement présenté chaque type de méthode même si les instructions qui les caractérisent restent des simples affichages de type println().

Dans l'exemple présenté, comme on peut le constater, nous commençons d'abord par déclarer un objet de type Calculator à travers le champ calculator. Il s'agit ici d'un champ de type static car nous comptons initialiser sa valeur dans une méthode de type @BeforeClass destinée à la préparation de l'environnement et les ressources de test : instanciation des objets, création de connections, etc.. Ici la méthode qui sert à préparer l'environnement et les ressources de test est nommée setUpClass(). Cette méthode est annotée avec @BeforeClass. A noter que les méthodes annotées avec @BeforeClass (et par ricochet @AfterClass) sont toujours de type static. Elles permettent d'initialiser les champs et attributs de type static. Dans l'exemple, la méthode setUpClass() permet d'initialiser la valeur du champ calculator. Ce champ représente, en effet, l'objet dont la méthode calculate() sera appelée pour réaliser le test unitaire.

Les méthodes setUp() et tearDown() sont respectivement des méthodes de type @Before et @After qui s'exécutent avant et après l'exécution de chaque méthode estampillée @Test. Ces méthodes s'exécutent même si la méthode de test renvoie une AssertionError c'est-à-dire un test finissant avec le statut failed.

Quant à la méthode tearDownClass(), elle joue théoriquement le rôle de la méthode qui nettoie l'environnement de test et met fin à la session de test. Encore une fois, étant donné que l'exemple que nous avons présenté est un cas pédagogique, la méthode tearDownClass() se limite ici simplement à afficher une ligne de message à travers l'usage de la méthode println().

S'agissant maintenant de la méthode testOperation(), elle représente la seule méthode de test définie dans cette classe de test. Etant donnée qu'elle est accompagnée de l'annotation @Test, elle doit contenir les instructions qui exécutent les tests proprement-dits. Pour cela, elle utilise, le cas échéant, les ressources préparées par les méthodes estampillées @Before et @BeforeClass. Dans l'exemple ci-dessus, nous réalisons deux tests : l'un à la suite d'une opération d'addition et l'autre à la suite d'une opération de multiplication. S'agissant du test sur l'opération d'addition, nous appelons d'abord la méthode calculate() de l'objet calculator et nous effectuons l'addition des nombres 7 et 5. Ensuite, nous vérifions que le résultat de cette opération est bien égal à 12. Pour cela, nous utilisons la classe de vérification d'assertion : assertEquals().

L'exécution de la classe CalculatorTest telle que définie ci-haut ne renvoie aucune erreur et exception. Ce qui signifie que tous les tests passent.

Mais, tentons par exemple d'exécuter la classe CalculatorTest en remplaçant la ligne assertEquals(12, (int)resAddition) par assertEquals(15,(int)resAddition). Cette exécution renverra alors l'output suivant :

```
Instructions à exécuter avant toute autre instruction de test
Instructions à exécuter avant le lancement de chaque méthode de test
Addition de 7.0 et de 5.0
Instructions à exécuter après le lancement de chaque méthode de test

java.lang.AssertionError:
Expected :15
Actual   :12
Instructions à exécuter à la fin complète des test
```

On obtient une exception de type java.lang.AssertionError qui signifie que le test spécifié ne passe pas. Les raisons de l'échec du test sont également fournies à travers les mots Expected et Actual. *Expected* est la valeur de comparaison spécifiée par l'utilisateur tandis que *Actual* est le résultat renvoyé par la méthode appelée. Lorsque la valeur *Expected* et la valeur *Actual* sont différentes, la classe assertEquals() renvoie une AssertionError pour indiquer que le test a échoué.

Appel d'autres classes d'assertion

En plus de la classe d'assertion assertEquals(), JUnit offre d'autres classes d'assertion pour vérifier si un test passe ou non. Il s'agit notamment des classes, assertNotEquals(), assertTrue(), assertFalse(), assertNull(), assertNotNull(), etc. Le code ci-dessous illustre l'utilisation de chacun des classes dans un scenario de cas passant.

```
package com.tuto.tu;

import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.AfterClass;

import com.tuto.tu.Calculator;

public class CalculatorTest {
    @Test
    public void testOperation() throws Exception {
        // Instancier la classe à tester
        Calculator calculator= new Calculator();
        // Tester l'opération addition
        double resAddition=calculator.calculate("addition",7,5);
        // Usage de assertNotEquals()
        assertNotEquals(15,(int)resAddition);
        // Usage de assertTrue()
        assertTrue((int)resAddition==12);
        // Usage asserFalse
        assertFalse((int)resAddition==15);
        // Usage asserNotNull
        assertNotNull(resAddition);
```

```
    };

}
```

Output

```
Addition de 7.0 et de 5.0
```

Dans cet exemple, la classe CalculatorTest est une version simplifiée de la première définition où nous avons décidé de nous focaliser sur une méthode annotée @Test, ignorant pour le coup tous les autres types de méthodes, en l'occurrence des annotations de type @Before* et @After* qui, on le rappelle, restent optionnelles dans une classe de test.

S'agissant des scénarios de test, nous avons utilisé les classes d'assertion assertNotEquals(), assertTrue(), assertFalse() et assertNotNull(). Chacune de ces classes est utilisée pour tester un cas passant. Mais on pouvait aussi utiliser ces méthodes pour tester des cas non passants. Mais cela reviendrait à générer des assertErrors. L'exemple ci-dessous illustre quelques cas qui génèrent des assertErrors.

```
package com.tuto.tu;

import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.AfterClass;

import com.tuto.tu.Calculator;

public class CalculatorTest {
    @Test
    public void testOperation() throws Exception {
        // Instancier la classe à tester
        Calculator calculator= new Calculator();
        // Tester l'opération addition
        double resAddition=calculator.calculate("addition",7,5);
        // Usage de assertNotEquals()
        assertNotEquals(12,(int)resAddition); // Cas non passant
        // Usage de assertTrue()
        assertTrue((int)resAddition==15);// Cas non passant
        // Usage asserFalse
        assertFalse((int)resAddition==12); // Cas non passant
        // Usage asserNotNull
        assertNull(resAddition); // Cas non passant

    };



}
```

Output

```
Addition de 7.0 et de 5.0
java.lang.AssertionError: Values should be different. Actual: 12
```

Dans cet exemple, tous les scénarios d'assertion définis sont non passants. Ils renvoient donc tous des AssertionErrors. Cependant, en lançant l'exécution de ce code, puisque la première assertion renvoie une AssertionError, l'exécution du code s'arrête et les autres tests d'assertion ne sont pas atteints.

15 GESTION DES DEPENDANCES EXTERNES DANS UN PROJET JAVA: UTILISATION DE L'OUTIL MAVEN

15.1 Généralités sur l'usage des dépendances externes dans un projet Java

La librairie native java.lang fournit un ensemble d'interfaces, de classes, d'enums et d'annotations permettant de traduire la plupart des instructions standards d'un programme Java. Cependant, dans de nombreuses situations, le package natif n'offre pas toutes les fonctionnalités nécessaires pour développer des programmes plus complexes ou ayant des spécificités particulières. Il faut parfois faire appel à des librairies externes. Typiquement, c'est le cas du logging avec le framework Log4J2, des tests unitaires avec le framework Junit, etc… Ces librairies sont développées hors du package natif par des parties tierces et mises à la disposition des utilisateurs. Elles fournissent un ensemble de fonctionnalités qui peuvent être importées et utilisées dans n'importe quel programme Java. Toutefois, pour pouvoir être utilisées, ces librairies doivent être d'abord installées dans le classpath du programme à développer. Le classpath désigne l'ensemble des répertoires et fichiers qui sont directement accessibles depuis le programme.

Pour utiliser une librairie externe dans programme Java simple, on procède habituellement par une installation manuelle. L'installation manuelle consiste à télécharger manuellement la librairie externe depuis un repository de gestion de dépendances externes (ex: https://mvnrepository.com/) et à l'installer dans le classpath du programme (par exemple dans le dossier lib du JDK ou du JRE). Ensuite, ajouter le fichier en tant que dépendance externe à votre projet Java[38]. Mais dans des projets plus complexes, l'installation manuelle est très fastidieuse et dévient très vite ingérable. Heureusement, il existe de nombreux outils permettant d'automatiser le téléchargement et l'installation des librairies externes. Les outils les plus couramment utilisés sont Maven, Gradle, Ivy, etc.. Dans ce chapitre, nous allons présenter le cas de Maven qui reste encore parmi les outils de gestion de dépendances externes les plus populaires dans la gestion de projet Java.

15.2 Création d'un projet Java avec la structure Maven

Pour pouvoir gérer les dépendances externes avec l'outil Maven, le projet Java doit d'abord être créé dans une structure spéciale appelée projet Maven. Un projet Maven est une structure dans laquelle les dépendances externes ainsi que les informations de build et de packaging sont spécifiées dans un fichier de configuration nommé pom.xml (POM qui est l'abrégé de Project Object Model). L'usage du fichier pom.xml permet non seulement d'automatiser l'installation des dépendances externes mais également de faciliter le buid et

[38] Pour voir un exemple d'application de cette approche d'installation de la librairie externe, voir la section 11.7.1 dédiée à l'installation de la librairie Log4J2. Ou encore, voir la section 14.3 pour l'installation de la librairie Junit.

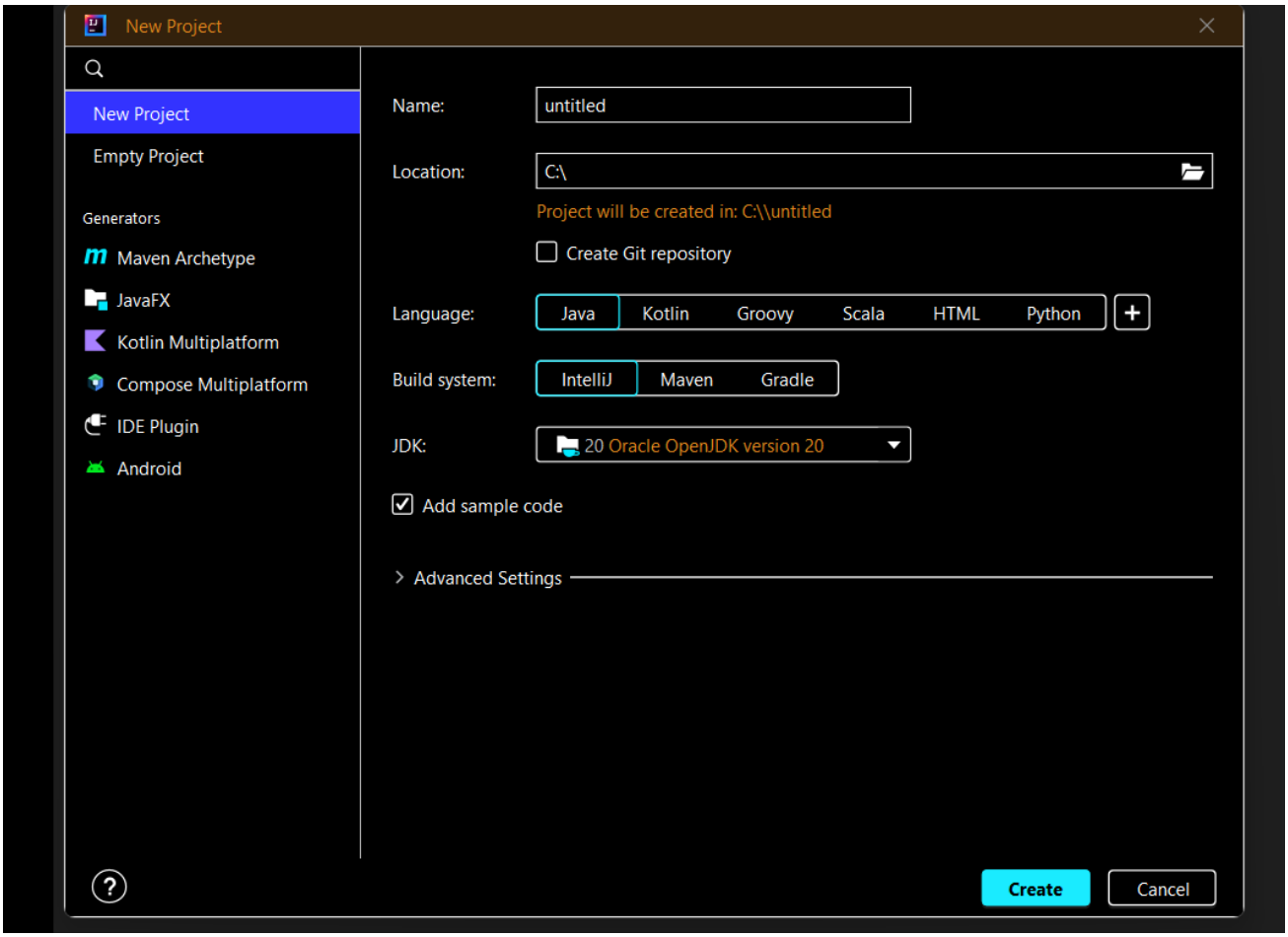
le packaging du projet Java. Le package[39] se présente alors sous forme d'un fichier archive jar ou war pouvant être transporté avec l'ensemble de ses dépendances et exécuté en autonomie dans n'importe quel environnement Java.

Dans cette section, nous présentons la procédure de création d'un projet Java dans une structure Maven ainsi que les configurations de base nécessaires pour charger automatiquement les dépendances et packager le projet Java. Etant donné que la création d'une structure Maven se fait généralement via un IDE, nous présenterons la procédure aussi bien pour le cas de Intellij, de Eclipse et de Netbeans.

15.2.1 Création d'un projet Maven avec Intellij IDEA

Les étapes ci-dessous décrivent la création d'un projet Java nommé javaTuto dans une structure Maven en utilisant l'IDE Intellij.

1. Lancer l'IDE Intellij,
2. Dans la barre des menus, cliquer File>New>Project. On obtient la fenêtre suivante:

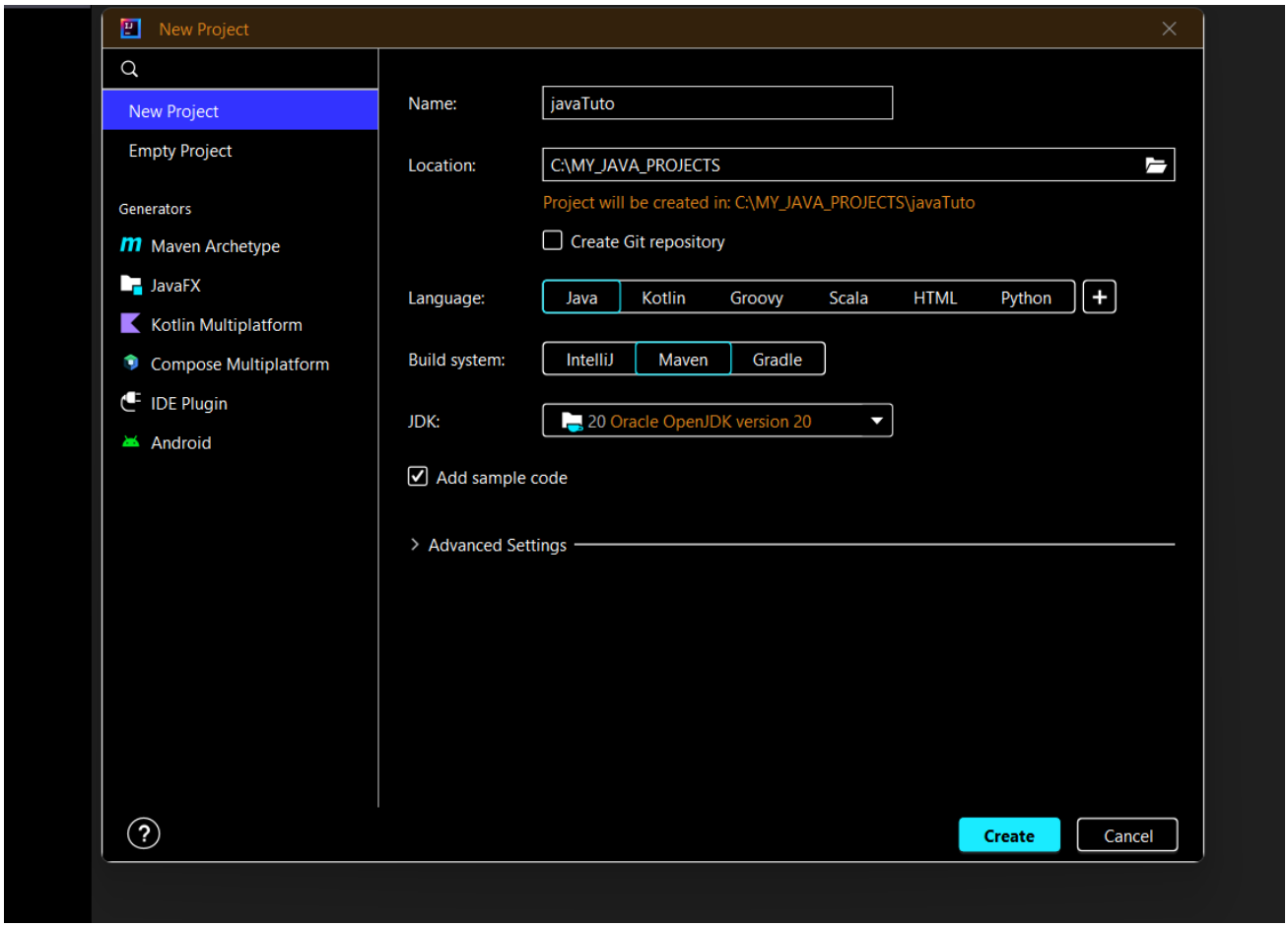


- Dans le champ Name, taper le nom du projet Java: ici javaTuto

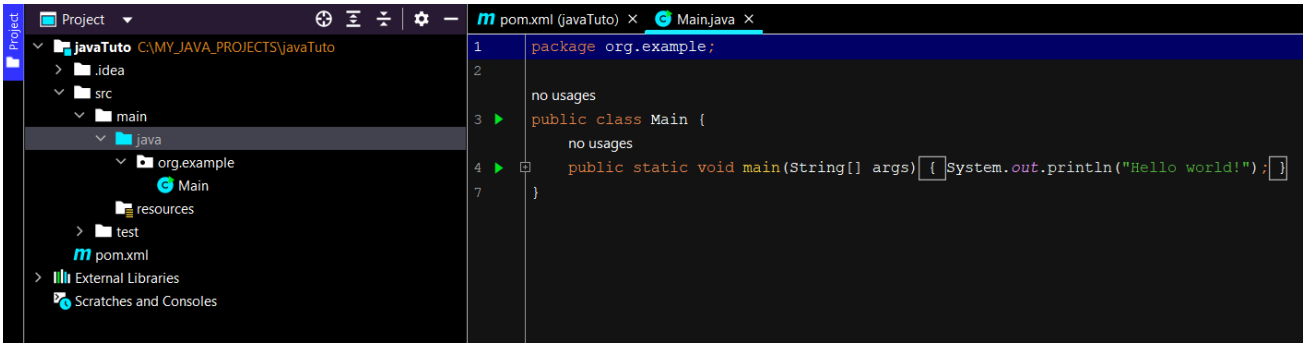
[39] Le terme package utilisé dans ce contexte ne doit pas être confondu avec le terme package utilisé pour désigner le répertoire dans lequel un élément Java est défini : classe, Enum, Interface, etc... Dans ce contexte, le package correspond à une arborescence physique définie sur le FileSystem.

- Dans le champ Location, indiquer votre workspace, c'est à dire un répertoire central où sont développés tous vos projets. Ici, on choisit C:\MY_JAVA_PROJECTS
- Dans Build System, choisir l'option Maven
- Dans JDK, choisir le JDK correspondant votre installation Java. Pour savoir comment installer un JDK, revoir les étapes déjà décrites dans la section 2.1.
- Laisser cocher l'option Add sample code pour que Intellij puisse ajouter une classe Main template qui pourra vous servir de base à écrire votre classe Main. Bien entendu, vous pouvez remplacer cette classe template par votre propre classe lors du développement de votre programme.

Après toutes ces configurations, la fenêtre se présente comme suit:



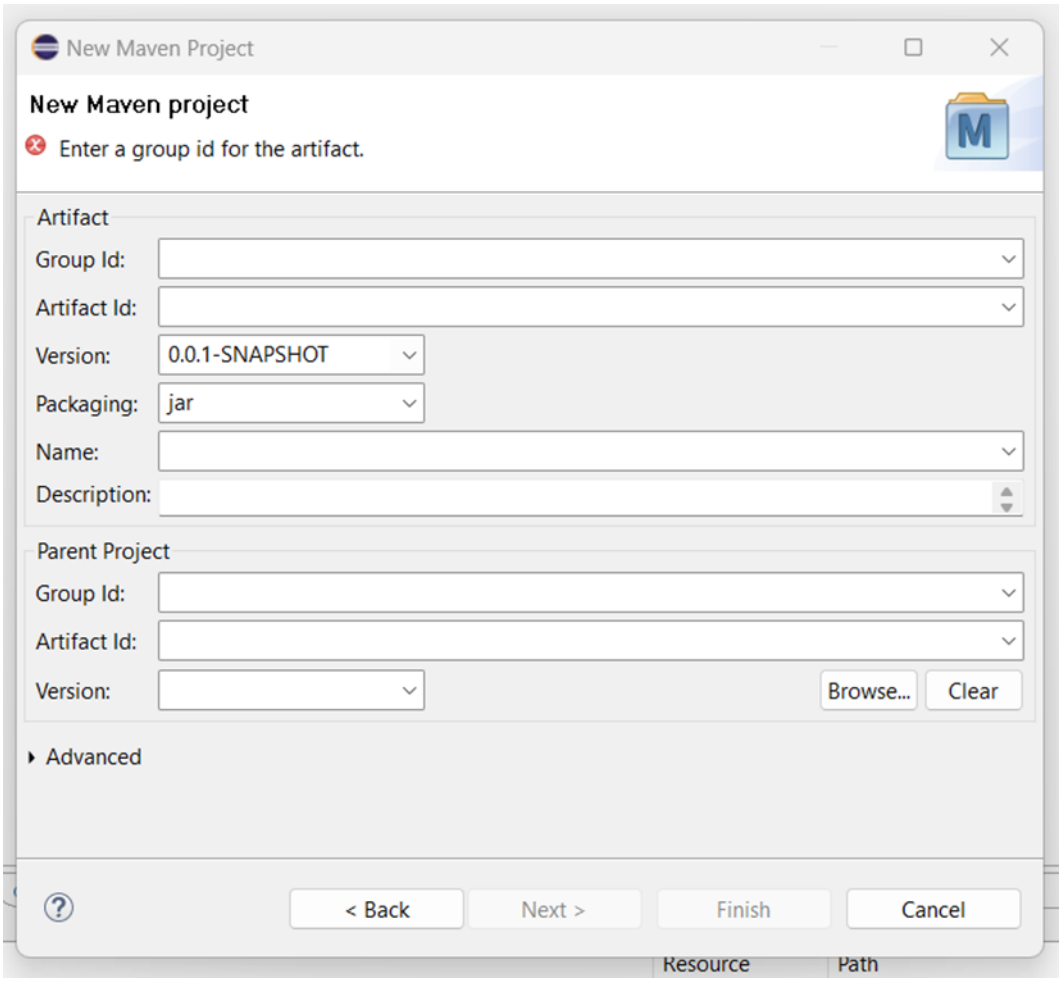
3. Cliquer sur create pour créer le projet Maven avec la configuration minimum du fichier pom.xml et l'ajout d'une classe template



15.2.2 Création d'un projet Maven avec Eclipse

Ci-dessous les étapes de création d'un projet Java nommé javaTuto dans une structure Maven en utilisant l'IDE Eclipse.

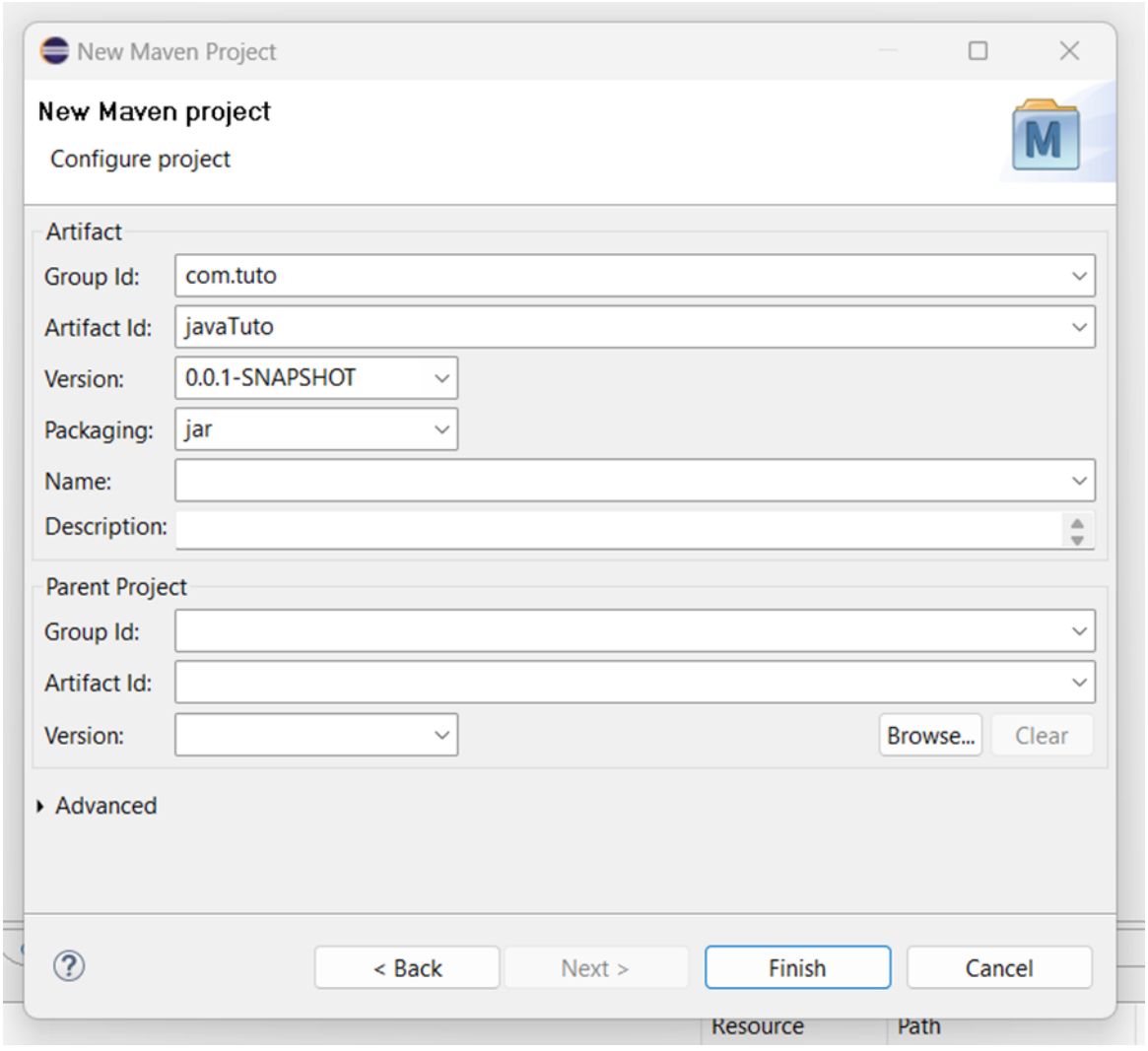
1. Lancer l'IDE Eclipse,
2. Dans la barre des menus, cliquer File>New>Project>Maven>Maven Project. Cliquer sur Next. Cocher l'option Create a simple projet (skip archetype selection). Laisser cocher l'option Use default Workspace location. Cliquer sur Next. On obtient la fenêtre suivante



- Dans le champ Group Id, indiquer le nom du package dans lequel les codes sources de votre projet seront positionnées. Ici, nous spécifions la valeur com.tuto
- Dans le champ Artifact Id, entrer le nom de votre projet Java. Ici il s'agit de javaTuto
- Dans le champ Version, laisser la valeur par défaut qui est 0.0.1-SNAPSHOT. Ce champ permet d'indiquer le numéro de version correspondant au prochain build et packaging de votre programme. Nous reviendrons plus tard sur le build et le packaging du programme Java via l'outil Maven.
- Dans le champ Packaging, garder la valeur jar. Car l'archive qui sera générée à l'issu de l'étape build et packaging sera un fichier jar. A noter que Maven permet de générer d'autres formats de packaging notamment war et pom. Le type war est dédié au packaging des codes de web services, et le type pom permet de livrer une spécification de la structure du projet.

Eclipse offre aussi la possibilité de renseigner les informations sur le projet parent à travers une section Parent Project. Renseigner cette section est utile dans le cas des projets multi-modules. Mais dans le cas d'un projet mono-module, on se limite simplement à renseigner les champs relatifs au seul module à créer.

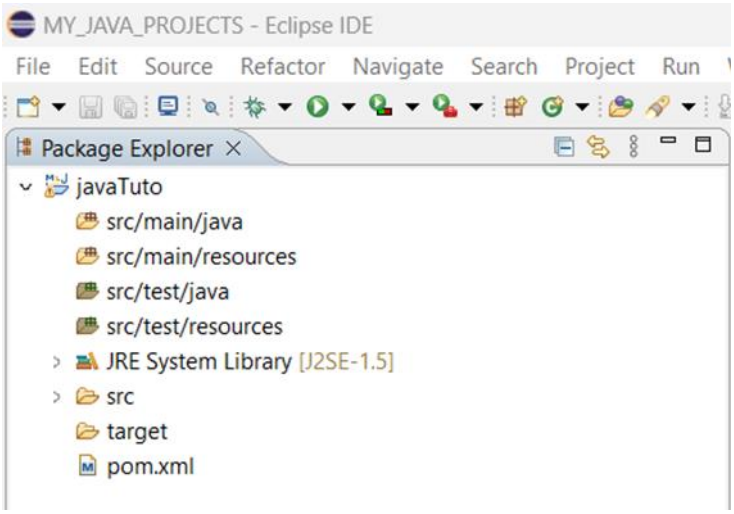
Après voir renseigné tous ces champs, la fenêtre se présente comme suit:



3. Cliquer sur Finish pour créer le projet.

   En cliquant sur Finish, le projet Maven est créé avec la configuration minimale du fichier pom.xml. Mais contrairement à Intellij, aucune classe Main template n'est initialisée. Mais la structure du dossier src reste identique entre Intellij et Eclipse. Nous reviendrons plus tard sur la structuration des dossiers dans un projet Maven.

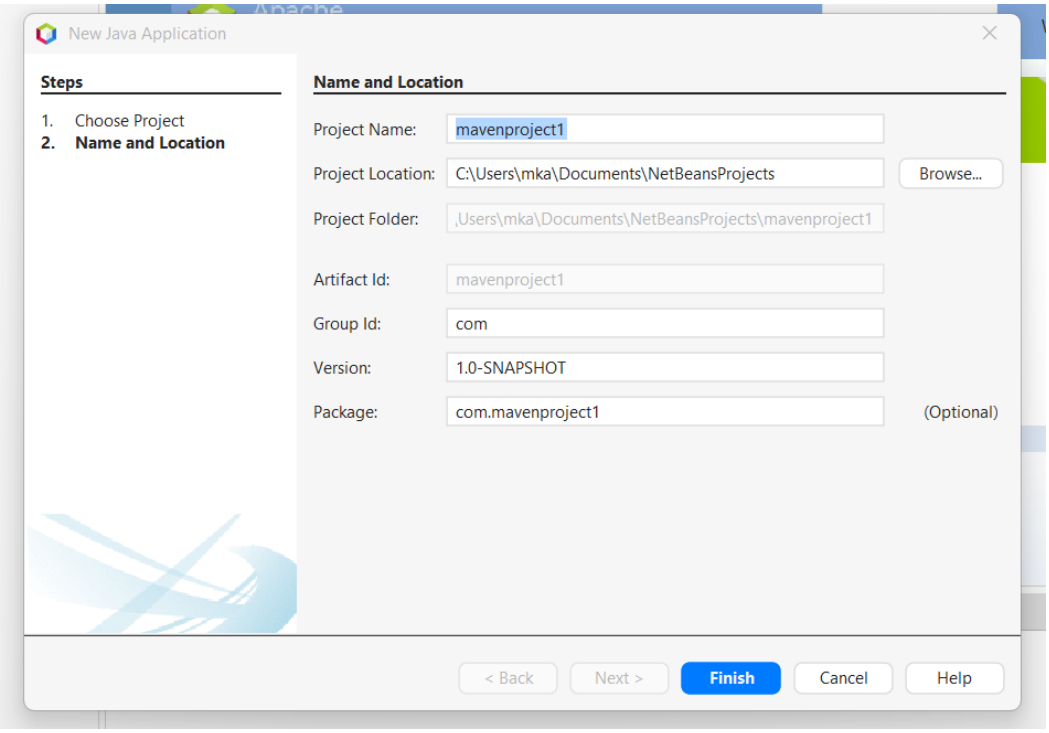
La structure du projet Java se présente alors comme suit:



15.2.3 Création d'un projet Maven dans NetBeans

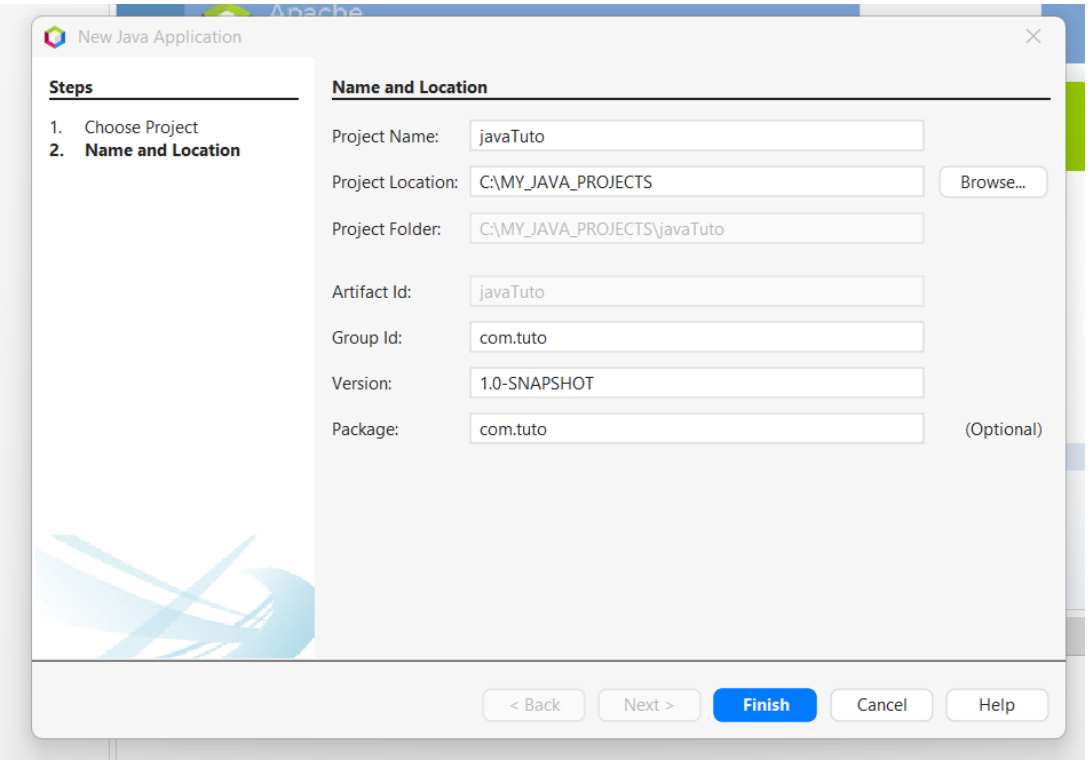
Ci-dessous les étapes de création d'un projet Java nommé javaTuto dans une structure Maven en utilisant l'IDE Netbeans.

1. Lancer l'IDE Netbeans,
2. Dans la barre des menus, cliquer File>New Project>Java with Maven. Sélectionner Java Application et cliquer sur Next. La fenêtre suivante apparaît:



- Dans le champ Project Name, indiquer javaTuto. Cette valeur sera automatiquement copiée dans le champs Artifact Id. Car le nom du projet ou du module correspond à l'artifact du projet.
- Dans le champ Project location, indiquer le chemin de votre workspace habituel des projets Java. Pour nous, il s'agit de C:\MY_JAVA_PROJECTS
- Dans le champ Group Id, indiquer le nom du package dans lequel les codes sources de votre projet seront positionnées. Ici, nous spécifions la valeur com.tuto
- Dans le champ Version, laisser la valeur par défaut qui est 1.0-SNAPSHOT. Ce champ permet d'indiquer le numéro de version correspondant au prochain build et packaging du programme. On peut spécifier n'importe quelle autre valeur qu'on souhaite. Nous reviendrons plus tard sur le build et le packaging de programme Java via l'outil Maven.
- Dans le champ Package, recopier simplement la valeur spécifiée dans le champs Group Id. Il s'agit ici de la valeur com.tuto.

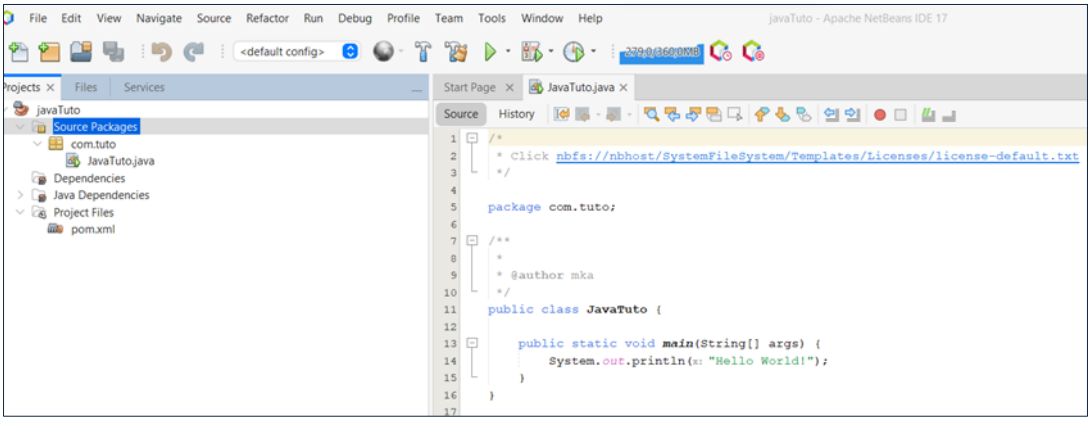
Après voir renseigné tous ces champs, la fenêtre se présente comme suit:



3. Cliquer sur Finish pour créer le projet.

En cliquant sur Finish, le projet Maven est créé avec la configuration minimale du fichier pom.xml. Et tout comme Intellij et contrairement à l'IDE Eclipse, Netbeans crée par défaut une classe template qui prend le même nom que celui du projet, c'est-à-dire javaTuto, qui est en fait une classe Main, définissant une méthode main().

La structure du projet créé se présente alors comme suit:



15.3 Structuration des dossiers et fichiers dans un projet Maven

Comme on peut le remarquer dans le projet créé, la structure de base d'un projet Maven est constituée d'un dossier nommé src et d'un fichier de configuration nommé pom.xml. Le

dossier src est le dossier qui contient les codes sources du programme tandis que le fichier pom.xml permet de spécifier les métadonnées du projet: nom, package, dépendances externes et informations de build et de packaging.

15.3.1 La structure du dossier src

Dans un projet Maven à la différence d'un projet Java libre, le dossier src est formé d'une sous-arborescence main/java et d'une sous-arborescence test/java. C'est dans l'arborescence main/java qu'est positionné l'ensemble des codes sources et fichiers ressources du programme principal. Et c'est dans l'arborescence test/java qu'est positionné l'ensemble des codes sources et fichiers ressources des tests y compris des tests unitaires. A noter que les codes sources, qu'il s'agisse du programme principal ou des tests, sont habituellement positionnés dans des packages. Ces packages représentés par des sous-répertoires définis à l'intérieur des arborescences racines main/java (pour les codes du programme principal) et test/java (pour les codes de tests). Par exemple, dans le projet créé ci-dessus, Intellij a automatiquement initialisé la classe Main dans un package nommé org.example. Ce package est un sous-répertoire du dossier src/main/java. Ainsi le chemin complet qui mène à la classe main est physiquement donc src/main/java/org/example. Au final, dans un projet Maven les arborescences src/main/java et src/test/java représentent ce qu'on appelle respectivement « Code source root» et «Test source root». Chacune de ces arborescences racines contient également un sous-dossier nommé resources qui contiennent des fichiers ressources et données utilisées soit lors du développement du programme soit lors de l'exécution du programme ou des tests.

15.3.2 La structure de base du fichier pom.xml

Intéressons-nous maintenant à la structure du fichier pom.xml. Dans le projet Maven que nous avons précédemment utilisé, cliquer sur le fichier pom.xml situé à la racine du dossier javaTuto. Le contenu du fichier se présente comme suit (il s'agit ici de la structure initialisée avec l'IDE Intellij). Des différences mineures peuvent exister avec un fichier initialisé avec Eclipse ou avec Netbeans. Mais dans l'ensemble la spécification générale du fichier pom.xml reste fondamentalement la même entre tous les IDEs.

Template minimaliste de fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>javaTuto</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>20</maven.compiler.source>
```

```
        <maven.compiler.target>20</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

Comme on peut le remarquer, le pom.xml est un fichier dont le contenu est au format XML, c'est à dire constitué d'un en-tête et d'un ensemble de balises dont chacun joue un rôle spécifique. Ci-après la description des balises présentées dans cette version minimale du pom.xml.

- La balise **<project>… </project>:** c'est la balise parente qui définit la structure générale du projet Maven. C'est à l'intérieur de cette balise que sont définies toutes les autres balises qui caractérisent le projet Maven.
- La balise **<groupId>… </groupId>** : cette balise permet de spécifier le package dans lequel sont situés les codes sources. Dans l'exemple ci-dessus la valeur du groupId est org.example. Nous modifierons cette valeur par la suite pour indiquer notre propre package. Par exemple: com.tuto.
- La balise **<artifactId>… </artifactId>**: cette balise permet d'indiquer le nom du projet Java ou le module Java dont la définition est spécifiée dans le fichier pom.xml dans un projet multi-modules. Ici, le projet est un projet mono-module nommé javaTuto.
- La balise **<version>… </version>**: permet d'attribuer un numéro de version au programme ou au package. Ce numéro de version est utile lors du packaging du programme. Il est généralement incrémenté à chaque buid d'une nouvelle version du programme à la suite d'une évolution. Dans l'exemple ci-dessus, le numéro de version attribué au package est 1.0-SNAPSHOT
- **<properties>…</properties>**: Cette balise permet de spécifier un ensemble de properties (variables) dont les valeurs sont automatiquement injectées dans le reste du fichier pom.xml et utilisées lors du build et packaging du projet Java. Dans le fichier pom.xml ci-dessus, Intellij a initialisé par défaut trois properties. Il s'agit notamment de:
  - **<maven.compiler.source>…</maven.compiler.source>** qui indique la version Java du code source. La valeur choisie est 20 qui correspond à la version du JDK que nous avons utilisée
  - **<maven.compiler.target>…</maven.compiler.target>** qui indique la version Java de l'environnement de build et d'exécution du code compile. Il s'agit également de la version 20 du JDK.
  - **<project.build.sourceEncoding>… </project.build.sourceEncoding>** qui permet de spécifier l'encodage des fichiers sources. Ici, il s'agit de UTF-8.

  Comme nous allons le voir plus tard, la balise <properties>…</properties> permet de spécifier plusieurs autres properties y compris celles définies par l'utilisateur lui-même. Ex: <my.plugin.version>1.5</my.plugin.version>. A noter que toutes ces

properties, qu'il s'agisse des properties standards Maven ou des properties définies par l'utilisateur, sont exploitables lors du build ou du packaging du projet Java.

15.4 Chargement des dépendances dans un projet Maven : ajout de la balise <dependencies>…</dependencies> au fichier pom.xml

Dans la section précédente, nous avons présenté la structure de base du fichier pom.xml qui caractérise un projet. En particulier, nous avons présenté les principales balises qui sont définies par défaut lorsqu'on initialise un projet Maven. Il s'agit en l'occurrence des balises <groupId>, <artifactId>, <version>, <packaging> et <properties> (voir plus haut, la signification et le rôle de chacune de ces balises). Cependant, la structure de base initialisée par l'IDE ne permet pas encore de gérer les dépendances externes. Il faut ajouter une balise supplémentaire en l'occurrence la balise <dependencies>…</dependencies>. Grâce à l'ajout de cette balise supplémentaire, on peut automatiquement charger les librairies externes et les rendre visibles dans le classpath de sorte à pouvoir importer et utiliser les classes qu'elles contiennent dans notre programme.

La balise <dependencies>…</dependencies> permet de charger les dépendances externes aussi bien depuis des sites internet distants (site de gestion de dépendances externes) que depuis un répertoire local situé dans le FileSystem de la machine hôte. Dans un premier temps, nous allons présenter la spécification de la balise <dependencies>…</dependencies> dans le cas d'un chargement de librairies depuis un site internet (site official de Maven, repository sur le réseau local ou tout autre site de tierces parties). Ensuite, nous ajouterons le cas où la librairie est directement chargée depuis un répertoire local situé sur la machine hôte.

15.4.1 Chargement des dépendances externes depuis un site distant

15.4.1.1 Chargement depuis le site central Maven: https://mvnrepository.com/

En reprenant le fichier pom.xml précédemment initialisé, on ajoute la balise <dependencies> tout en y ajoutant deux librairies externes que sont Log4j2 (pour le logging) et Junit (pour les tests unitaires).

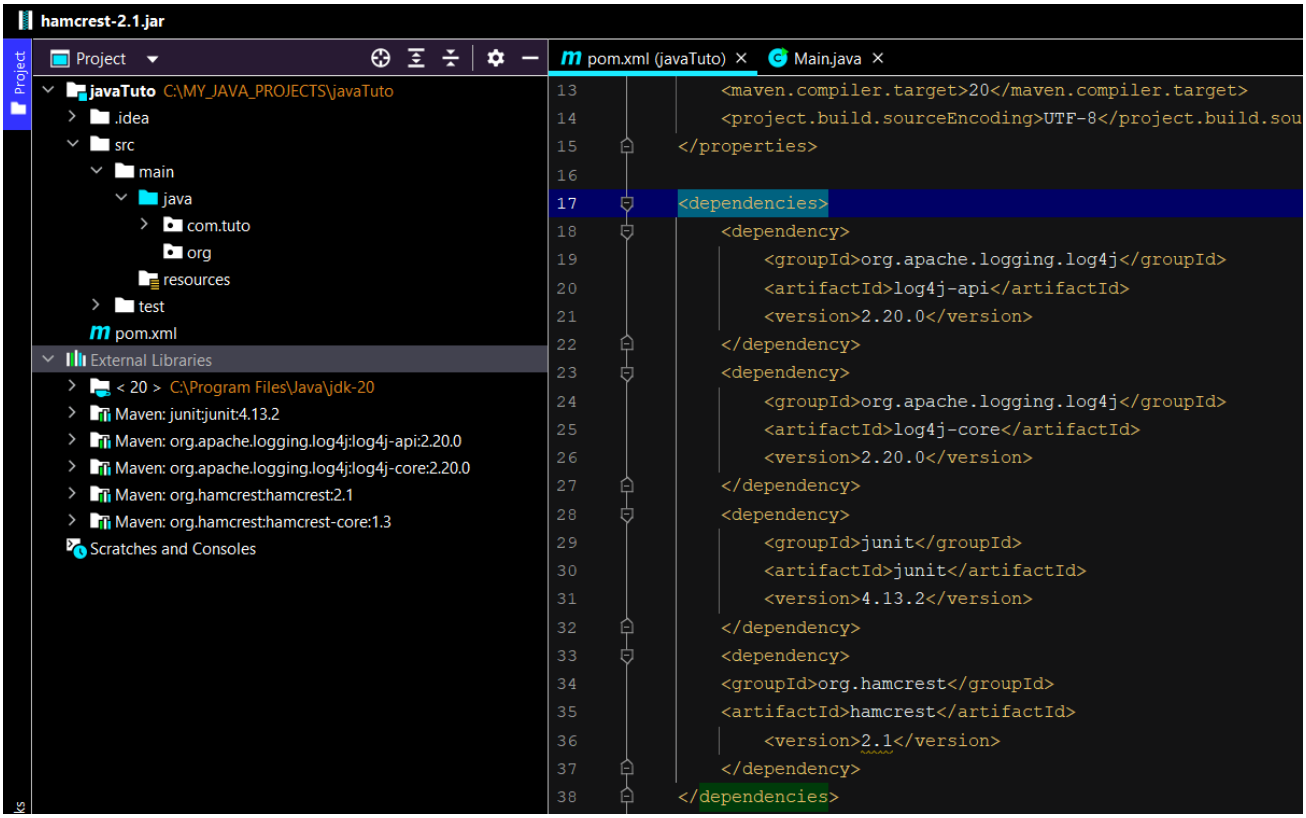
Après l'ajout d'une balise de chargement des dépendances externes, le fichier pom.xml se présente comme suit:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tuto</groupId>
    <artifactId>javaTuto</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>20</maven.compiler.source>
        <maven.compiler.target>20</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.20.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.20.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
        </dependency>
        <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest</artifactId>
            <version>2.1</version>
        </dependency>
    </dependencies>

</project>
```

Dans cet exemple, nous avons spécifié les librairies du framework Log4j2 (log4j-core et log4j-api) servant pour le logging Java. Nous avons également spécifié les librairies du framework Junit (junit et hamcrest) servant à mettre en place les tests unitaires Java. Chaque librairie est spécifiée dans une balise dédiée <dependency>…</dependency>.

A noter que, par défaut, sans aucun paramétrage supplémentaire, toutes les librairies spécifiées dans la balise <dependencies>… </dependencies> sont automatiquement téléchargées depuis le site maven central https://mvnrepository.com/. Elles sont ensuite déposées sur la machine hôte dans un répertoire local Maven créé par défaut par l'outil Maven. Ce répertoire est situé sur le home de l'utilisateur. Par exemple, ce sera C:\Users\{username}\.m2\repository pour un utilisateur Windows et ~/.m2/repository pour un utilisateur Unix/Mac Os. Ce répertoire local Maven étant automatiquement ajouté au classpath du projet Java, toutes les librairies chargées sont visibles dans l'IDE dans External Libraries. Voir la capture d'écran ci-dessous.



Toutes les librairies externes dont les noms sont préfixés par Maven: sont celles qui ont été téléchargées par l'outil Maven en se basant sur les informations spécifiées dans le fichier pom.xml.

Par ailleurs, les dépendances externes étant automatiquement installées par l'outil Maven et rendues visibles dans le classpath, on va pouvoir utiliser les classes disponibles en les important simplement dans notre propre programme. Par exemple, on peut directement importer les classes Logger, LogManager, Configurator et Level depuis la librairie Log4J2 et logguer des lignes. Voir illustration dans le bout de code ci-dessous.

```
package com.tuto;

// Import des classes depuis la librairie Log4j2 chargée avec Maven
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.core.config.Configurator;
import org.apache.logging.log4j.Level;

public class Main {

    // Appel des classes importtées depuis Log4j2.
    private static final Logger LOGGER =
LogManager.getLogger(Main.class.getName());
    public static void main(String[] args) {

        Configurator.setRootLevel(Level.DEBUG);
```

```
        LOGGER.info("Hello World");

    }
}
```

Output:

```
16:04:24.095 [main] INFO  com.tuto.Main - Hello World
```

15.4.1.2 Chargement depuis un site de tierce-partie ou un repository situé sur le réseau local.

Dans le fichier pom.xml défini précédemment, les dépendances sont automatiquement téléchargées depuis le site central de Maven (https://mvnrepository.com/). Mais il peut exister de nombreuses situations où la dépendance externe qu'on souhaite ajouter à notre projet n'est pas disponible sur le site Maven, mais plutôt sur un autre site repository. Cela peut être le cas par exemple d'une librairie développée à l'interne par une organisation tierce et mise à la disposition du public via un serveur distant. Cela peut également être le cas d'une librairie développée par un autre projet au sein de la même organisation et partagée en interne via un repository sur le réseau local entreprise (ex: un serveur Nexus interne). Dans tous ces cas, la configuration de base du fichier pom.xml via la balise <dependencies>… </dependencies> n'est plus suffisante. Il faut ajouter des options supplémentaires permettant d'ajouter des repositories de gestion de dépendances externes autres que le site Maven central. L'ajout d'autres repositories se fait via la balise <repositories>… </ repositories>. L'exemple ci-dessous illustre l'usage de la balise <repositories>… </ repositories> pour spécifier différentes repositories dans un fichier pom.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tuto</groupId>
    <artifactId>javaTuto</artifactId>
    <version>1.0-SNAPSHOT</version>

        <repositories>
            <repository>
                <id>my-repo1</id>
                <name>Site Maven central </name>
                <url>https://repo1.maven.org/maven2/</url>
            </repository>
            <repository>
                <id>my-repo2</id>
                <name>Autre repository </name>
                <url>https://repo.osgeo.org/repository/release/</url>
            </repository>
```

```
            <!-- Indiquer l'url de votre repository interne si existe
            <repository>
                  <id>my-repo3</id>
                  <name>Repository interne Nexus</name>
                  <url>{url_serveur_nexus}</url>
              </repository>
              -->
        </repositories>

    <properties>
        <maven.compiler.source>20</maven.compiler.source>
        <maven.compiler.target>20</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.20.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.20.0</version>
        </dependency>
    </dependencies>


</project>
```

Dans cette définition du fichier pom.xml, nous avons ajouté deux repositories: le repository Maven central que nous avons nommé my-repo1 dont l'url est https://repo1.maven.org/maven2 et un deuxième repository nommé my-repo2 dont l'url est https://repo.osgeo.org/repository/release. On peut ajouter autant de repositories qu'on souhaite. Par exemple, on peut aussi ajouter un repository interne tel qu'un serveur Nexus interne. L'ajout d'une balise <repositories>…</repositories> offre ainsi la possibilité de charger les dépendances externes depuis plusieurs repositories sources.

15.4.2 Chargement des dépendances depuis un fichier jar local

Il arrive parfois que la librairie externe que nous souhaitons utiliser ne soit pas disponible dans un repository distant comme Maven central, serveur Nexus interne ou tout autre repository de gestion de dépendances. Par exemple, il arrive qu'au sein d'une petite entreprise qui ne dispose pas de son propre serveur interne de gestion de dépendances, que les librairies soient distribuées entre les projets par le moyen de partage de fichiers. Dans une telle situation, le développeur fait une copie de cette librairie en local de sa machine. Mais pour pouvoir utiliser cette librairie dans son projet Maven, il doit ajouter une balise supplémentaire <dependency>… </dependency> afin d'ajouter cette librairie. Toutefois, la spécification de la balise <dependency>… </dependency> doit se faire de sorte à indiquer à Maven que la librairie externe doit être chargée depuis le système de fichier local et non à

partir d'un repository distant. Par exemple, le fichier pom.xml définit ci-dessous illustre le chargement de la librairie externe myCustomLibrary.jar depuis le dossier resources situé à la racine de notre projet Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tuto</groupId>
    <artifactId>javaTuto</artifactId>
    <version>1.0-SNAPSHOT</version>

        <repositories>
            <repository>
                <id>my-repo1</id>
                <name>Site Maven central </name>
                <url>https://repo1.maven.org/maven2/</url>
            </repository>
            <repository>
                <id>my-repo2</id>
                <name>Autre repository </name>
                <url>https://repo.osgeo.org/repository/release/</url>
            </repository>
            <!-- Indiquer l'url de votre repository interne si existe
            <repository>
                  <id>my-repo3</id>
                  <name>Repository interne Nexus</name>
                  <url>{url_serveur_nexus}</url>
              </repository>
              -->
          </repositories>

      <properties>
          <maven.compiler.source>20</maven.compiler.source>
          <maven.compiler.target>20</maven.compiler.target>
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      </properties>

      <dependencies>
          <dependency>
              <groupId>org.apache.logging.log4j</groupId>
              <artifactId>log4j-api</artifactId>
              <version>2.20.0</version>
          </dependency>
          <dependency>
              <groupId>org.apache.logging.log4j</groupId>
              <artifactId>log4j-core</artifactId>
              <version>2.20.0</version>
          </dependency>

          <!--Dépendance externe chargée sur le FileSystem local -->
          <dependency>
              <groupId>com.tuto.mylibs</groupId>
              <artifactId>myCustomLibrary</artifactId>
              <version>1.0-SNAPSHOT</version>
```

```
            <scope>system</scope>

<systemPath>${project.basedir}/src/main/resources/myCustomLibrary.jar</systemPa
th>
          </dependency>


      </dependencies>


  </project>
```

Dans cette définition, la librairie myCustomLibrary.jar est chargée depuis le répertoire >${project.basedir}/src/main/resources situé en local sur la machine hôte et non sur un site distant. Et grâce à cette spécification, toutes les classes disponibles dans la librairie sont visibles, importables et utilisables dans le programme principal.

16 COMPILATION, BUILD ET PACKAGING D'UN PROJET JAVA VIA L'OUTIL MAVEN

Dans le chapitre précédent, nous avons montré comment charger les dépendances externes dans un projet en utilisant les balises <dependencies> … </dependencies> et <repositories>…</repositories> dans le fichier pom.xml. Dans le présent chapitre, nous allons configurer le fichier pom.xml en ajoutant des balises supplémentaires de sorte à pouvoir compiler, builder et packager le projet Java en vue de son exécution.

16.1 Configuration du fichier pom.xml: ajout de la balise <build>… </build>

Pour builder un projet Maven, il faut ajouter une balise <build>… </build> en renseignant un ensemble d'informations dont les détails seront présentés plus tard. Le fichier pom.xml défini ci-dessous illustre un exemple de base permettant le build et le packaging d'un projet Maven.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.tuto</groupId>
    <artifactId>javaTuto</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

        <repositories>
            <repository>
                <id>my-repo1</id>
                <name>Site Maven central </name>
                <url>https://repo1.maven.org/maven2/</url>
            </repository>
            <repository>
                <id>my-repo2</id>
                <name>Autre repository </name>
                <url>https://repo.osgeo.org/repository/release/</url>
            </repository>
            <!-- Indiquer l'url de votre repository interne si existe
            <repository>
                  <id>my-repo3</id>
                  <name>Repository interne Nexus</name>
                  <url>{url_serveur_nexus}</url>
              </repository>
              -->
          </repositories>

      <properties>
          <maven.compiler.source>20</maven.compiler.source>
          <maven.compiler.target>20</maven.compiler.target>
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      </properties>
```

```
      <dependencies>
          <dependency>
              <groupId>org.apache.logging.log4j</groupId>
              <artifactId>log4j-api</artifactId>
              <version>2.20.0</version>
          </dependency>
          <dependency>
              <groupId>org.apache.logging.log4j</groupId>
              <artifactId>log4j-core</artifactId>
              <version>2.20.0</version>
          </dependency>
      </dependencies>

    <!-- Etape de build -->
    <build>
              <plugins>
                  <plugin>
                      <groupId>org.apache.maven.plugins</groupId>
                      <artifactId>maven-shade-plugin</artifactId>
                      <version>3.2.4</version>
                      <executions>
                          <execution>
                              <phase>package</phase>
                              <goals>
                                  <goal>shade</goal>
                              </goals>
                              <configuration>
                                  <transformers>
                                      <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransfo
rmer">
                                          <mainClass>com.tuto.Main</mainClass>
                                      </transformer>
                                  </transformers>
                              </configuration>
                          </execution>
                      </executions>
                  </plugin>
            </plugins>

    </build>

  </project>
```

Cette définition appelle quelques petits commentaires. D'abord, dans l'en-tête du fichier, nous avons défini le type de packaging avec la balise <packaging>…</packaging>. Nous avons choisi la valeur jar qui signifie que le package généré sera un fichier de type jar. Pour rappel, un projet Maven peut être buildé dans d'autres types comme war (pour les web services) et pom (pour un packaging sous forme de spécification projet). Dans notre cas ici, il s'agit d'un package de type jar. Une remarque peut être également faite sur les balises <artifactId>…</artifactId> et <version>…</version> . Pour rappel artifactId représente le nom du projet Java et version permet d'attribuer un numéro de version au package à générer. Au final c'est la combinaison des valeurs des balises artifactId, version et packaging qui permet de définir le nom complet du package à générer. Dans le cas présent, le package qui sera généré après le build se nommera javaTuto-1.0-SNAPSHOT.jar.

Maintenant, concernant la balise <build>…</build>, nous avons choisi la configuration minimale constituée d'une seule balise enfant <plugins>… </plugins>, elle-même constituée d'une seule balise enfant <plugin>… </plugin>. La balise <plugins>… </plugins> est l'élément de base nécessaire dans la balise <build>…</build> permettant de définir les outils nécessaires pour builder et packager le projet Maven avec l'ensemble de ses dépendances pour générer un fat jar. En effet, pour générer un package contenant le programme principal et l'ensemble des dépendances externes, nous avons besoin de spécifier un plugin Maven en l'occurrence le plugin maven-shade-plugin. La spécification de ce plugin est accompagnée par un ensemble d'informations comme la balise <executions>…</executions>. Dans le cas présent, nous avons défini une balise <execution>…</execution> décorée par un ensemble de sous-balises dont <phase>, <goal> et <configuration>. Pour avoir des détails sur le rôle de chacune de ces balises dans un fichier pom.xml, consulter la documentation Maven: https://maven.apache.org/pom.html.

16.2 Build et packaging du projet Maven

Le cycle de vie d'un projet Maven se déroule en plusieurs phases allant de la validation du code au déploiement du package dans un repository de gestion de dépendance. Pour avoir plus de détails sur l'ensemble des phases du cycle de vie d'un projet Maven, consulter la page suivante: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

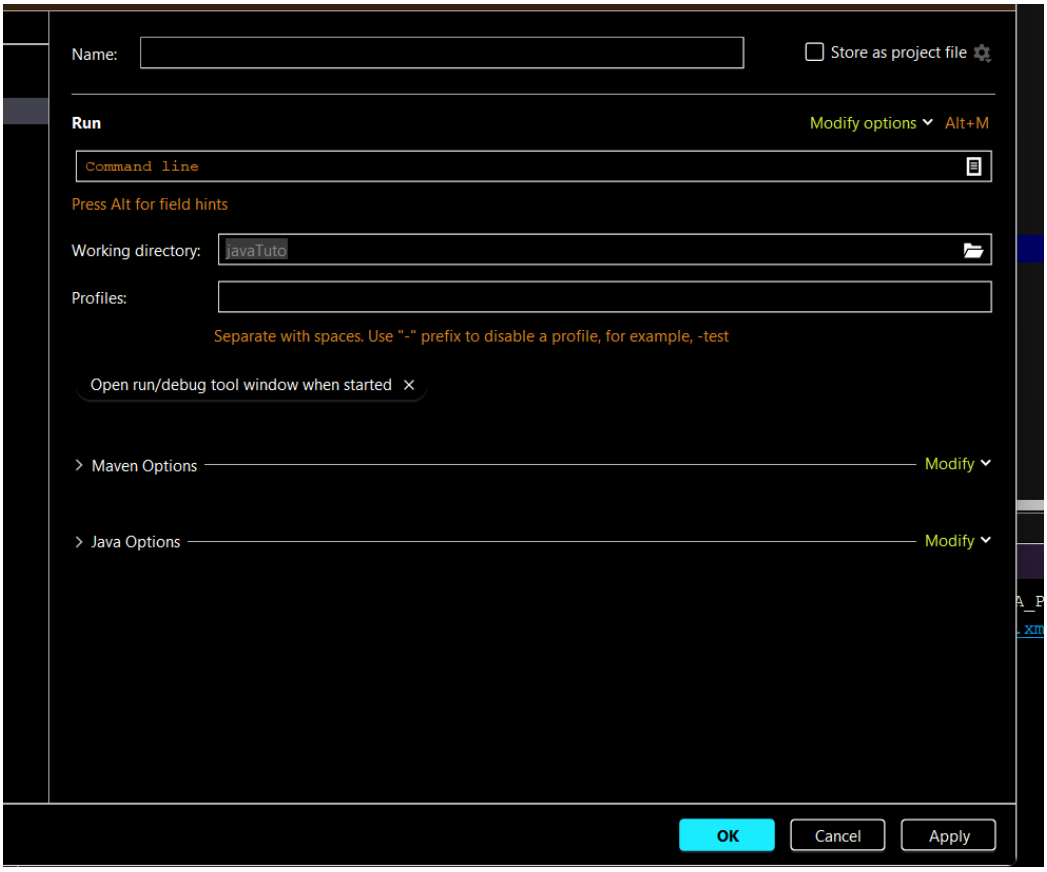
Ici, nous nous focalisons sur le build et le packaging de l'application Java, c'est-à-dire la génération du fichier jar contenant le programme principal et l'ensemble de ses dépendances externes. Dans un environnement shell, l'outil Maven fournit des commandes permettant le build et le packaging d'un projet Java. Il s'agit en l'occurrence des commandes mvn compile et mvn package. La première permet de compiler l'ensemble du projet Java pour vérifier que tout est ok dans le programme: accessibilité des classes depuis les librairies externes, import et instanciation des classes, appel des méthodes, etc… La commande mvn compile permet aussi de générer des bytecodes du programme puisqu'elle permet de générer des fichiers portant l'extension .class à partir des fichier sources portant l'extension .java. Quant à la commande mvn package, elle permet générer le package du projet Java suivant le type de package spécifié dans le pom.xml qui, on le rappelle, peut être le type jar, war ou pom. Dans notre cas ici, nous avons plutôt choisi le type jar.

Par ailleurs, notons que la plupart des IDEs offre des fonctionnalités permettent de lancer ces commandes directement dans l'IDE. C'est le cas par exemple de l'IDE Intellij, Eclipse ou même Netbeans. Les sous-sections ci-dessous illustrent le packaging des projets Maven dans chacun des IDEs.

16.2.1 Packaging du projet Maven sous l'IDE Intellij

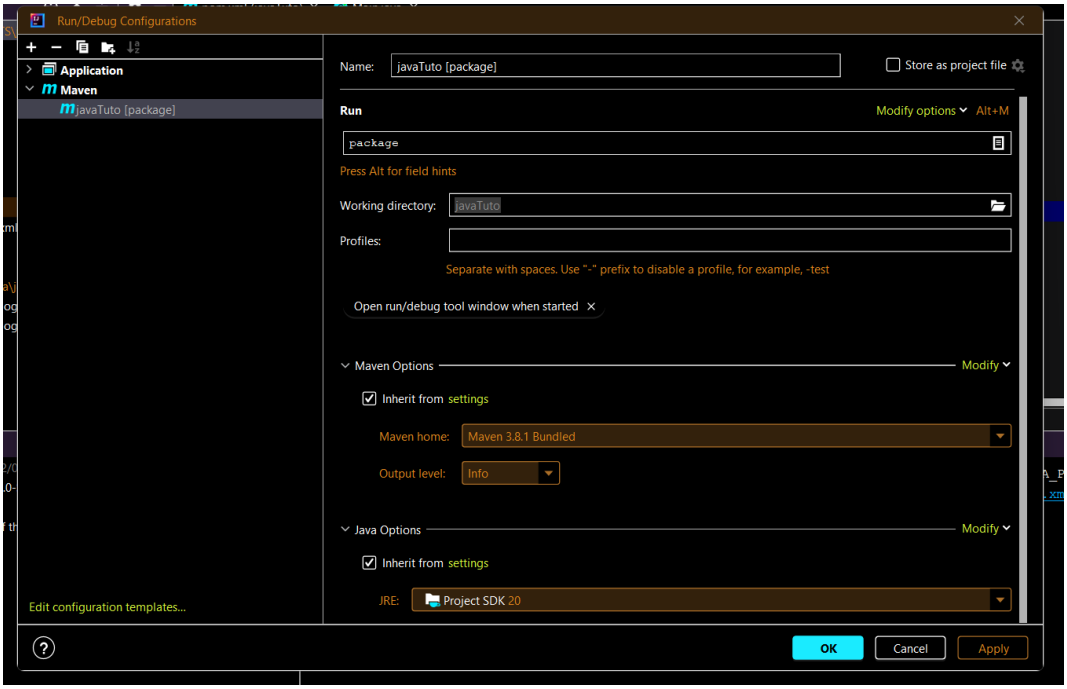
Pour packager le projet sous l'IDE Intellij, suivre les étapes suivantes:

1. Cliquer dans le menu Run>Edit configuration> Add new configuration>Maven. On obtient ainsi la fenêtre suivante:



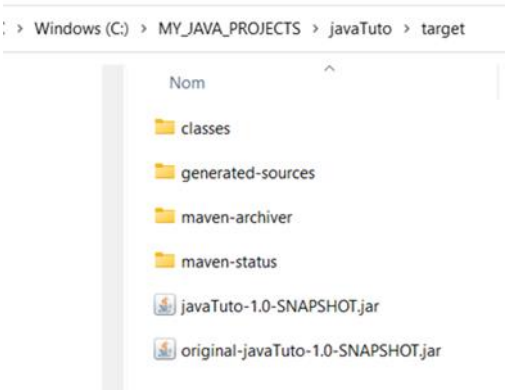
2. Dans le champ Name, indiquer le nom de votre projet Java ou de votre module. Pour notre cas ici, il s'agit de javaTuto.
3. Dans le champ Run, taper le mot package. Celui-ci permet d'indiquer qu'on souhaite générer le package. Noter que Intellij propose plusieurs autres choix dont notamment: compile, clean, install, deploy, validate, verify, test, etc… voir la liste de l'ensemble des phases du cycle de vie d'un projet Maven sur cette page: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Après avoir renseigné ces informations, la fenêtre de dialogue se présente alors comme suit:



4. Valider cette configuration en cliquant sur Apply et OK.
5. Cliquer une nouvelle fois dans le menu Run>Run 'javaTuto [package]'.

Après l'exécution, vérifier dans le contenu du répertoire ${project.basedir}/target. Pour notre cas, il s'agit de C:\MY_JAVA_PROJECTS\javaTuto\target. Le contenu de ce dossier se présente comme suit:



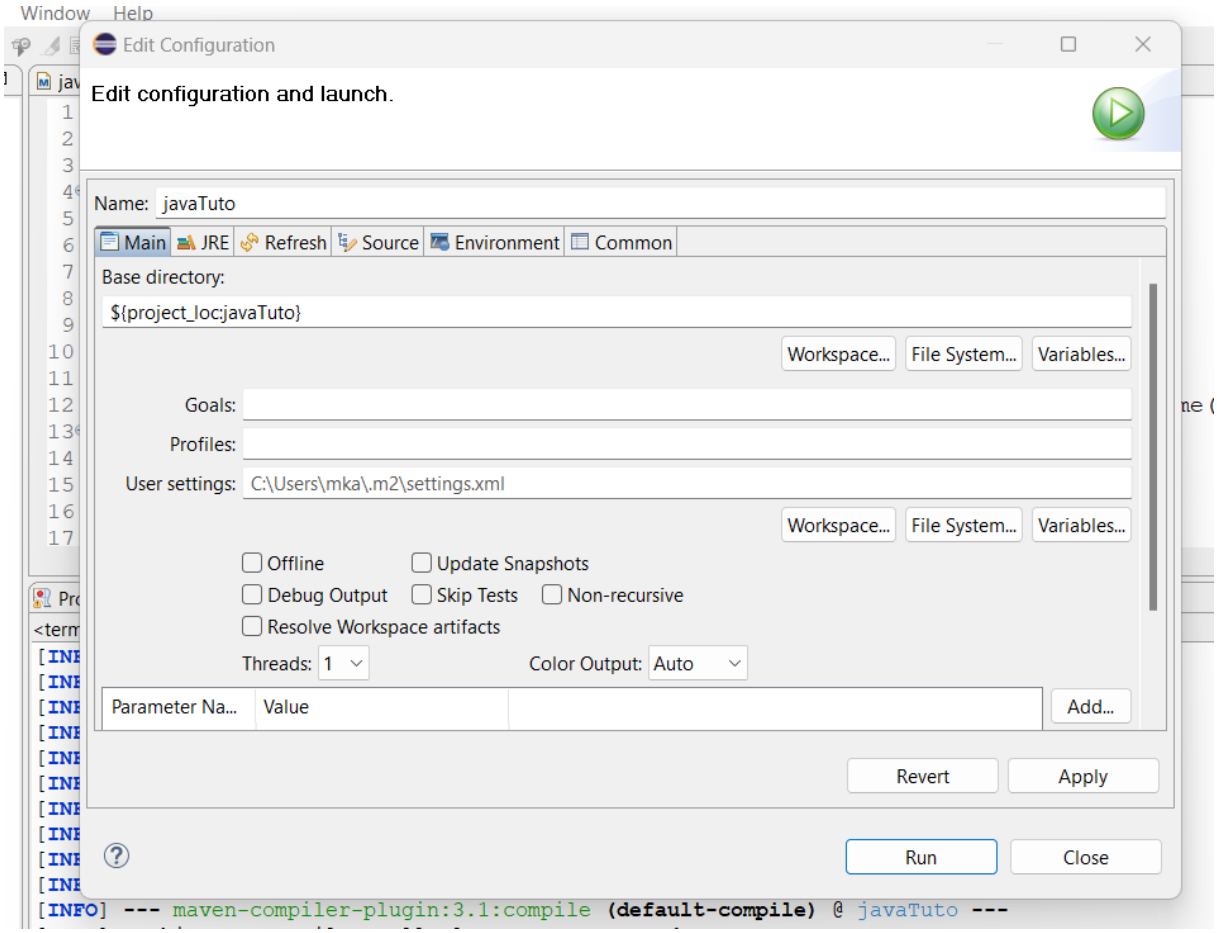
Dans ce dossier on remarque la présence de deux fichiers jars: original-javaTuto-1.0-SNAPSHOT.jar et javaTuto-1.0-SNAPSHOT.jar. Le premier représente le package contenant le programme principal sans ses librairies externes. Le deuxième représente le package qui contient à la fois le programme principal et ses différentes dépendances externes. On l'appelle communément le "fat jar" ou "jar with dependencies". Bien qu'il soit beaucoup plus lourd, le fat jar offre l'avantage de disposer d'un package portable et exécutable sur tous les environnements Java sans aucune autre configuration ou de chargement de librairies externes. Car le package embarque non seulement le code

principal mais aussi l'ensemble des dépendances externes nécessaires pour compiler et exécuter le code.

16.2.2 Packaging du projet Maven sous l'IDE Eclipse

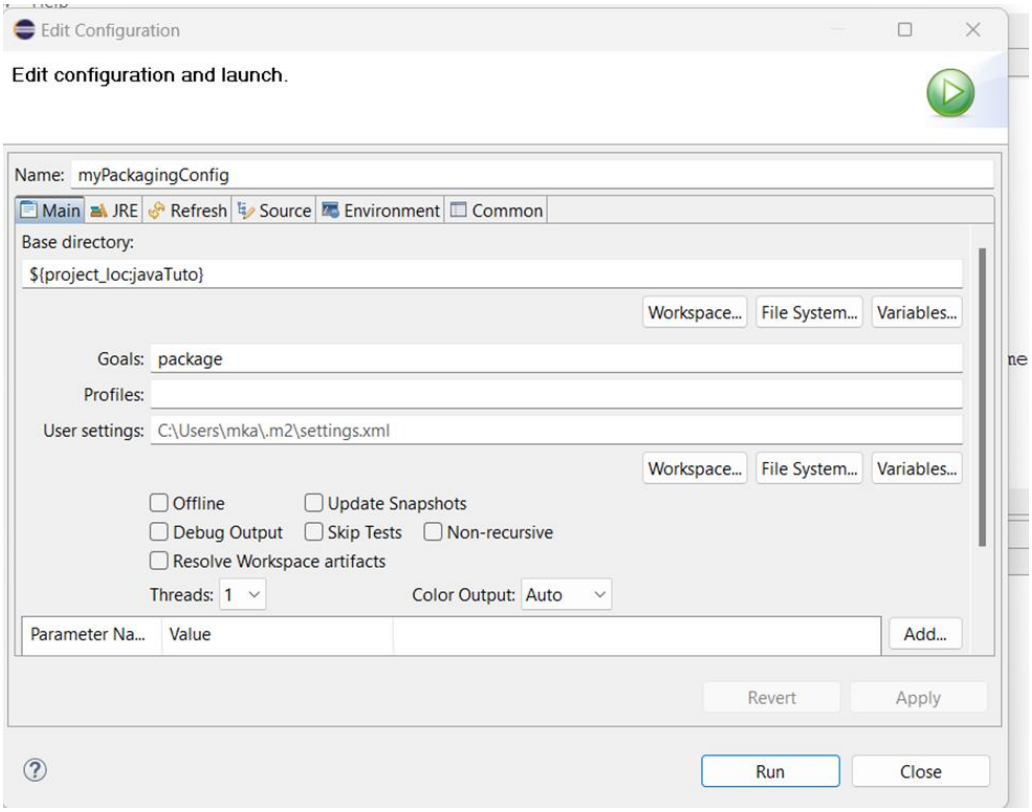
Pour packager le projet sous l'IDE Eclipse, suivre les étapes suivantes:

1. Cliquer dans le menu Run>Run> Add new configuration. Choisir Maven build…. Cliquer sur OK. On obtient ainsi la fenêtre suivante:



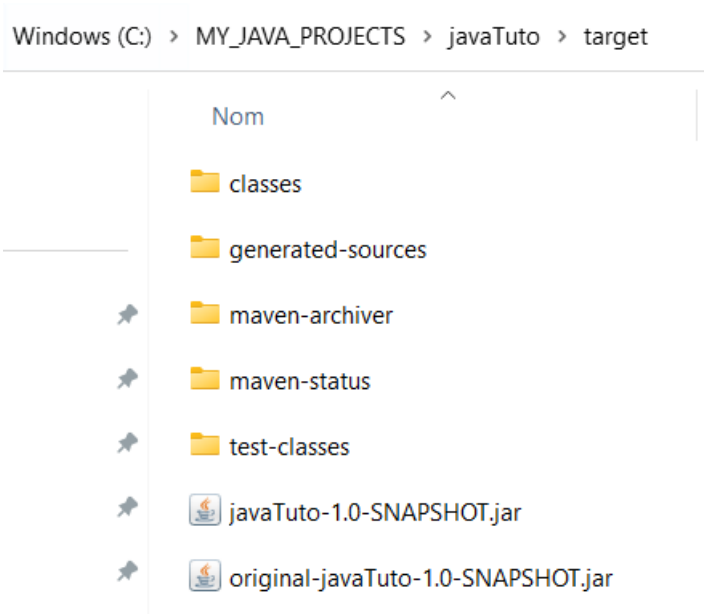
2. Dans le champ Name, spécifier le nom que vous souhaitez donner à cette configuration. Ex: myPackagingConfig
3. Dans le champ Goals, indiquer la valeur package.
4. Cliquer sur Apply pour appliquer ces configurations. Noter qu'on pouvait renseigner des paramètres supplémentaires, notamment fournir un fichier setting.xml plus personnalisé, ou renseigner des informations de profile. Mais ici on se limite à la configuration minimale.

La fenêtre de configuration se présente alors comme suit:



5. Cliquer sur Run pour exécuter.

A la suite de l'exécution, les fichiers suivants seront générés dans le dossier target à la racine du projet.



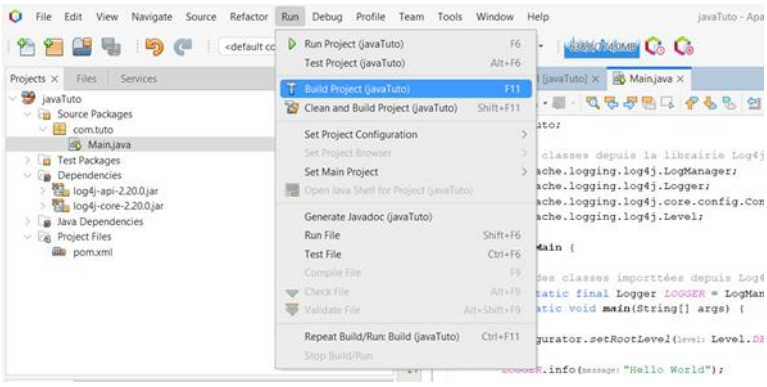
Dans ce dossier, on remarque la présence de deux fichiers jars: original-javaTuto-1.0-SNAPSHOT.jar et javaTuto-1.0-SNAPSHOT.jar. Le premier représente le package contenant le programme principal sans les librairies externes. Tandis que le deuxième représente le package qui contient à la fois le programme principal et ses différentes

dépendances externes. Ce package est communément appelé “fat jar” ou “jar with dependencies”. Bien qu’il soit beaucoup plus lourd, le fat jar offre l’avantage de disposer d’un package portable et exécutable sur tous les environnements Java sans aucune autre configuration ou de chargement de librairies externes. Il embarque non seulement le code principal mais aussi l’ensemble des dépendances externes nécessaires pour compiler et exécuter le code.

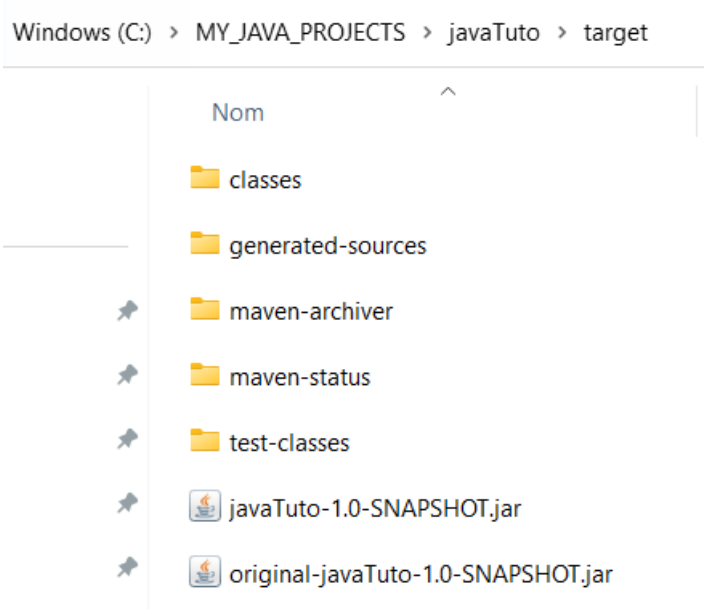
16.2.3 Packaging du projet Maven sous l’IDE Netbeans

Pour packager le projet sous l’IDE Netbeans, suivre les étapes suivantes:

1. Cliquer dans le menu Run>Build project.



Cette action fait le build du projet et génère automatiquement le package dans le dossier target situé à la racine du projet. Voir capture d’écran ci-dessous.



On remarque dans ce dossier, la présence de deux fichiers jars: original-javaTuto-1.0-SNAPSHOT.jar et javaTuto-1.0-SNAPSHOT.jar. Le premier représente le package contenant le programme principal sans les librairies externes. Tandis que le deuxième

représente le package qui contient à la fois le programme principal et ses différentes dépendances externes. Ce package est communément appelé “fat jar” ou “jar with dependencies”. Bien qu’il soit beaucoup plus lourd, le fat jar offre l’avantage de disposer d’un package portable et exécutable sur tous les environnements Java sans aucune autre configuration ou de chargement de librairies externes. Il embarque non seulement le code principal mais aussi l’ensemble des dépendances externes nécessaires pour compiler et exécuter le code.

17 EXECUTER LE PROGRAMME JAVA

17.1 Rappels sur l'étape de compilation des codes sources

Faison remarquer d'entrée que Java est un langage interprété, en ce sens que les codes sources ne sont pas directement exécutables par le système d'exploitation. Ils doivent d'abord être soumis à un système tiers (en l'occurrence la JVM) qui joue le rôle d'interprète lors de l'exécution.

Rappelons que les langages comme C++ ou Go, etc.. sont des langages compilés. Leur codes sources sont directement convertis en langage machine par le compilateur et soumis comme tels à exécution au système d'exploitation. Les langages comme Python sont des langages purement interprétés. Les codes sources sont directement soumis à un interpréteur pour exécution sans aucune étape de compilation. Quant au langage Java et d'autres langages comme Groovy, Scala, ils sont considérés comme des langages semi-interprétés. En effet, leurs codes sources (les fichiers avec l'extension .java) sont d'abord compilés dans un format intermédiaire appelé bytecode (fichiers avec l'extension .class). Ce bytecode est ensuite soumis à la Machine Virtuelle Java (JVM) qui l'interprète et l'exécute.

Le cycle de vie d'un code source Java se résume donc à l'étape de compilation et d'exécution dans un environnement fourni par la JVM. Dans le chapitre précédent, nous avons montré comment à travers un outil de gestion de projet comme Maven, on peut écrire les codes sources, les compiler et les packager. Dans ce présent chapitre, qui se veut très bref, nous montrons comment exécuter le code Java pré-compilé et pré-packagé.

17.2 Vérification de l'installation Java

A noter que pour pouvoir exécuter du code Java sur un sytème d'exploitation, Java doit d'abord être installé avec au minimum la composante JRE (Java Runtime Environment). Nous avons déjà vu dans la section 2.1 comment installer Java notamment le JDK et préparer l'environnement de développement. Pour rappel, l'installation du JDK n'est nécessaire que lors de la phase de développement du programme. Lorsqu'il s'agit uniquement d'exécuter le code Java sur un environnement, l'installation de Java avec le composant JRE est suffisant.

Pour vérifier si Java est correctement installé sur votre environnement, taper la commande shell suivante :

java –version

Cette commande devrait produire un résultat similaire à celui-ci.

```
java version "20"
Java(TM) SE Runtime Environment (build 20+36-2344)
Java HotSpot(TM) 64-Bit Server VM (build 20+36-2344, mixed mode,
sharing)
```

17.3 Lancer l'exécution du programme

Pour exécuter le programme Java que nous avons préalablement buildé et packagé avec l'outil Maven, nous suivons les étapes suivantes.

1- Dans notre workspace C:\MY_JAVA_PROJECTS, nous créons un sous-dossier nommé run. Ce sous-dossier fait office d'espace de livraison des packages Java. On aurait pu aussi choisir n'importe quel autre répertoire situé en dehors du projet. Par exemple un répertoire comme C:/applications/run pour un système de type Windows ou un répertoire comme /applications/run pour un système de type Unix, etc..

2- Copier le fichier javaTuto-1.0-SNAPSHOT.jar depuis le répertoire target où a été généré le package. Dans notre cas, il s'agit du répertoire C:\MY_JAVA_PROJECTS\javaTuto\target. Et déposer le fichier copié dans le sous-dossier run que vous avez précédemment créé.

   Il faut noter ici que dans un contexte industrialisé, le package que nous avons buildé aurait été déployé sur un serveur de centralisation de packages comme Nexus. Mais ici, étant donné que nous sommes dans un cadre tutoriel, pour des besoins d'illustration, le package est généré et déposé dans le répertoire target. A partir de ce répertoire, il peut être copié sur l'environnement d'execution.

3- Après avoir copié le fichier javaTuto-1.0-SNAPSHOT.jar dans le répertoire run, ouvrir une session shell sur votre machine et exécuter les commandes ci-dessous

Sous Windows

```
set CLASSPATH=C:\MY_JAVA_PROJECTS\run\javaTuto-1.0-SNAPSHOT.jar
java -cp %CLASSPATH% com.tuto.Main
```

Sous Unix/Linux

```
export CLASSPATH=C:\MY_JAVA_PROJECTS\run\javaTuto-1.0-SNAPSHOT.jar
java -cp $CLASSPATH com.tuto.Main
```

Dans cette commande, nous commençons d'abord par définir le classpath de l'application. Ce classpath pointe sur le fichier javaTuto-1.0-SNAPSHOT.jar qui est en fait une archive jar qui contient le code du programme principal ainsi que toutes les dépendances externes dont il a besoin pour s'exécuter. Ce package est généralement qualifié de « fat jar » ou « jar with dependencies ».

Ensuite, nous lançons la commande java avec l'option -cp (abrégé de classpath) auquel on attribue la valeur de la variable CLASSPATH que nous avons précédemment définie. En plus du classpath, nous spécifions aussi la classe Main à exécuter. Dans notre cas ici, il s'agit de la classe com.tuto.Main.

En exécutant, notre petit programme Java qui fait simplement du logging Log4j2, nous obtenons le résultat suivant :

```
WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will
impact performance.
17:24:51.690 [main] DEBUG com.tuto.Main - Début d'exécution du programme
17:24:51.692 [main] INFO  com.tuto.Main - Happy Birthday to You
17:24:51.692 [main] DEBUG com.tuto.Main - Fin d'exécution du programme
```

18 RESSOURCES DOCUMENTAIRES

Je tiens ici à remercier tous les auteurs de documents techniques, d'articles, de blogs et de ressources numériques que j'ai consultés lors de la revue documentaire mais qui ne figurent pas dans la liste ci-dessous.

Bibliographie

Arnold, Ken, James Gosling et David Holmes (2005), The Java Programming Language, 3rd edition, NJ: Prentice Hall.

Balagurusamy, E, (2007), Programming with JAVA: A Primer, 3rd edition, New Delhi: Tata McGraw-Hill.

Burd Barry, (2019,Java pour les Nuls, 4e éd. Broché, First Interactive, ISBN-10 2412046638

Cadenhead, Rogers (2000), Teach Yourself Java in 24 Hours, New Delhi: SAMS Techmedia.

Deitel, Harvey et Paul Deitel (2003), JAVA How to Program, 5th edition. NJ: Prentice Hall.

Delannoy Claude, (2020), Programmer en Java: Couvre Java 10 à Java 14, EYROLLES, ISBN-10 2416000187

Eckel, Bruce (2003), Thinking in JAVA, 3rd edition, NJ: Prentice Hall.

Evans Ben et Flanagan David, (2019),Programmer avec Java - Concepts fondamentaux et mise en oeuvre par l'exemple , First Interactive, ISBN-10 2412045127

Flanagan, David (2002). JAVA in Nutshell. Sebastopol, CA: O'reilly Media.

Gervais Luc, (2020), Apprendre la Programmation Orientée Objet avec le langage Java - (avec exercices pratiques et corrigés) (3e édition), Editions ENI, ISBN-10 2409026303

Groussard Thierry et Richard Thierry, (2019), Javva 11 - Les fondamentaux du langage (avec exercices pratiques et corrigés), Editions ENI, ISBN-10 2409020607

Herby Cyrille, (2018), Apprenez à programmer en Java, EYROLLES, ISBN-10 2212675216

Horstmann, Cay S, (2006), Big JAVA, 2nd edition,John Wiley.

Horstmann, Cay S. and Gary Cornell (2005). Core JAVA 2, 7th edition. NJ: Pearson Education.

Kanerva, Jonni (2004), The Java FAQ. NJ: Prentice.

Tasso Anne,(2019),Le livre de Java premier langage: Avec 109 exercices corrigés Broché, EYROLLES, ISBN-10 2212678401

Urls documents numériques

https://dev.java/

https://docs.oracle.com/en/java/javase/20/docs/api/index.html

http://sdz.tdct.org/sdz/apprenez-a-programmer-en-java.html

https://fr.bitdegree.org/tutos/programmer-en-java/

https://jenkov.com/tutorials/java/index.html

https://koor.fr/Java/Tutorial/Visibilite.wp

https://mkyong.com/tutorials/java-8-tutorials/

https://www.baeldung.com/get-started-with-java-series

https://www.data-transitionnumerique.com/apprenez-programmation-java/

https://www.guru99.com/java-tutorial.html

https://www.jannaud.fr/java

https://www.java.com/fr/

https://www.javaguides.net/p/java-tutorial-learn-java-programming.html

https://www.javatpoint.com/java-tutorial

https://www.jmdoudoux.fr/java/dej/chap-poo.htm

https://www.mygreatlearning.com/blog/java-tutorial-for-beginners/

https://zestedesavoir.com/tutoriels/646/apprenez-a-programmer-en-java/