



NOVAO[®]
LEARNING

JAVA POO



CONCEPT D'OBJET

QU'EST-CE QU'UN OBJET ?

Un objet est une représentation d'une chose, réelle ou abstraite.

Imagine que tu veux modéliser une voiture. Cette voiture a des **caractéristiques** comme la couleur, le modèle et la marque. Elle peut aussi **faire des actions** comme démarrer, freiner ou accélérer.

On peut donc décrire un objet en développement avec:

- Des **CARACTERISTIQUES**
- Des **ACTIONS**

Exemples:

- *Chien:*
 - *Caractéristiques: nom, couleur, race, poids, ...*
 - *Actions: manger, aboyer, renifler, ...*
- *Bicyclette:*
 - *Caractéristiques : nombre de vitesse, vitesse courante, couleur, ...*
 - *Actions : tourner, accélérer, changer de vitesse, ...*

POURQUOI UTILISER L'OBJET ?

Les variables de types primitifs sont trop restreintes.

Comment définir une voiture par seulement un String ?

Comment définir une personne avec un seul type ? Son âge ? Son nom ? Son prénom ? Son adresse ?

On pourrait créer plusieurs variables (String nom; String prenom, int age;). Mais rien dans le code ne nous dit que ces variables définissent la même personne.

L'objet en informatique permet justement cela, avec une sorte de « boîte » qui contient les informations, ainsi que les actions, que peuvent effectuer un concept. Le tout permettant aussi de mieux organiser et structurer le code.

L'OBJET EN JAVA

En JAVA, techniquement, tout est un objet.

Une classe peut être décrit comme étant un modèle décrivant les caractéristiques communes et le comportement d'un ensemble d'objets.

La classe est un **moule** et l'**objet** est ce qui est moulé à partir de cette classe.

L'état de chaque objet est indépendant des autres:

- Les objets sont des représentations dynamiques (appelées **instances**) du modèle défini au travers de la classe.
- Une classe permet d'instancier autant d'objets que nécessaires. Ils auront en commun leur moule, mais seront tous indépendant des autres.
- Chaque objet est une instance d'une seule classe.
- Une classe créée pour définir un objet est donc un « type » de variable que l'on crée nous-même.



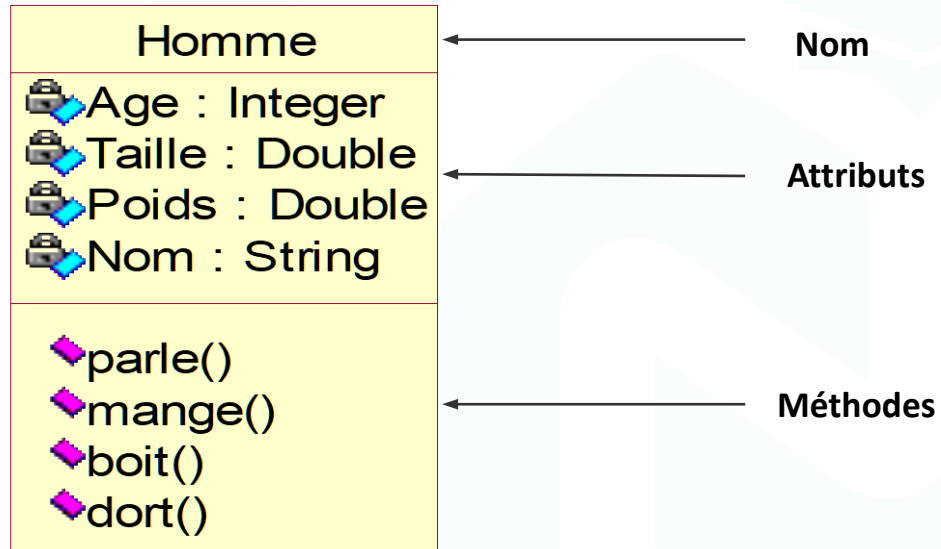
SYNTAXE

LA CLASSE EN JAVA

En JAVA, une classe sera composée de plusieurs choses:

- Un **nom**: explicite pour bien définir son utilité
- Des **attributs**: des champs nommés ayant une valeur. Ils caractérisent l'état de l'objet pendant l'exécution du programme.
- Des **méthodes**: des méthodes représentant le comportement de l'objet de cette classe. Elles manipulent les attributs des objets et caractérisent les actions pouvant être effectués par les objets.

REPRÉSENTATION GRAPHIQUE



SYNTAXE DE LA CLASSE

```
public class Chien {
```

Nom

```
    public String race;  
    public String couleur;
```

Attributs

```
    public void aboyer() {  
        System.out.println("Wouaf Wouaf");  
    }
```

```
    public void manger() {  
        System.out.println("Je mange des croquettes");  
    }
```

Méthodes

```
    public void dormir() {  
        System.out.println("Je dors dans mon panier");  
    }
```

```
}
```

INSTANCIATION

DÉFINITION

Si la classe est le **moule** de l'objet. La création de l'objet grâce à ce moule est appelé **instanciation**. L'objet alors créé sera appelé une **instance**.

L'instanciation peut donc être vu comme la concrétisation d'une classe en un objet « concret » et exploitable.

Cela est très similaire à la création de variables de type primitif (int, char, double, ...). On pourra créer autant d'instance que nécessaire.

On a une classe voiture et on veut gérer 3 voitures ? On pourra instancier 3 voitures avec la classe, qui seront **indépendantes**, comme n'importe quelle variable.

Par abus de langage « instance » et « objet » sont tous les deux utilisés pour définir la même chose.

SYNTAXE

```
public class Main {  
  
    public static void main(String[] args) {  
        Chien monChien = new Chien();  
    }  
}
```

Le type de la variable « monChien »
est le nom de la classe

On utilise le mot clé **new** pour créer une nouvelle instance de chien.
Le **Chien()** est une fonction appelée **constructeur**.

LES CONSTRUCTEURS

Toutes les classes créées ont un constructeur par défaut, qui, si on l'écrit nous-même, ressemble à ceci:

```
public NomDeLaClasse () {  
}
```

Cela permet d'instancier un objet avec la classe en prenant les valeurs par défaut des attributs créés dans celle-ci.

On peut nous-même redéfinir un ou plusieurs constructeurs avec le comportement que l'on veut. Si on veut pouvoir instancier un *chien* en définissant tous ses attributs à la création, on peut créer un constructeur comme celui-ci:

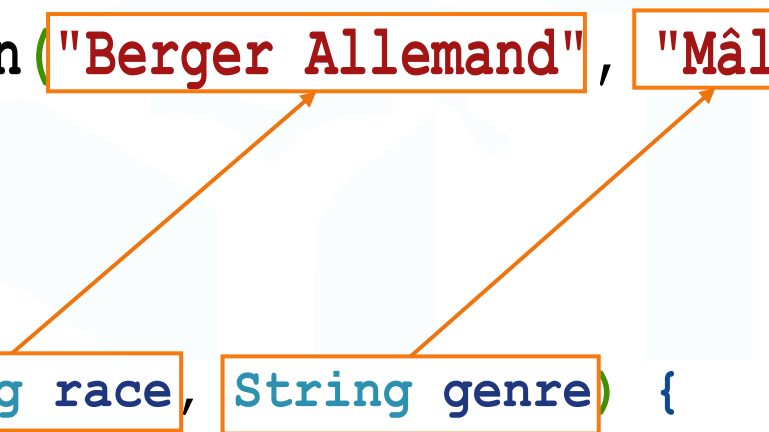
```
public Chien(String race, String genre) {  
    this.race = race;  
    this.genre = genre;  
}
```

Le **this** représente la classe.
Donc **this.race** veut dire qu'on fait
référence à l'attribut **race** dans la classe.

UTILISATION DU CONSTRUCTEUR

```
Chien monChien = new Chien("Berger Allemand", "Mâle");
```

```
public Chien(String race, String genre) {  
    this.race = race;  
    this.genre = genre;  
}
```



The diagram consists of two orange rectangular boxes. The first box, located below the first code line, contains the text 'String race'. The second box, located below the second code line, contains the text 'String genre'. Two orange arrows originate from these boxes: one points from 'String race' to the first argument 'Berger Allemand' in the code above, and the other points from 'String genre' to the second argument 'Mâle' in the code above.

A noter que lorsque l'on crée nous-même ce type de constructeur, celui par défaut n'existera plus.

Mais rien ne nous empêche de le recréer nous-même !

```
Chien[] tabChiens = new Chien[3];  
List<Chien> listChiens = new ArrayList<>();
```

MULTIPLE CONSTRUCTEUR

On peut très bien avoir plusieurs constructeurs dans une classe. Chacun permettant d'instancier la classe de différentes manières. Cela s'appelle de la **surcharge** de méthodes.

Généralement lorsqu'on crée plusieurs constructeurs on a:

- Constructeur vide permettant d'instancier chaque attribut avec sa valeur par défaut.
- Constructeur avec tous les attributs en paramètre.

```
public class Chien {  
  
    public String race;  
    public String genre;  
  
    public Chien() {  
    }  
  
    public Chien(String race, String genre) {  
        this.race = race;  
        this.genre = genre;  
    }  
}
```

ACCES AUX ATTRIBUTS ET METHODES

RÉCUPÉRER LES ATTRIBUTS

Après avoir créé une instance d'un objet, on a accès non seulement à la variable mais également aux différents attributs pour pouvoir les récupérer et en faire ce que l'on veut (l'afficher, la transmettre à une autre variable, l'utiliser dans un calcul, etc...).

Mais le seul moyen de pouvoir accéder aux attributs est d'utiliser une instance !
Elles n'existent qu'à travers l'instance.

Pour cela il faut juste faire: `nomVariable.nomAttribut`

```
Chien monChien = new Chien("Berger Allemand", "Mâle");  
System.out.println(monChien.genre);  
String genre = monChien.genre;
```

Affiche « femelle » dans la console

Récupère « femelle » pour le transférer à une nouvelle variable

MODIFIER LES ATTRIBUTS

De la même manière que l'on récupère les attributs pour les utiliser, on utilise la même nomenclature pour les modifier.

Que les attributs aient des valeurs ou non, cela va remplacer ou donner une valeur à l'attribut de l'instance.

```
monChien.genre = "Femelle";  
monChien.race = "Labrador";
```

ACCÈS AUX MÉTHODES

Tout comme les attributs, lorsqu'on instancie un objet, on a accès aux méthodes de celle-ci et on peut y accéder comme pour les attributs.

Comme n'importe quelles autres méthodes. On peut utiliser du VOID ou du RETURN.

Encore une fois, on n'oublie pas que la SEULE façon d'y accéder est par une instance.

```
Chien monChien = new Chien();
```

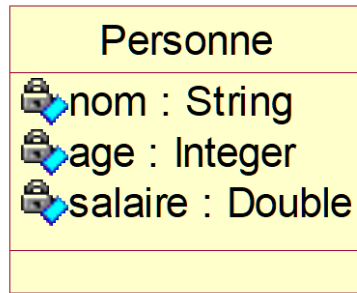
```
monChien.aboyer();
```

```
monChien.manger();
```

```
monChien.dormir();
```

DIFFERENTES INSTANCES ET LEUR UTILISATION

RÉSUMÉ CLASSE / OBJET



UML



```

class Personne{
    String nom;
    int age;
    double salaire;
}
  
```

CLASSE



```

Personne jean, pierre;
jean = new Personne ();
pierre = new Personne ();
  
```

INSTANCIATION

MULTIPLE INSTANCE

Un objet n'est ni plus ni moins que la création d'un type de variable, tout comme **String** qui est par essence un objet également.

A partir de cela, rien n'empêche de créer autant de variable que l'on veut du type de notre classe. Tout comme on peut créer plusieurs String, on peut créer plusieurs instances différentes, et chacune d'entre elles sera **indépendante**.

```
Chien monChien = new Chien("Berger Allemand", "Mâle");  
Chien unAutreChien = new Chien("Labrador", "Femelle");  
Chien encoreUnAutreChien = new Chien("Caniche", "Femelle");
```

TABLEAUX / LISTE

Tout comme des types primitifs, des objets peuvent très bien être utiliser avec des listes ou des tableaux. Exactement de la même manière et avec la même utilisation.

```
Chien[] tabChiens = new Chien[3];  
List<Chien> listChiens = new ArrayList<>();
```

ENCAPSULATION

DÉFINITION

L'encapsulation est un principe qui consiste à protéger les données et avoir des méthodes de manipulation des attributs à la place.

Seule la classe doit avoir accès aux attributs, qui seront donc en private, et on doit créer des méthodes pour y accéder ou les modifier.

UTILITÉ

1. **Protection des données**: Permet de ne pas modifier les données par accident.
2. **Contrôle d'accès**: Permet d'imposer des règles sur la lecture et la modification des données.
3. **Abstraction**: Permet de cacher le détail d'implémentation
4. **Maintenabilité**: La modification de la classe n'a pas d'impact sur le reste du code.
5. **Réduction d'erreurs**: En limitant les accès non contrôlés on limite les possible erreur.
6. **Explicité**: Permet d'avoir des méthodes distinctes de modification et récupération.
7. **Meilleure organisation du code**: Tout se trouvant dans la même classe, cela permet une meilleure lisibilité du code.

EXEMPLE

Pour chaque attribut créé, qui sera en **private**, on crée généralement deux méthodes:

- **Getter** : une méthode *getNomAttribut* permettant de récupérer l'attribut pour de la lecture.
- **Setter** : une méthode *setNomAttribut* permettant de modifier l'attribut, et permettant d'y ajouter des règles de modification.

Cela est appelé des *accesseurs* et des *modificateurs*.

Le plus souvent on appellera ça des **GETTER** et **SETTER**.

```
public class Personne {

    private String nom;
    private int age;

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom.toLowerCase();
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

COMPOSITION D'OBJET

DÉFINITION

Fonctionnant sur l'idée que les objets, étant des types de variables, peuvent être formé d'autres objets.

Cela permet de:

- Déléguer une partie des responsabilités à une autre classe.
- Réutiliser les sous-objets indépendamment ou dans d'autres objets.
- Expliciter les concepts.

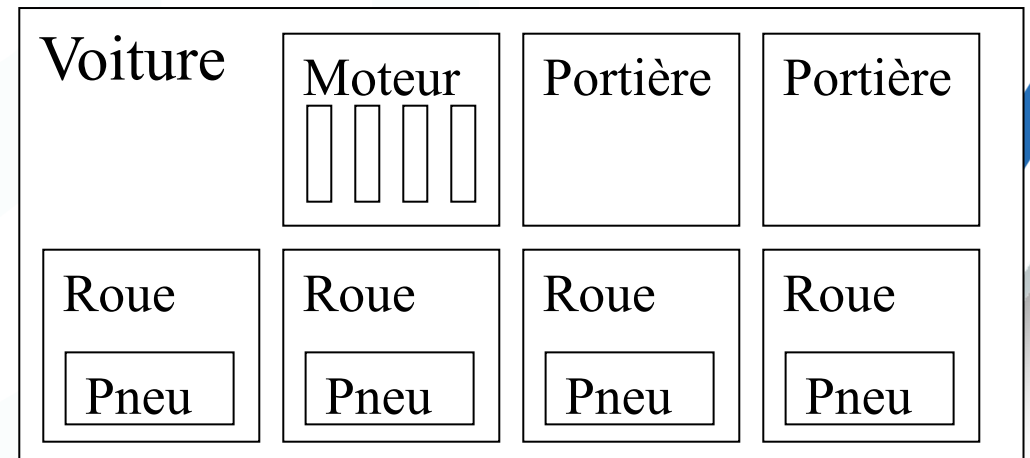
EXEMPLE

Imaginons une application devant gérer des voitures d'un point de vue mécaniques. Pour par exemple un garage automobile ou un jeu vidéo de voiture.

On doit donc gérer les différentes parties de la voiture, certains étant plusieurs fois le même concept (portes ou roues).

En dehors des attributs de type primitif, la voiture sera composée de:

- Moteur
 - Plusieurs cylindres
- 2 Portières
- 4 Roues
 - 1 Pneu



CODE

```
public class Voiture {
    private String marque;
    private String modele;
    private Moteur moteur;
    private Roue roueAVG;
    private Roue roueAVD;
    private Roue roueARG;
    private Roue roueARD;
    private Porte porteG;
    private Porte porteD;
}
```

```
public class Moteur {
    private int puissance;
    private Cylindre cylindre;
}
```

```
public class Cylindre {
    private String type;
}
```

```
public class Roue {
    private String taille;
    private Pneu pneu;
}
```

```
public class Pneu {
    private String marque;
    private int pression;
}
```

```
public class Porte {
    private String etat;
}
```

INSTANCIATION

```
public static void main(String[] args) {
    Pneu pneu1 = new Pneu("Michelin", 1);
    Roue roueAG = new Roue("17", pneu1);
    Pneu pneu2 = new Pneu("Michelin", 1);
    Roue roueAD = new Roue("17", pneu2);
    Pneu pneu3 = new Pneu("Michelin", 2);
    Roue roueAR = new Roue("17", pneu3);
    Pneu pneu4 = new Pneu("Michelin", 2);
    Roue roueAV = new Roue("17", pneu4);

    Cylindre cylindre = new Cylindre("V6");
    Moteur moteur = new Moteur(100, cylindre);

    Porte porteG = new Porte("GOOD");
    Porte porteD = new Porte("AVERAGE");

    Voiture voiture = new Voiture(
        "Peugeot",
        "308",
        moteur,
        roueAV,
        roueAD,
        roueAG,
        roueAR,
        porteG,
        porteD
    );
}
```

Pour créer la voiture, il faudra donc créer chaque objet la composant, et créer chaque objet les composant eux.

Un peu comme des poupées russes.

UTILITÉ

On peut donc voir qu'avec une série d'objet, on pourra définir des concepts très compliqués et étendus de manières détaillées et découpées.

Plus tard on pourra voir que les objets de notre application seront une représentation des tables SQL de notre BDD. D'où l'importance de la conception d'application, qui permettra de créer une conception de base de données, qui donnera donc les tables, puis permettra de créer les objets JAVA, de même sur le frontend.

Cela permettra donc d'avoir la même réflexion partout dans l'application.



toString

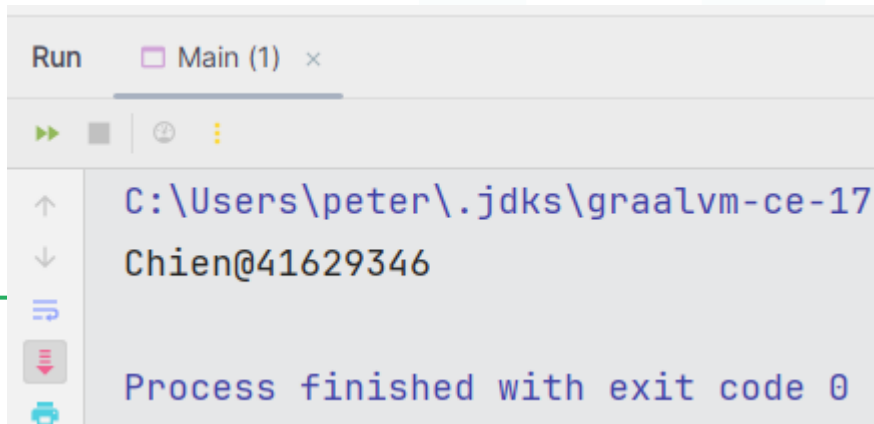
DÉFINITION

Dans chaque classe créée, JAVA crée une méthode (cachée) s'appelant *toString()*. Cela permet de « stringifié » la classe pour permettre son utilité en tant que texte.

Par exemple, cela permet de pouvoir afficher la classe dans une console avec un `System.out.print`

Lorsqu'on utilise cette fonction, on aura par contre en affichage une donnée étrange pour nous:

```
public static void main(String[] args) {  
    Chien monChien = new Chien("Berger allemand", "Mâle");  
    System.out.println(monChien);  
}
```



```
Run  Main (1) x  
C:\Users\peter\.jdk\graalvm-ce-17  
Chien@41629346  
Process finished with exit code 0
```

Ce code correspond au nom de la classe suivi du hashage de l'adresse mémoire où est enregistré l'instance de l'objet dans la mémoire du programme.

Donc ni explicite, ni utile.

REDÉFINITION

Par défaut quand on utilise le toString() JAVA affiche toujours un code (nom classe + adresse mémoire). Pourquoi ? Simplement parce que JAVA ne sait pas ce qu'on veut afficher, et donc il faut lui définir.

Pour cela, on peut réécrire la fonction pour l'overrider:

```
public class Chien {  
    private String race;  
    private String genre;  
  
    public Chien(String race, String genre) {  
        this.race = race;  
        this.genre = genre;  
    }  
  
    @Override  
    public String toString() {  
        return "Chien: race='" + this.race + "', genre='" + this.genre + "'.";  
    }  
}
```

```
Chien: race='Berger allemand', genre='Mâle'.
```

HERITAGE

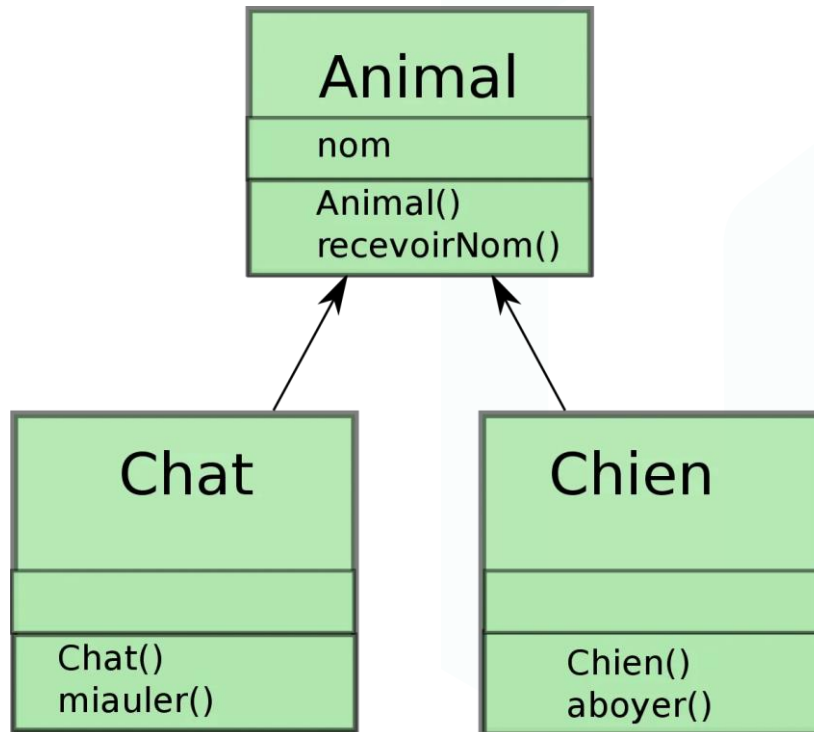
DÉFINITION

L'héritage est le principe de créer une classe qui va bénéficier des attributs et méthodes d'une autre. Cela permet de généraliser un contexte pour le réutiliser dans d'autres.

Utile pour:

- Organiser le code
- Regrouper un type de fonctionnalité
- Facilité à ajouter de nouvelles fonctionnalités
- Polymorphisme: permet d'utiliser des classes différentes en les castant dans leur classe parente

UML



La classe ANIMAL définit un animal de manière générique, qui va donc concerner tous les animaux.

On pourra, en reprenant les attributs et méthodes, créer d'autres classes d'animaux mais plus précis, ceux-ci pouvant avoir leurs propres attributs et méthodes.

Un CHIEN est un ANIMAL qui peut également aboyer().

Un CHAT est un ANIMAL qui peut également miauler().

SYNTAXE

Pour définir une classe comme héritant d'une autre, il faudra ajouter le terme **extends** après le nom de celle-ci, puis ajouter le nom de la classe parent derrière.

```
public class MaClasseEnfant extends MaClasseParent
```


CODE

```
public class Animal {  
  
    private String nom;  
  
    public void recevoirNom(String nom) {  
        this.nom = nom;  
    }  
  
}
```

```
public class Chat extends Animal {  
  
    public Chat() {  
    }  
  
    public void miauler() {  
        System.out.println("Miaou miaou !");  
    }  
  
}
```

```
public class Chien extends Animal {  
  
    public Chien() {  
    }  
  
    public void aboyer() {  
        System.out.println("Wouaf wouaf !");  
    }  
  
}
```



POJO

DÉFINITION

```
public class Person {

    private String prenom;
    private String nom;

    public Person(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    @Override
    public String toString() {
        return "Person{" +
            "prenom='" + prenom + '\'' +
            ", nom='" + nom + '\'' +
            '}';
    }

}
```

POJO ou **P**lain **O**ld **J**ava **O**bject (ou bon vieil objet java) est une classe JAVA la plus simple possible.

Celle-ci contiendra uniquement:

- Attributs
- Constructeurs
- Getter / Setter
- toString

Cela permet une bonne simplicité et une meilleure lisibilité. Ainsi que de permettre de séparer le modèle (ou moule) et l'exécution métier de l'application.