

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W11-03-publicprivate-JAVA-pt3

Concepts (extraits des sous-titres générés automatiquement) :

Référence this. Exemple sethauteur. Mémoire des objets. Notion de masquage. Méthode sethauteur. Notion de programmation. Programme de façon indépendante. Hauteur du rectangle. Partie interface. Concepteur de la classe rectangle. Petit complément technique. Nom de l'attribut hauteur. Situation de masquage this. Petits programmes. Nom x.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en Java).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>



public **et** private

(Partie 3)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
    public double largeur;
    public double hauteur;
    public String label;
}
```



Une autre remarque qui revient souvent de la part des étudiants c'est pourquoi diable faire tout ceci, c'est bien compliqué, c'est fastidieux

notes

résumé

0m 1s



Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
    public double largeur;
    public double hauteur;
    public String label;
}
```

mais dans ce cas ...

```
Rectangle rect = new Rectangle();
rect.hauteur = -36;
System.out.print(rect.label.length());
```



d'écrire tout cela, pourquoi pas simplement laisser en public, la largeur, la hauteur, et tous les attributs, comme cela on pourrait les manipuler comme on veut. C'est justement là le problème, on pourrait les manipuler comme on veut et par exemple dans ce cas, donc si on déclare une instance rect1 de la classe rectangle, on pourrait tout à fait affecter la valeur -36 à la hauteur du rectangle. Alors que si au lieu de passer comme ça directement avec un accès à l'attribut, on passait par une méthode, par exemple setHauteur, alors on pourrait dans la méthode setHauteur avoir un mécanisme de vérification, par exemple si h est négatif ou nul même, par ex. avoir une gestion de l'erreur ici, à détailler, et puis sinon effectivement à ce moment là affecter la hauteur qui a été vérifiée et qui est valide. Ce passage par une méthode permet donc un mécanisme de vérification et de garantie de l'intégrité des données. Supposons aussi que l'on dépende aussi d'un attribut qui est un objet plus complexe, ici simplement une String, par exemple un label qui ait une représentation interne d'une certaine information et qu'on utilise, donc on peut y accéder en externe, donc on écrirait rect.label.length en utilisant donc les propriétés du fait que c'est une String et supposons qu'on écrive par exemple 300-400 lignes comme ça en utilisant directement le fait que ce soit une String et puis le concepteur de la classe rectangle décide de changer String par exemple une arrayList, résultat on doit jeter nos 300-400 lignes à la poubelle et puis tout réécrire, alors que si on n'utilise que l'interface, laquelle interface ne doit pas changer et que l'on n'accède pas directement comme ça à des champs mais que l'on passe de l'interface, à ce moment là la responsabilité de tout changer par exemple String en arrayList

notes

résumé

0m 10s



Fastidieux d'écrire des « Accesseurs »/« manipulateurs » alors que l'on pourrait tout laisser en **public** ?

```
class Rectangle
{
    public double largeur;
    public double hauteur;
    public String label;
}
```

mais dans ce cas ...

```
Rectangle rect = new Rectangle();
rect.hauteur = -36;
System.out.print(rect.label.length());
```



incombera au développeur de la classe

notes

résumé

Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

Situation typique en POO : un paramètre cache un attribut

```
void setHauteur(double hauteur) {
    hauteur = hauteur; // Hmm.... pas terrible !
}
```

mais nos 400 lignes resteront par contre inchangées puisqu'elle respectent l'interface de la classe. C'est pour ces raisons fondamentales, pouvoir vérifier les valeurs que l'on affecte aux différents champs et pouvoir modifier un programme de façon indépendante, ce qu'on appelle la modularisation, qu'on recommande de passer par des méthodes et que l'on recommande de ne mettre aucun attribut dans la partie interface, dans la partie publique. Il est évident que sur des petits programmes que l'on fait en exercices ça ne prend pas beaucoup de sens, mais dans le cadre de grands projets logiciel c'est justement dans ce cadre là que la notion de programmation orientée objet a été développée cela prend tout son sens, et donc dans cet esprit de bonne programmation et d'aller vers de gros programmes respectez donc ces consignes et essayez de séparer au maximum donc ce qui est l'interface de ce qui est les détails d'implémentation. Un autre petit complément technique pour finir, la notion de masquage. Masquage à lieu quand un nom utilisé par un certain objet cache un nom pour désigner un autre objet, et c'est quelque chose qui en programmation orientée objet est typique lorsqu'un paramètre, par exemple ici le paramètre que l'on aurait appelé hauteur

notes

résumé

2m 13s



Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est une **référence à l'instance courante**

`this` \simeq « moi »

Syntaxe pour spécifier un attribut *en cas d'ambiguïté* :

`this.nomAttribut`

Exemple :

```
void setHauteur(double hauteur) {  
    this.hauteur = hauteur; // Ah, là ça marche !  
}
```

L'utilisation de `this` est obligatoire dans les situations de **masquage** (mais évitez ces situations !)

pour la méthode `setHauteur`, cache le nom de l'attribut `hauteur`. Dans ce cadre là on voudrait désigner par cette hauteur là l'attribut et puis par cette hauteur ci bien sur la hauteur qu'on a reçu comme paramètre. Mais dans ce contexte là le compilateur ne saura pas faire la différence et donc effectivement ce genre d'écriture n'est pas terrible, donc que faire dans ce cas là ? Et bien, soit tout simplement et c'est ce que je vous recommande, utiliser des noms qui sont différents, donc comme on avait fait tout à l'heure, on avait `h` pour le nom de paramètre, là il n'y a aucune ambiguïté, soit sinon on peut utiliser ce que l'on appelle une technique de démasquage en utilisant donc la référence `this` qui est donc une référence sur l'instance courante, donc vous pouvez si vous voulez traduire donc `this` à peu près par moi, moi l'instance courante, donc dans l'exemple précédent ça nous donnerait, cette hauteur où on a gardé ici le nom `hauteur` pour le paramètre qu'on veut passer à la méthode `setHauteur`, et ici il faut démasquer le nom `hauteur` qui désigne l'attribut en disant `ma hauteur` si vous voulez, la hauteur à moi, donc `this.hauteur` pour désigner ma hauteur.

notes

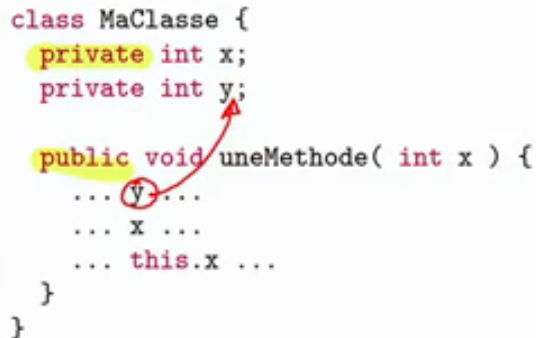
résumé

3m 25s



La portée des attributs dans la définition des méthodes est résumée par le schéma suivant :

```
class MaClasse {  
    private int x;  
    private int y;  
    public void uneMethode( int x ) {  
        ... y ...  
        ... x ...  
        ... this.x ...  
    }  
}
```



Donc dans ces situation de masquage this est absolument obligatoire, mais je vous recommande d'éviter ces situations et comme je disais, donc de choisir des noms qui évitent toute ambiguïté, qui soient clairs pour tout le monde mais qui évitent tout ambiguïté. Résumons ce que nous avons vu concernant les droits d'accès ainsi que les portées des attributs et des méthodes d'une classe. Pour les droits d'accès nous avons vu qu'il y a les mots clés private et public. Private permet de définir les parties détails d'implémentation, les parties privées, les parties accessibles uniquement depuis l'intérieur de la classe, alors que l'aspect public permet de définir ce que l'on appelle l'interface donc les parties auxquelles on peut accéder depuis l'extérieur de la classe, depuis le main, depuis d'autres classes. La portée donc des attributs et des méthodes est limitée à toute la classe, donc par exemple ici x et y sont des attributs qui sont accessibles n'importe où dans la classe, et en particulier à n'importe quelle méthode de cette classe, donc par exemple ici ce nom y réfère l'attribut de l'instance auquel on appliquerait une méthode ; on peut donc aussi avoir des phénomènes de masquage lorsque l'on désigne par un nom ici qui est le même nom qu'un attribut,

notes

résumé

4m 37s





lorsqu'on désigne un paramètre d'une méthode avec un tel nom, alors à ce moment là et bien localement la résolution de portée va faire que `x` dans cette méthode ici référera le paramètre en question. Alors que si l'on veut parler de l'attribut `x`, il faut utiliser donc la référence `this` pour démasquer le nom `x` et donc `this.x` ici réfère à l'attribut `x` de l'instance dans laquelle on appliquerait la méthode.

notes

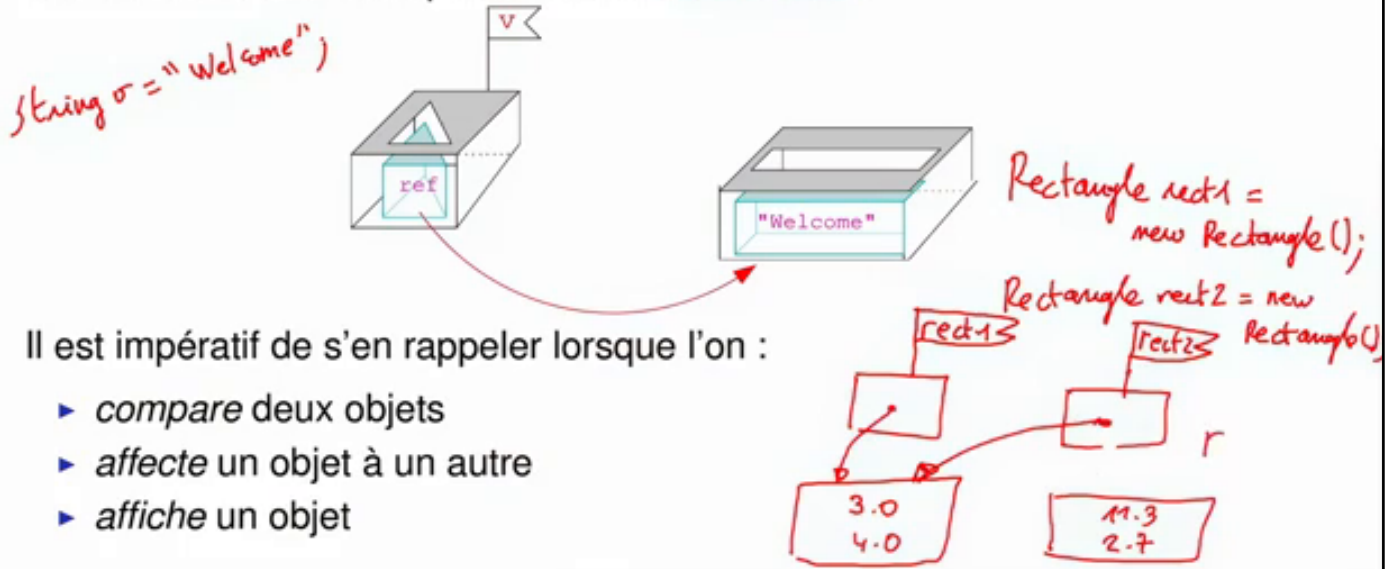
résumé

5m 49s



Objets en mémoire

Attention : Comme nous l'avons déjà vu pour la classe prédéfinie `String` (et pour les tableaux), les objets, contrairement aux entités de types élémentaires, sont manipulés via des **références** :



Terminons cette séquence par un aspect un peu plus technique de la représentation en mémoire des objets ; aspect technique mais auquel il faut néanmoins prêter attention lorsque l'on écrit du code. Je vous rappelle comme nous l'avons vu dans le premier cours d'introduction à la programmation en java que les objets comme les Strings par exemple, les tableaux, sont représentés en mémoire au travers de références. Par exemple si je déclare une String `v` initialisée avec "Welcome", alors en mémoire nous avons le schéma suivant, `v` est en fait une référence qui réfère un endroit mémoire où la valeur "Welcome" est stockée. De la même façon si je déclare 2 objets `rectangle`, `rect1` et `rect2`, alors en mémoire j'ai également des références vers des zones mémoire qui contiennent effectivement donc la hauteur, admettons par exemple qu'à `rect1` on ait ensuite affecté les valeurs 3 et 4 à la hauteur et à sa largeur, et puis `rect2`, donc par exemple on avait une hauteur 11.3 et puis 2.7. Donc voilà le schéma que l'on aurait en mémoire, ces objets sont en fait des références sur les endroits où sont vraiment stockés les valeurs, et ça c'est vraiment très important lorsqu'on veut comparer des objets parce que : est-ce qu'on va comparer des références ou comparer effectivement des valeurs, à ce stade là si on ne fait rien de particulier on va comparer les références. Lorsqu'on affecte un objet à un autre, on va faire quelque chose comme ceci,

notes

résumé

6m 25s



La constante prédéfinie `null` peut être affectée à n'importe quel objet de n'importe quelle classe.

Affectée à une variable, elle indique que celle-ci ne référence aucun objet :

```
Rectangle rect = null; // la variable rect
                      // ne référence aucun objet
```

Avant de commencer à utiliser un objet, il est souvent judicieux de tester s'il existe vraiment, ce qui peut se faire par des tournures telles que :

```
if (rect == null) {...}
if (rect != null) {...}
```

si je fais `rect2=rect1`, et puis aussi lorsque donc on affiche un objet c'est pas la valeur mais la référence que l'on souhaitera affecter/afficher mais bien sûr il faudrait (en général plutôt) afficher le contenu. Nous reviendrons en détail sur ces aspects là plus tard dans le cours. Les objets étant des références ils peuvent aussi ne pas référencer d'objet, référencer aucun objet et ceci est indiqué donc avec la valeur prédéfinie le mot réservé `null`. Donc par exemple si je déclare ici une instance `rect` de la classe `Rectangle` et je dis qu'elle est `null`, c'est à dire qu'en fait c'est une référence vers nulle part, ce qui pourrait donc se représenter comme ceci, donc on pourra voir typiquement ce genre de code si `rect==null` à ce moment là faire quelque chose, donc par exemple l'affecter à un nouveau rectangle, ou aussi ce genre de tournure, si `rect` n'est pas `null` à ce moment là donc par exemple l'utiliser, par exemple appeler la méthode `surface`. par exemple appeler la méthode `surface`.

notes

résumé

8m 13s

