Lycée Aristide Bergès

POO

Notions de programmation orientée objet

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision
2019-03-20	Création du cours

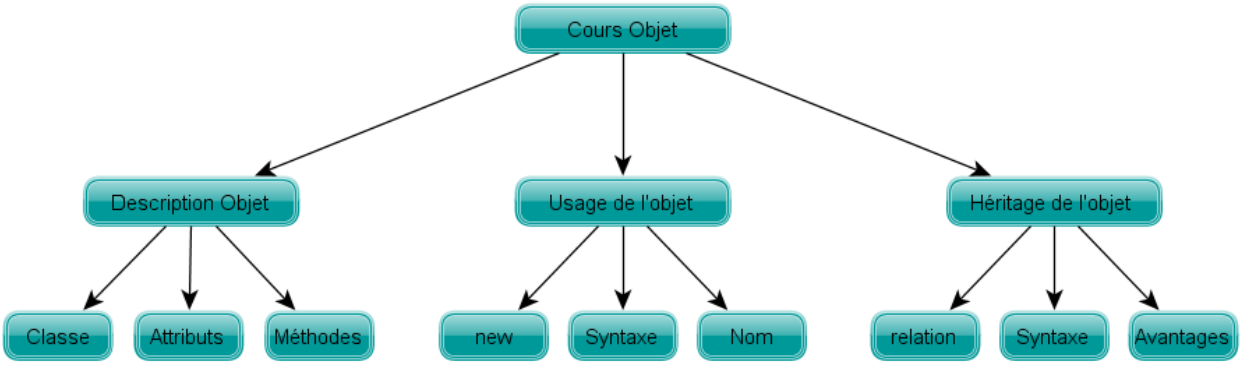
Sommaire

A Présentation

A.1 Plan du cours

Ce cours aborde les premières notions d'objet et de programmation objet. Il s'agit d'un concept basé sur la programmation procédurale, mais poussé à son maximum.

Le plan est le suivant :



Vous apprendrez donc, qu'est-ce qu'un objet et comment le décrire.

Vous verrez ensuite comment l'utiliser dans un programme.

Vous découvrirez enfin ce que l'héritage apporte à la conception orientée objet.

A.2 Objectifs du cours

À la fin de ce cours vous serez capable de manipuler les concepts de base de la programmation orientée objet.

- Savoir reconnaître un objet ou une classe
- Savoir décrire un objet ou une classe
- Savoir utiliser un objet ou une classe
- Comprendre la notion d'héritage

C'est parti !

B Introduction

Avant de parler programmation, il est utile de parler des objets, en tant qu'éléments faisant partie de notre vie courante.

B.1 Description d'une lampe Schmõlblück

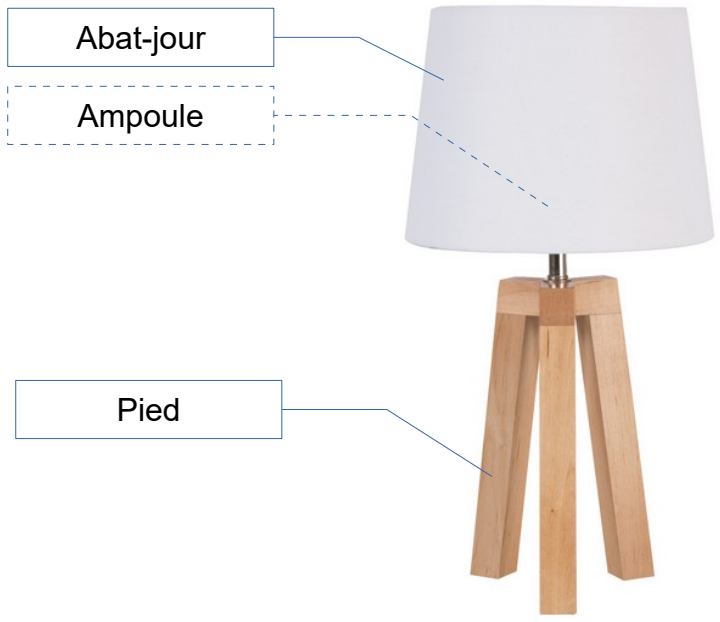
B.1.1 Objet réel

Prenons l'exemple d'une lampe d'un fabriquant suédois, dont le concept est la modularité. Pour avoir une lampe complète, le client peut acheter plusieurs éléments différents :

- Il existe 2 pieds de lampe différents : 20 cm ou 40 cm
- Il y a 3 puissances d'ampoule : 40 w, 60 w et 75 w
- Il est possible de choisir 4 coloris d'abat-jour : blanc, noir, jaune et Polnergha (célèbre couleur)

Cependant, quels que soient les choix des utilisateurs, **la notice de montage reste la même ainsi que le prix**.

Nous allons rédiger les caractéristiques de cette lampe :



SCHMOLBUCK
–
abat-jour : blanc, noir, jaune ou Polnergha

ampoule : 40, 60 ou 75 watts

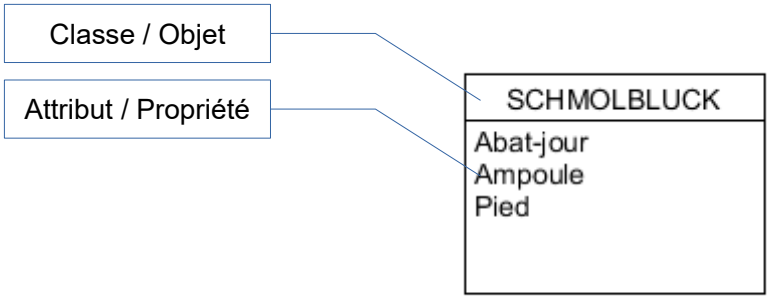
pied : 20 ou 40 cm

On peut ainsi dire que la lampe à plusieurs attributs (abat-jour, ampoule et pied) ayant chacun plusieurs valeurs mais que ce qui va caractériser cette lampe, c'est l'ensemble de ces attributs.

Pour commander une lampe Schmolbluck, il faut donc remplir les valeurs des 3 attributs : toutes les lampes n'auront donc pas la même couleur, le même pied ou la même puissance d'éclairage.

B.1.2 Objet informatique

Pour décrire la lampe, il existe un langage de conception graphique en informatique, UML. Nous verrons plus tard tout ce qu'il peut décrire mais voici comment modéliser notre lampe :



Il s'agit d'un cadre contenant le nom générique de l'objet, une séparation, et les propriétés (attributs) de l'objet.

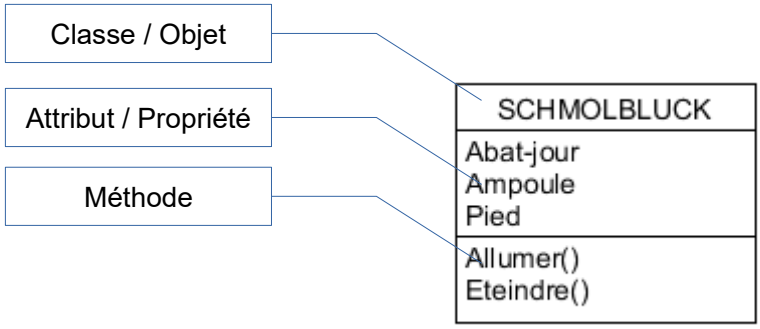
Ici, on parle désormais de classe. Une classe d'objet doit vous rappeler les films sur les sous-marins : la description est souvent « il s'agit d'un sous-marin de classe Typhon… » qui signifie que cette classe dispose de ces propres attributs.

Dans notre exemple, « **il s'agit d'une lampe de classe Schmolbluck** »

Pour le moment, ce n'est que le plan de montage de la lampe. Il manque à ce modèle, les actions possibles. Pour une lampe, il s'agit de ce que l'utilisateur pourra faire avec :

- Allumer la lampe
- Éteindre la lampe

En informatique, on appelle ces actions, des méthodes :



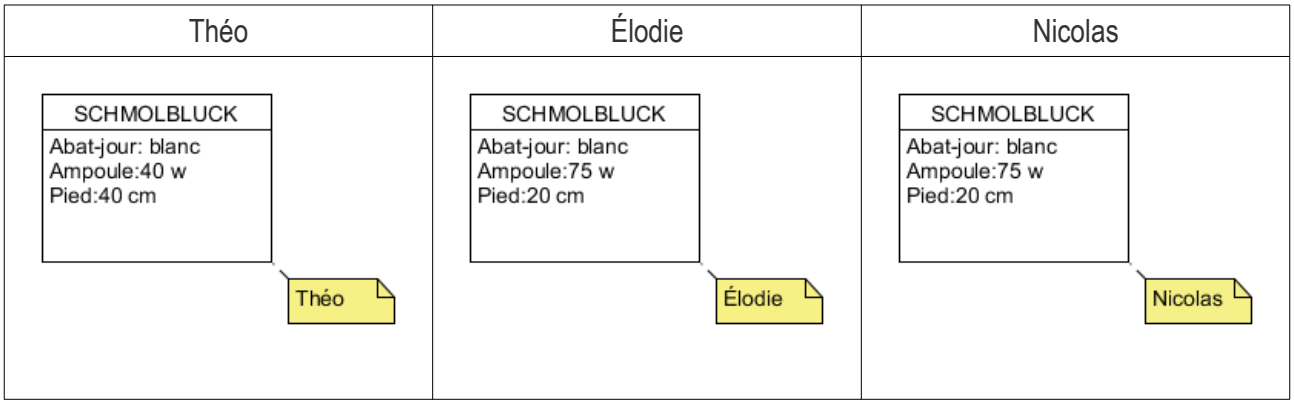
Notre modèle informatique est désormais complet !

En résumé :Un objet réel est décrit par une classe qui contient des attributs et des méthodes. Les méthodes sont des actions qui peuvent modifier la valeur d'un attribut. En programmation, une méthode est une fonction ou une procédure.

B.2 Usage de l'objet

B.2.1 Objet réel

Trois acheteurs vont maintenant sortir du magasin avec chacun une lampe :

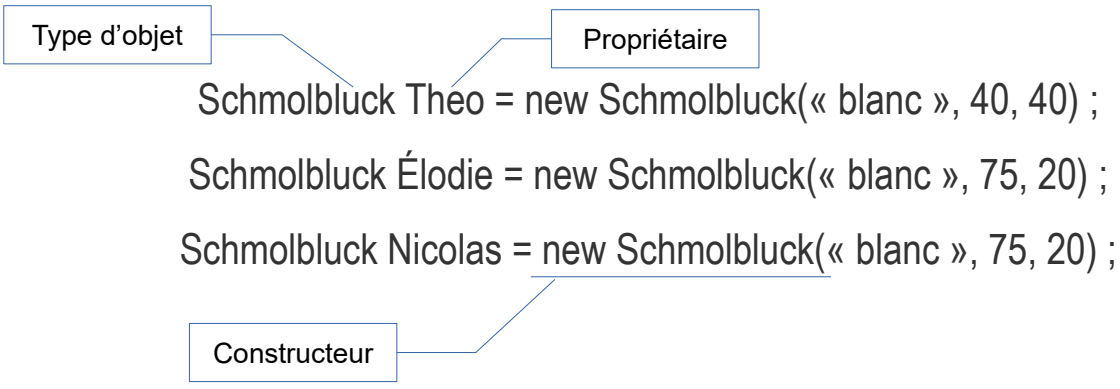


Si vous observez les trois achats, vous pouvez voir qu'Élodie et Nicolas ont acheté la « même » lampe mais évidemment, il s'agit bien de trois objets différents, qui utilisent le même modèle de lampe. Ils ont **créé** leur propre lampe et ont quitté le magasin, chacun avec sa propre lampe.

- Théo a construit une lampe Schmolbluck avec abat-jour blanc, ampoule 40 watts et grand pied
- Élodie a construit une lampe Schmolbluck avec abat-jour blanc, ampoule 75 watts et petit pied
- Nicolas a construit une lampe Schmolbluck avec abat-jour blanc, ampoule 75 watts et petit pied

B.2.2 Objet informatique

Nous pouvons faire la même démarche en informatique, en respectant la syntaxe suivante :



Le mot clé « new » signifie construire mais en langage orienté objet, on utilise souvent le terme **instancier**.

Le constructeur est en réalité le nom de la classe après le mot « new » (il n'est pas possible d'utiliser autre chose qu'un constructeur après le mot « new »).

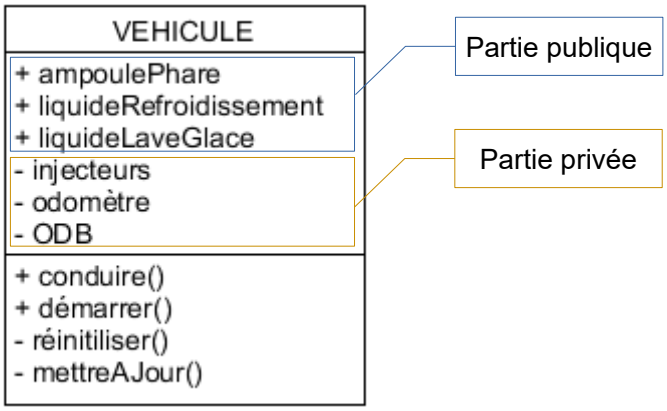
B.3 Modification de l'objet

B.3.1 Objet réel

Un objet n'est généralement pas statique, nous pouvons interagir avec lui. Cependant, nous ne pouvons pas modifier certaines parties des objets. Prenons cette fois comme exemple les voitures :

- Vous pouvez conduire votre voiture, laver celle-ci ou refaire les niveaux de lave-glace, d'huile ou de liquide de refroidissement…
- Le garagiste dispose d'un accès direct aux éléments techniques : lire des anomalies de l'ordinateur de bord, mettre à jour le firmware, changer les injecteurs, etc.

On pourrait décrire l'objet « voiture » en UML comme ceci :



B.3.2 Objet informatique

En informatique, la notion d'objet fait appel à la notion **d'encapsulation** : cela signifie que les attributs et les méthodes à l'intérieur d'une classe peuvent êtres invisibles depuis l'extérieur de celle-ci.

Dans l'exemple précédent, les attributs publics sont visibles et modifiables depuis l'extérieur.

```
Vehicule maVoiture = new Vehicule() ;
maVoiture.Conduire() ;
maVoiture.liquideRefroidissement = 5 ; // en litres
```

Il sera toutefois impossible d'utiliser les attributs et méthodes suivantes.

```
maVoiture.mettreAJour() ;
maVoiture.odomètre = 51000 ; // en km
```

L'**encapsulation** assure la protection de certaines parties du code.

B.4 Ce qu'il faut retenir

Une **classe** décrit les **membres** d'un objet : les **attributs** et les **méthodes**.

Un objet n'existe concrètement, qu'à partir du moment où on l'**instancie**.

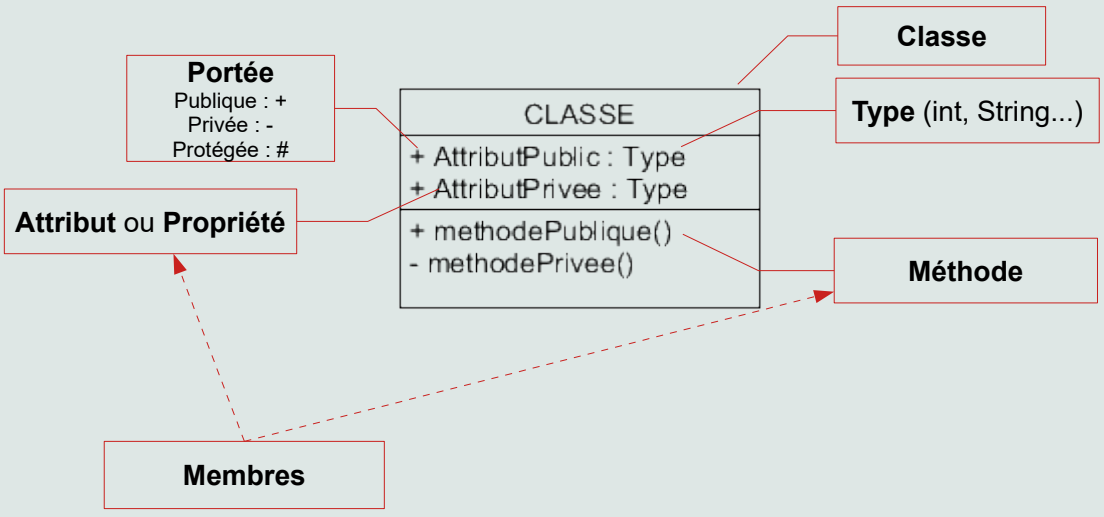
La création d'un objet renvoie une valeur qui est stockée dans un variable qui sert d'étiquette et qui est de type de la classe.

```
Voiture monAuto = new Voiture() ;
```

Le **constructeur** est une méthode qui permet de créer l'objet : cette méthode est créée automatiquement pour chaque classe, mais pourra être définie par le programmeur. L'important c'est de comprendre qu'il **existe toujours un constructeur**.

Les méthodes et les attributs à l'intérieur d'une classe ou d'un objet peuvent être protégés et rendus inaccessibles en dehors de la classe. On appelle cela l'**encapsulation**. Pour gérer cela, on utilise une indication de **portée** : **publique** (+), **privée** (-) ou **protégée** (#).

En informatique, il existe un langage pour décrire graphiquement les classes de programme, c'est **UML** ! Vous devez être capable de lire les diagrammes de classe UML.



C Utilisation des classes

Les classes et les objets sont intéressants en programmation, car ils permettent de développer un code plus sécurisé, plus fiable et moins énergivore. Comment ?

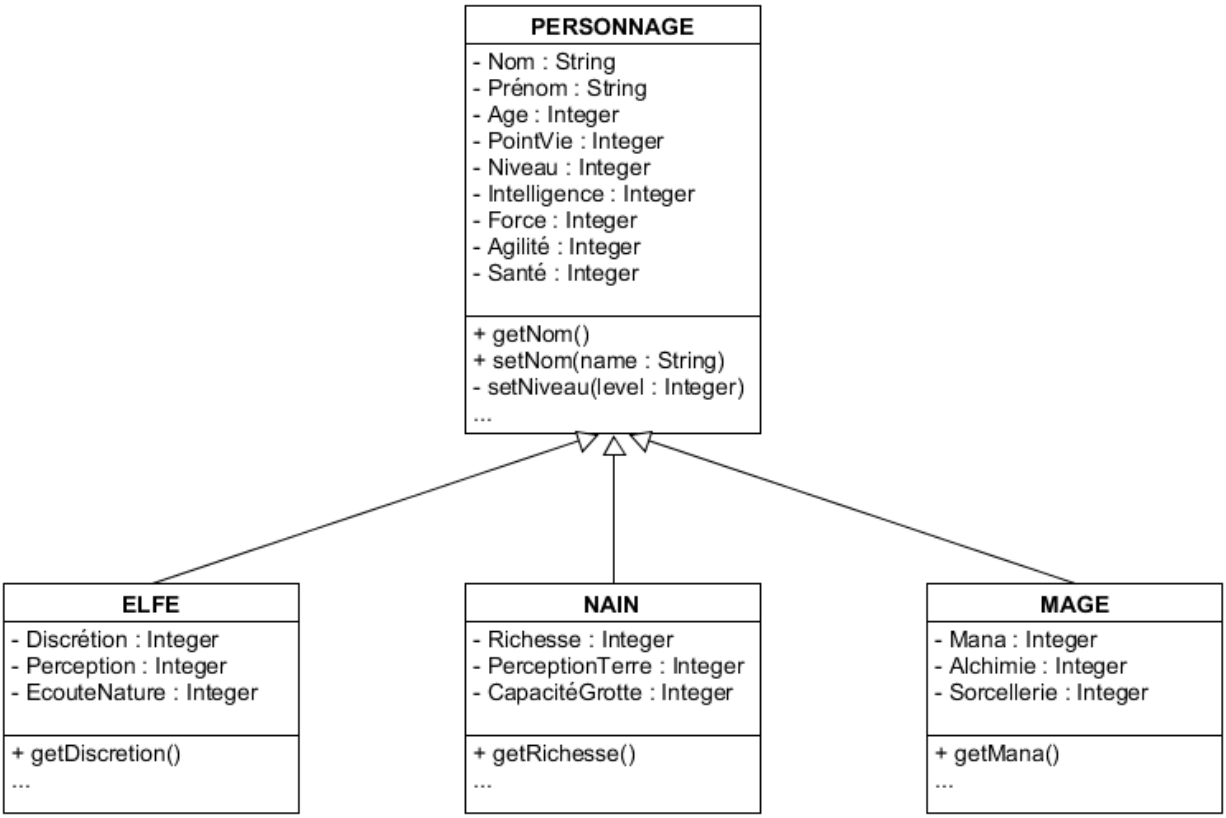
C.1 Héritage

La force de la programmation réside dans l'**héritage**. Elle permet principalement de réutiliser un code existant et d'y ajouter des fonctions sans le modifier. Prenons un exemple simple : dans un jeu de rôle, les joueurs ont une fiche de personnage relativement commune. Ils vont cependant personnaliser leur personnage en ajoutant des croquis, des caractéristiques nouvelles, etc.

C.1.1 Exemple : feuille de jeu de rôle

Plutôt que de devoir créer une feuille exhaustive de toutes les caractéristiques des personnages, l'héritage nous permet de créer une feuille commune puis d'ajouter les spécificités sur une autre feuille.

Avec les symboles UML, voici ce que cela donne :



Ainsi, la fiche de personnage peut être fournie par l'éditeur du jeu de rôle, mais le maître de jeu peut créer ses propres créatures, sans devoir recréer les attributs de la feuille de personnage.

C.1.2 Code informatique (Java)

Pour représenter les 4 classes et les liens, voici un extrait de ce qu'un codeur peut écrire :

```java
public class Personnage {
    protected String nom;
    protected String prenom;
    protected Integer age;
    protected Integer niveau;
    // ...
    public String getNom() {
        return this.nom;
    }
    public void SetNom(String nom) {
        this.nom = nom;
    }
    // ...
}

public class Elfe extends Personnage {
    protected Integer Discretion;
    // ...
}

public class RPG_example {
    public static void main(String[] args) {
        // TODO code application logic here
        Personnage PNJ = new Personnage();
        Elfe Legolas   = new Elfe();
        Legolas.SetNom("Legolas");
    }
}
```

La classe Personnage

Un **accesseur**

Un **mutateur**

Héritage (Java)

Le début du programme...

Création d'un personnage

Création d'un elfe

Changement du nom de l'Efle

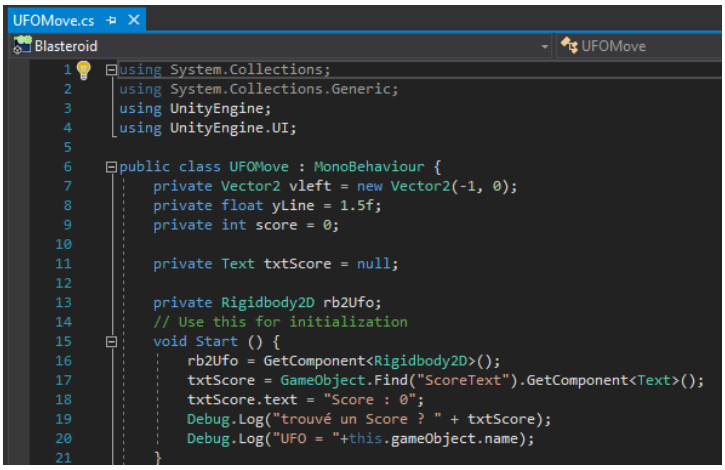
En C#, l'héritage est représenté par le symbole « : », la classe Elfe s'écrirait :

```java
public class Elfe : Personnage {
    protected Int16 Discretion;
    // ...
}
```

Comme vous pouvez le Voir, peu de différences entre Java et C#

L'usage de l'héritage est très courant dans les langages orientés objets, notamment, pour la création et l'utilisation de fenêtre dans les applications graphiques ! Unity3D utilise également l'héritage pour gérer vos objets avec ses méthodes !

Regardez la classe UFOMove ci-contre, qui hérite de la classe MonoBehavior…



C.2 Constructeur

Il reste à éclaircir le mystère du constructeur, cette méthode qui n'existe pas, mais qui existe lorsqu'on instancie une classe.

En réalité, le compilateur créé la méthode automatiquement : elle n'a qu'un rôle de créer les attributs dans la mémoire de l'ordinateur, mais ceux-ci ont tous une valeur nulle ou vide (selon leurs types).

Nous pouvons heureusement créer nos propres constructeurs, ce qui rendra l'usage de notre classe plus conviviale.

Reprenons l'exemple de la classe Personnage : la création d'un personnage sans nom et sans avec ses attributs vides reste pratique, mais nous pouvons créer un constructeur qui acceptera un nom et un âge en paramètre. Il s'agit d'une méthode qui doit obligatoirement porter le nom de la classe (en PHP, le constructeur s'appelera *__constructor()*)

```
public class Personnage {
    protected String nom;
    protected String prenom;
    protected Integer age;
    protected Integer niveau;
    // ...
    public Personnage(String nom, Integer age) {
        this.nom = nom;
        this.age = age;
    }
    public void SetNom(String nom) {
        this.nom = nom;
    }
    // ...
}
```

La classe Personnage

Un **constructeur**

*Nous parlerons de **this** plus tard*

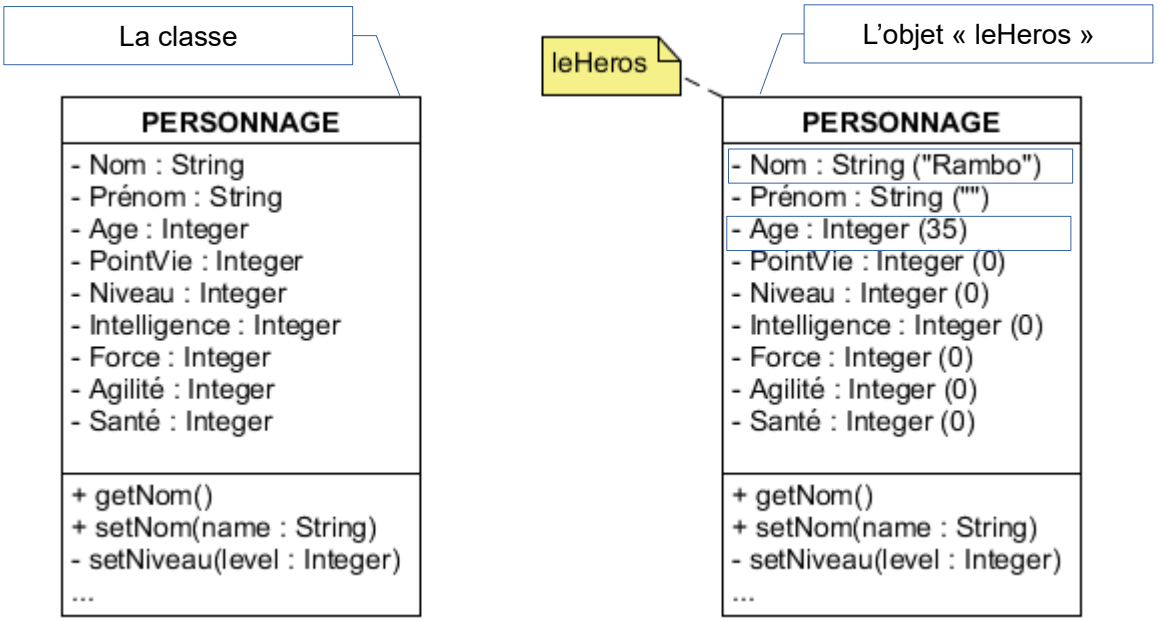
Ici, le constructeur acceptera d'être appelé avec deux paramètres, le nom et l'âge. Cela n'empêchera pas le compilateur de créer un constructeur vide, les deux lignes ci-dessous ne sont donc pas contradictoires.

```
    Personnage unPNJ   = new Personnage();
    Personnage leHeros = new Personnage("Rambo", 35);
```

Pour information, les attributs unPNJ et leHeros ne sont que des « étiquettes » ou des labels. Ils ne contiennent pas l'objet directement mais indique où il se trouve. Ce sont des pointeurs : ils fournissent au compilateur l'adresse de la zone mémoire où est stocké l'objet.

Nous allons maintenant nous intéresser à l'objet « leHeros » dont le nom est modifié par une ligne bizarre. Pour le moment, considérez que **this contient la même adresse que l'étiquette de l'objet**. La commande **this.nom = nom** signifie que la chaîne nom de l'objet doit être remplacée par la chaîne nom passée en paramètre du constructeur. L'autre utilité de this, est d'éviter la confusion entre nom… et nom !

Voici le résultat de notre constructeur :



Il est possible de créer autant de constructeur que l'on veut mais le nombre de paramètres doit être différents ou bien de type différents (Integer, String, Float...)

C.3 Portée

Nous avons vu que les attributs et méthodes d'une classe sont protégés par l'encapsulation : seules les membres publics sont visibles hors de la classe. Voici maintenant le détail des 3 portées.

- **Publique** : les membres peuvent être accédés depuis n'importe quelle partie du code qui utilise cette classe. La plupart du temps, ce sont les méthodes qui sont publiques. Pour modifier les attributs privés, on crée des méthodes qui peuvent lire les attributs (la méthode appartenant à la classe, elle peut accéder à ses attributs) ou les modifier : ce sont les **accesseurs** (en anglais, *getter*) et **mutateurs** (en anglais, *setter*)
- **Privée** : les membres sont inaccessibles hors de la classe, ils sont donc invisibles. C'est généralement le cas des attributs et des méthodes internes à la classe.
- **Protégée** : les membres sont accessibles uniquement à la classe et aux classes filles. Cette partie introduit la notion d'héritage (juste en dessous), mais pour résumer, les membres de portée « protégée » sont accessibles à toutes les classes qui hérite de la classe qui contient ces membres.

C.4 Ce qu'il faut retenir

L'**héritage** est une notion capitale en POO : c'est l**a capacité d'une classe à hériter des membres d'une autre classe**. En Java et C#, on ne peut hériter que d'une seule classe.

La **portée** est également un élément de sécurité et il existe trois niveaux :

public	protégé	privé

Les méthodes publiques permettant de lire des attributs sont appelées « **accesseur** ». En anglais, g**etter**.

Les méthodes publiques permettant de modifier des attributs sont appelées « **mutateur** », En anglais, **setter**.

Par convention de nom, elles commencent souvent par get… ou set…

Le **constructeur** est une méthode qui **porte le nom de la classe** et qui ne renvoie jamais de valeur.