***Les versions de java de la 8 à la 19***

1. Java 8

***Les lambda expressions***

• Une Lambda Expression est tout simplement une fonction qui peut être définie sans appartenir à une classe

• évite d'avoir à écrire le code nécessaire à la déclaration de la classe anonyme et de la méthode.

• Elle peut être passée en paramètre comme un objet et exécutée à la demande

• Particulièrement utile notamment lorsque le traitement n'est utile qu'une seule fois : elle évite d'avoir à écrire une méthode dans une classe

• Une expression lambda permet d'encapsuler un traitement pour être passé à d'autres traitements. C'est un raccourci syntaxique aux classes anonymes internes pour une interface qui ne possède qu'une seule méthode abstraite. Ce type d'interface est nommé interface fonctionnelle.

• La syntaxe d'une expression lambda est composée de trois parties :

• un ensemble de paramètres, d'aucun à plusieurs

• l'opérateur ->

• le corps de la fonction

***Les interfaces***

• Une interface est un contrat.

• La vitrine de l'objet=> ce qu'il fait mais pas comment il le fait

• Lorsqu'un objet client d'une interface collabore avec celle-ci, il ne connait pas son type réel mais il sait comment le "manipuler" (quels messages il peut lui envoyer).

• Le but des interfaces est de diminuer le couplage entre deux classes.

• L'interface permet de ne présenter au client "que ce qui l'intéresse".

• Un autre aspect est la protection des variations.

• un objet client d'une interface peut collaborer avec n'importe quel type d'objet pourvu qu'il réalise l'interface.

• Les Interfaces ne peuvent avoir que des propriétés de type:

public static final (des constantes)

• On y accède avec : NomInterface.NomConstante

• L'héritage multiple entre interface est autorisé

• Dans les précédentes versions de Java, seules les méthodes public et sans corps pouvaient apparaître dans une interface

• Aujourd'hui il est possible de définir des méthodes (ver.8 et plus) :

• static => permet de spécifier les factory méthodes par exemple

• default => des méthodes avec un corps. Celles-ci pourront être utilisées par la classe d'implémentation telles quelles ou bien redéfinie (override)

• private=> à partir de Java 9

***L’API java.time p51***

***Parcours de l'API : LocalDate***

• D'abord il faut distinguer entre Local date/time et Zoned time

• Local date/time fait référence à une date locale et/ou une heure du jour

• E.g. LocalDate.now() =>2020-04-30

• E.g. LocalTime.now()=>01:20:42.493

• E.g. LocalDateTime.now()=> 2020-04-30T01:20:42.493

• Pour créer une LocalDate

//maintenant

LocalDate aujourdhui=LocalDate.now();

//les mois commencent par 1

LocalDate independanceD=LocalDate.of(1789, 7, 14);

//avec l'enum Month

LocalDate independanceC=LocalDate.of(2020, Month.MAY, 11);

• Pour calculer la différence entre deux dates=>Period

Period.between(independanceD, independanceC)).toTotalMonths()

***LocalTime***

• Représente l'heure et on peut la créer des façons suivantes :

LocalTime nowt=LocalTime.now();

System.out.println(nowt);//02:35:36.191

LocalTime dinerTime=LocalTime.of(20, 00);

System.out.println(dinerTime);//20:00

***Zoned Time***

• Certaines applications requièrent la prise en compte du décalage horaire et la prise en compte d'heures d'été/hiver

• Conférence prévue à Paris à 10h mais vous êtes à NY!

• Chaque zone time a un ID. Plus de 600 ID répertoriés par [www.iana.org](http://www.iana.org)

• Ex. America/New\_York

• ZoneId.of(id)=>retourne un objet ZoneId positionné sur la zone avec l'id passé en paramètre

Parser une chaîne en date

• Utiliser la méthode parse("chaine")

LocalDate nye=LocalDate.parse("2022-01-01");

System.out.println(nye); //2022-01-01

1. Java 9

***Interfaces: les méthodes private***

• Nouveautés Java 9

• Utilité: parfois plusieurs méthodes default on besoin d'exécuter le même bout de code. Ce dernier est sorti dans une méthode privée de l'interface

• Règles sur les méthodes private:

• Ne peuvent pas être abstraites

• Sont utilisées uniquement dans l'interface

• Les méthodes private non static ne peuvent pas être appelées par une méthode private static

1. Java 10

***Local-variable type inference : var***

Une des nouveautés les plus attendues de Java 10 .

• Utilisée dans les cas suivants:

• Variables locales avec initialisation

• Index dans des boucles

• Dans le contexte des lambda expressions

var list = new ArrayList(); // infers ArrayList

var stream = list.stream(); // infers Stream

Limites :

• Ne peut pas être utilisé sur des variables non initialisées

var actor;

• Impossible pour la déclaration de plusieurs variables

var x = 5, y = 10;

• Null ne pas être utilisé pour initialisée une var

var author = null; // Null cannot be inferred to a type

***Collections : copyof(Colleection c)***

Ajout d'une méthode copyOf(Collection) pour les List, Map et Set. Retourne une collection (copie) non modifiable de la collection passée en paramètre. Si la collection source change, la copie ne sera pas impactée

List actors = new ArrayList<>();

actors.add("Jack Nicholson");

actors.add("Marlon Brando");

System.out.println(actors); // prints [Jack Nicholson, Marlon Brando]

// New API added - Creates an UnModifiable List from a List.

List copyOfActors = List.copyOf(actors);

System.out.println(copyOfActors); // prints [Jack Nicholson, Marlon Brando]

// copyOfActors.add("Robert De Niro"); Will generate an UnsupportedOperationException

1. Java 11

***Améliorations***

• La commande java en ligne de cde (JEP 330)

• Maintenant elle compile et exécute en même temps

• var pour les Lambda

***Des packages supprimés***

• Des packages sur Java EE et CORBA ont été supprimés

• Étaient en Deprecated dans la version 9

***Amélioration sur HTTP Client (JEP 321)***

• Java 11 supporte HTTP/1.1 et HTTP/2

• Meilleures performances sur l'envois/réception de requêtes/réponses à partir d'un client vers le serveur

• Support natif de WebSockets

***Epsilon Garbage Collector***

• Gère les allocations d'objets en mémoire (heap) mais pas leur suppression??!! Utilité alors?=>

• Tests de performances: mesurer les performances sans bruit/biais du process du GC

• Des applications à durée de vie très courtes. Lancer le GC peut retarder leur fin

• Pour l'activer, comme tout autre GC, avec l'option en line de commandes: **-XX:+UseEpsilonGC**

• Démo avec code sur Eclipse!

1. Java 12

***L'expression Switch***

• Possibilité de plusieurs labels dans le même case

• Retourner des valeurs à travers le break (puis yield depuis la version 13)

• Retourner les valeurs à travers le symbole ->

private static int getValueViaArrow(String mode){

int result = switch (mode) {

case "a", "b" -> 1;

case "c" -> 2;

case "d", "e", "f" -> { // do something here...

System.out.println("Supports multi line block!");

yield 3;

}

default -> -1; };

return result;

}

***La méthode File.mismatch()***

• Compare deux fichiers

public static long mismatch(Path path, Path path2) throws IOException

• Retourne

• La position du premier mismatch ou -1L si les fichiers sont identiques

• La taille du fichier le plus petit en cas de différence de taille des fichiers

• Démo sur Eclipse

***Une nouvelle version d'instanceof***

• Avant

if (obj instanceof String) {

String s = (String) obj; // use s in your code from here

}

• Après

if (obj instanceof String s) {

// can use s directly here

}

1. Java 13

Deux grosses nouveautés pour les développeurs sont apparues sur Java 13, **les blocs de texte** ainsi que l’apparition du mot clé **« yield »** sur les expressions switch.

Ensuite plusieurs nouveautés moins visibles pour les développeurs ayant un impact sur la mémoire et les performances ont été ajoutées, en l’occurrence la nouvelle implémentation de l’API Socket, une amélioration de la ZGC et enfin sur l’archivage dynamique des classes.

En ce qui concerne les blocs de textes ainsi que le mot clé « yield », il s’agit ici d’évolutions visibles uniquement sur la preview de Java 13.

Ces évolutions sont présentes dans le JDK mais ne sont pas activées par défaut. Il faut les activer explicitement pour pouvoir les utiliser. Il est préférable d’éviter de les utiliser en production car elles risquent d’être supprimées ou retirées dans les prochaines releases.

En ce qui concerne l’installation, tout d’abord il vous faut télécharger le [**JDK 13**](https://jdk.java.net/13/).

Les blocs de textes permettent d’écrire du texte sur plusieurs lignes sans avoir à utiliser le caractère **« \n »,**mais également d’utiliser des **caractères spéciaux** sans avoir à les échapper avec le caractère **« \ ».**

StringJoiner sj = new StringJoiner(" \n");

sj.add("SELECT U.NAME FROM USER U");

sj.add("INNER JOIN CITY C ON C.ID = U.CITY\_ID");

sj.add("WHERE CITY.NAME ? ");

sj.add("AND U.STATUS = ?");

String sql = sj.toString();

Voici l’écriture simplifiée en utilisant les blocs de textes de Java 13 :

String sqlWithTextBlock = """

SELECT U.NAME FROM USER U

INNER JOIN CITY C ON C.ID = U.CITY\_ID

WHERE CITY.NAME ?""";

Pour résumer ce qu’il faut retenir des blocs de texte :

* Il faut commencer les blocs de texte par “”” et un retour à la ligne et les terminer également avec “””
* Toute l’indentation est conservée en dehors de l’indentation automatique (accidentelle de l’ide)
* Il n’est plus nécessaire d’échapper les caractères spéciaux dans les blocs de texte

**Évolution sur les expressions « switch » dans Java 13**

De même que pour les « blocs de texte » les switch expressions restent également en preview. Le mot clé « yield » a été ajouté aux switch expressions, celui-ci remplace le break et permet de sortir de l’expression switch courante.

String errorLabel = switch (errorCode) {

case 404:

yield "Non trouvé!";

case 418:

yield "Je suis une théière!";

case 500:

yield "Erreur interne!";

default:

yield "Code inconnu!";

};

1. Java 14

### *Switch expressions standardisées (*[*JEP361*](https://openjdk.java.net/jeps/361)*)*

Ça y est, après une Preview (Java 12) et une Second Preview (Java 13), il aura fallut 2 versions pour enfin standardiser les switch expressions.

Concrètement, il n’y a aucun changement de cette fonctionnalité entre la version 13 et 14 de Java (excepté le fait qu’il n’est plus nécessaire d’activer le mode preview).

Cependant, voici une piqûre de rappel:

Avant, on utilisait les switch expressions de manière très verbeuse, où le break perd de son sens et les cases sont répétés ligne par ligne :

var weekPartType;

**switch** (day) {

**case** MON:

**case** TUE:

**case** WED:

**case** THU:

**case** FRI:

weekPartType = WORK\_WEEK;

**break**;

**case** SAT:

**case** SUN:

weekPartType = WEEK\_END;

**break**;

**default**:

**throw** new UnsupportedOperationException("Unknown day: " + day);

}

Maintenant, on peut l’écrire comme les lambdas :

var weekPartType = **switch** (day) {

**case** MON, TUE, WED, THU, FRI -> WORK\_WEEK;

**case** SAT, SUN -> {

yield WEEK\_END;

}

**default** -> **throw** new UnsupportedOperationException("Unknown day: " + day);

}

Chaque bloc de code dans une switch expression doit impérativement terminer par un yield, qui vient donc replacer le break.

1. Java 15

***Les classes sealed***

C'est la grosse nouveauté de Java 15, nécessaire à l’arrivée du filtrage par motif dans le langage, les sealed classes ont pour objectif de restreindre l’implémentation ou l’héritage de celles-ci.  
La déclaration d’une classe comme sealed se fait de la manière suivante :

package com.example.geometry;

public abstract sealed class Shape

    permits Circle, Rectangle, Square {...}

Le mot clef permits sert à lister les classes autorisés à hériter de la sealed class.  
Il est aussi possible d’inclure ces sous-classes directement dans le même fichier où la sealed class est déclarée, sans avoir à définir permits :

package com.example.geometry;

abstract sealed class Shape {...}

... class Circle    extends Shape {...}

... class Rectangle extends Shape {...}

... class Square    extends Shape {...}

Grâce aux sealed classes, il sera prochainement possible de faire du filtrage par motif de cette manière :

|  |
| --- |
| int getCenter(Shape shape) {      return switch (shape) {          case Circle c    -> ... c.center() ...          case Rectangle r -> ... r.length() ...          case Square s    -> ... s.side() ...      };  } |

Les sealed classes apportent cependant quelques contraintes :

* L'héritage doit être explicité dans la classe parente
* Les sealed classes et leur sous classes doivent être dans le même module ou dans le même package
* Les sous classes doivent directement hériter de leur sealed classe
* Les sous classes doivent être déclarées comme sealed, non-sealed ou final pour décrire comment la hiérarchie de celles-ci s'achève.

Notez qu’une interface peut aussi être définit comme sealed.  
Cette fonctionnalité est disponible en aperçu et sera rendu finale avec le JDK 16 ou 17.

## *Les Text Blocks en version définitive*

Apparu avec le JDK 13, légèrement modifié avec [Java 14](https://www.arolla.fr/blog/2020/06/quoi-de-neuf-dans-java-14/), les Text Blocks, qui comme leur nom l’indique permettent d’écrire un bloc de chaînes de caractères sur plusieurs lignes sans concaténation sont enfin disponible en version définitive dans Java 15.

|  |
| --- |
| String html = """                <html>                    <body>                        <p>Hello, world</p>                    </body>                </html>                """; |

## *Suppression du moteur Nashorn*

Nashorn, le moteur JavaScript introduit avec JDK 8, déprécié avec Java 11, est supprimé car il était trop coûteux à maintenir en vu de la très rapide évolution d'ECMAScript.

## Livraison de nouveaux Garbage Collectors

Les ramasses miettes ZGC et Shenandoah ne sont plus expérimentaux et sont livrés en version finale dans le JDK 15. Cependant G1 reste pour le moment le ramasse miette par défaut.

## Le JDK sur GitHub

OpenJDK, qui utilisait Mercurial depuis 2008 comme logiciel de gestion de version a migré le JDK vers Git pour différentes questions d’outillage et d’hébergement, et c’est finalement GitHub qui a été sélectionné pour héberger le [code source du JDK](https://github.com/openjdk/jdk).

1. Java 16

## *Les Records*

La première grande fonctionnalité du langage fournie dans Java 16 s'appelle les Records. Les Records permettent de représenter des données sous forme de données dans du code Java plutôt que sous forme de classes arbitraires. Avant Java 16, lorsque nous avions simplement besoin de représenter certaines données, nous nous retrouvions avec une classe arbitraire comme celle indiquée dans l'extrait de code ci-dessous.

public class Product {

private String name;

private String vendor;

private int price;

private boolean inStock;

}

Ici, nous avons une classe Product qui a quatre membres. Cela devrait être toutes les informations dont nous avons besoin pour définir cette classe. Bien sûr, nous avons besoin de beaucoup plus de code pour que cela fonctionne. Par exemple, nous avons besoin d'un constructeur. Nous avons besoin de méthodes getter correspondantes pour obtenir les valeurs des membres. Pour le compléter, nous devons également avoir des implémentations d'equals(), hashCode() et toString() qui sont congruentes avec les membres que nous avons défini. Une partie de ce code boilerplate peut être généré par un IDE, mais cela présente certains inconvénients. Vous pouvez également utiliser des frameworks comme Lombok, mais ils présentent également certains inconvénients.

Ce dont nous avons vraiment besoin, c'est d'un mécanisme dans le langage Java pour décrire plus précisément ce concept d'avoir des classes de données uniquement. Et donc en Java 16, nous avons le concept de Records. Dans l'extrait de code suivant, nous avons redéfini la classe Product en tant que record.

public record Product(

String name,

String vendor,

int price,

boolean inStock) {

}

## *Pattern matching avec instanceof*

Cela nous amène au deuxième changement du langage dans Java 16, et c'est le pattern matching avec instanceof. C'est une première étape dans un long voyage pour apporter le pattern matching à Java. Pour l'instant, je pense que c'est déjà très bien que nous ayons le support initial dans Java 16. Jetez un coup d'œil à l'extrait de code suivant.

if (o instanceOf String) {

String s = (String) o;

return s.length();

}

Vous reconnaîtrez probablement ce schéma où un bout de code vérifie si un objet est une instance d'un type, dans ce cas la classe String. Si la vérification est réussie, nous devons déclarer une nouvelle variable locale, effectuer un cast et affecter la valeur, et seulement alors nous pouvons commencer à utiliser la variable typée. Dans notre exemple, nous devons déclarer la variable s, cast o en un String et ensuite appeler la méthode length(). Bien que cela fonctionne, c'est verbeux, et ce n'est pas vraiment un code révélateur de l'intention. Nous pouvons faire mieux.

A partir de Java 16, nous pouvons utiliser la nouvelle fonctionnalité de pattern matching. Avec le pattern matching, au lieu de dire o est une instance d'un type spécifique, nous pouvons faire correspondre o à un pattern de type. Un motif de type se compose d'un type et d'une variable de liaison. Voyons un exemple.

if (o instanceOf String s) {

return s.length();

}

Ce qui se passe dans le bout de code ci-dessus, c'est que si o est bien une instance de String, alors String s sera immédiatement liée à la valeur de o. Cela signifie que nous pouvons immédiatement commencer à utiliser s comme une chaîne de caractères sans un cast explicite à l'intérieur du corps du if. L'autre chose intéressante ici est que la portée de s est limitée au corps de if. Une chose à noter ici est que le type de o dans le code source ne doit pas être un sous-type de String, car si c'est le cas, la condition sera toujours vraie. Et donc, en général, si le compilateur détecte que le type d'un objet qui est testé est un sous-type du type du motif, il lancera une erreur de compilation.

Une autre chose intéressante à souligner est que le compilateur est assez intelligent pour déduire la portée de s selon que la condition est évaluée à true ou false comme vous le verrez dans l'extrait de code suivant.

if (!(o instanceOf String s)) {

return 0;

} else {

return s.length();

}

1. Java 17

***Classes et interfaces scellées***

Nous sommes tous familier avec l’héritage, pilier fondateur de la logique de programmation orientée objet. Avec les classes scellées, il est désormais possible de définir avec précision le périmètre d’une classe, en définissant quelles classes seront autorisées à en hériter. Cela fonctionne aussi avec les interfaces et leurs implémentations. La mise en place de cette nouveauté passe par l’ajout de 2 mots clés : **sealed** et **permits**. Voyons tout cela au moyen d’un exemple :

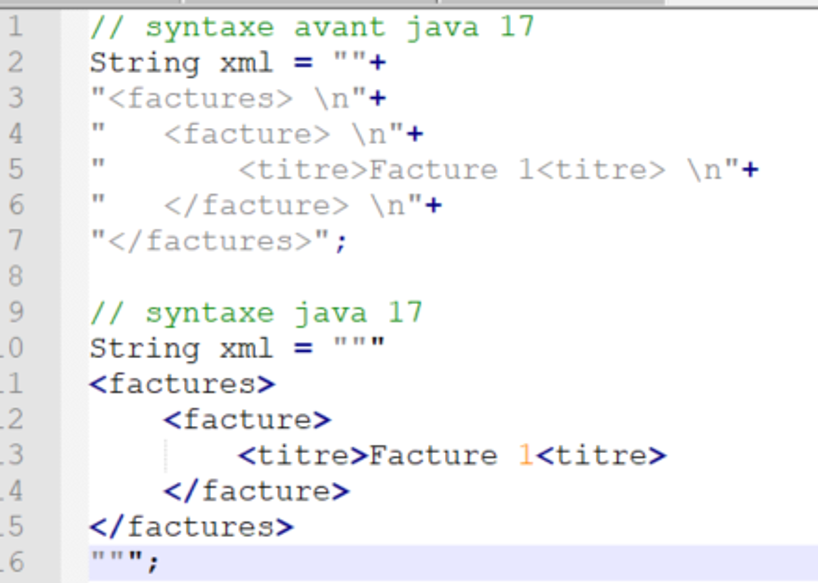
Public sealed interface Service permits Voiture, Camion{

}

J’ai défini ici une interface scellée nommée **Service** (notez la présence du mot-clef *sealed*). Seules les classes **Voiture** et **Camion** pourront implémenter cette interface, grâce au mot-clef *permits*.

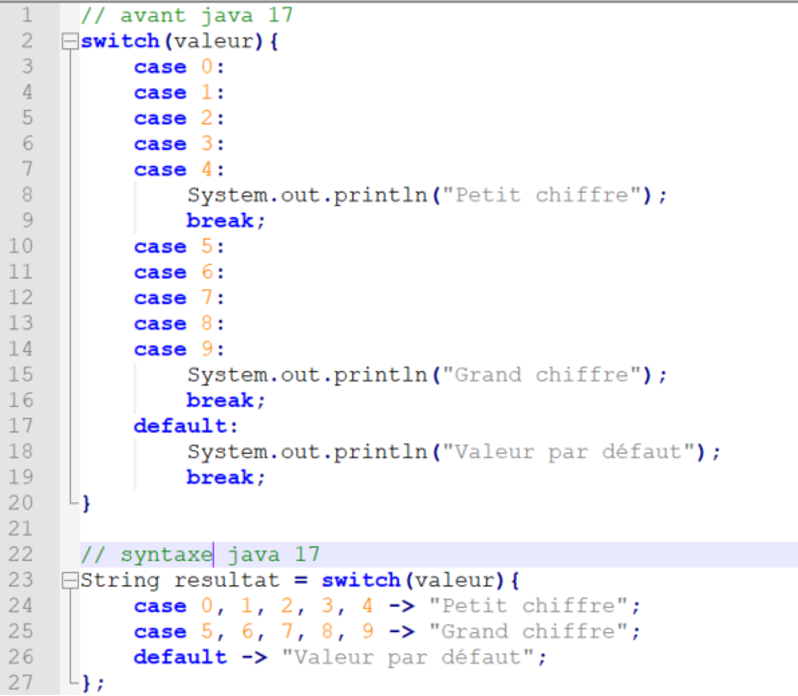
***Blocs de texte***

Il est enfin possible de créer des blocs de texte de façon simple ! Cette nouvelle syntaxe s’avère très utile pour écrire du xml ou du json, par exemple.



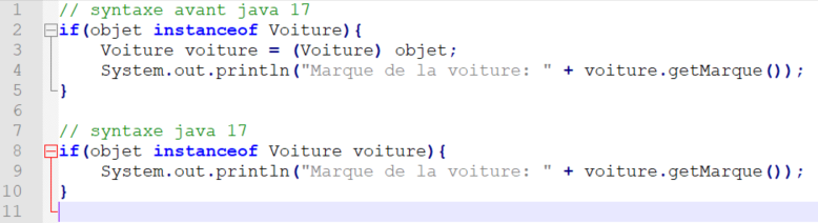
***Nouveau switch***

La nouvelle syntaxe des switch permet une réduction drastique du nombre de lignes de code :



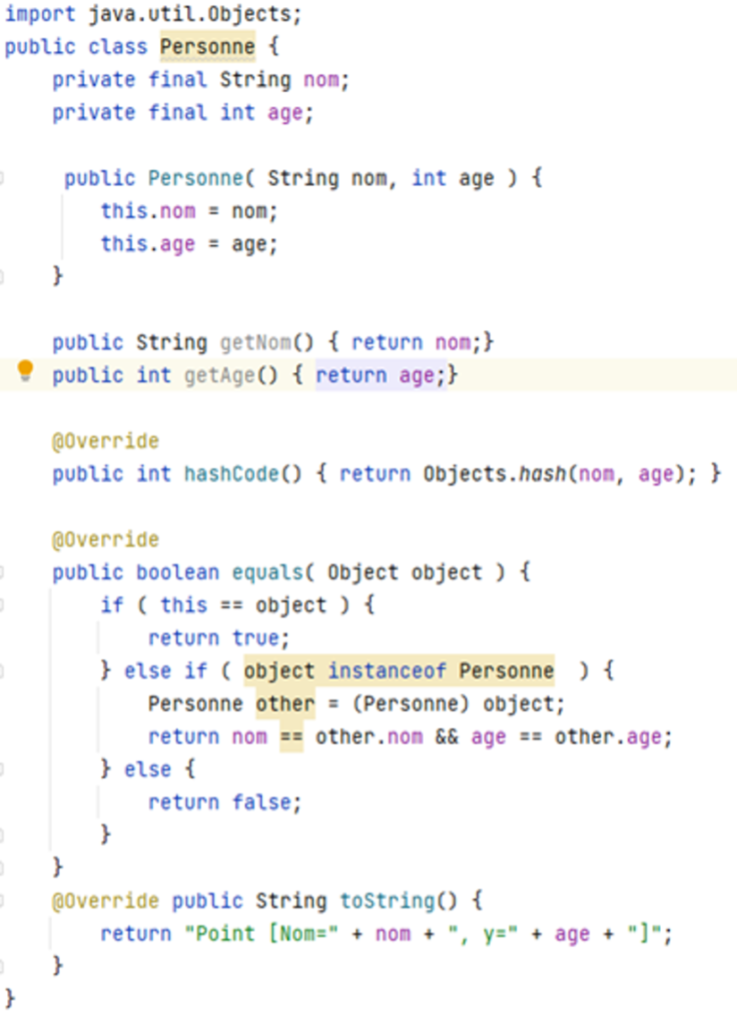
***Pattern matching***

Cette fonctionnalité concerne le mot-clé **instanceof**, qui permet de vérifier le type d’un objet.



***Type Record***

Un record permet de facilement créer des objets immutables. Voilà un exemple avec une classe **Personne**, qui contient des données liées à une personne *(nom, âge…)*.  
Voilà la syntaxe utilisée dans la plupart des cas avant Java 17 pour définir cette classe :



Et voilà la nouvelle syntaxe Java 17 :

