

Support de cours

Cours:

## Introduction à la programmation orientée objet (en Java)

Vidéo:

### W13-01-heritageintro-JAVA-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Niveau des méthodes. Personnages d'un jeu. Exemple des guerriers. Notions d'encapsulation. Classe personnage. Sous-classe c1. Méthodes communes. Durée de vie. Niveau d'énergie. Ensemble des classes. Classe de type personnage. Notion d'héritage. Séquence vidéo. Classe c. Super-classes.**



[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/>



# Héritage : concepts

## (Partie 1)

### Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Après les notions d'encapsulation et d'abstraction,

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

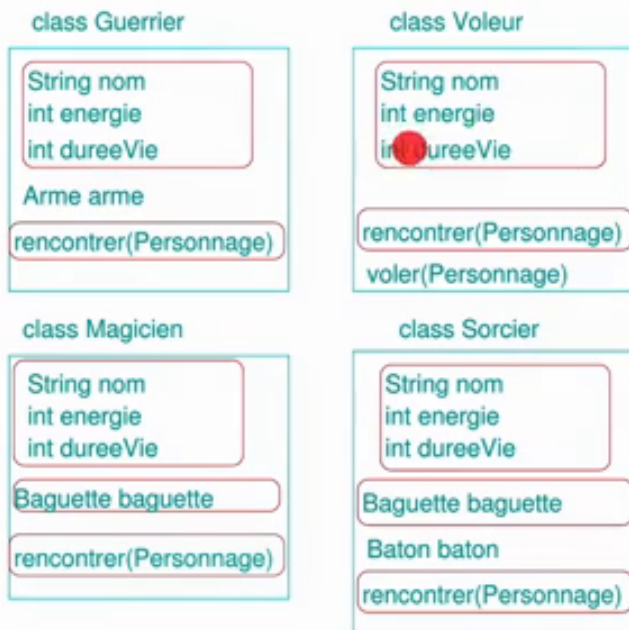
---

---

---

---

## Exemple : classes pour les personnages



## Exemple : classes pour les personnages

class Guerrier

String nom  
int energie  
int dureeVie

Arme arme

rencontrer(Personnage)

class Voleur

String nom  
int energie  
int dureeVie

rencontrer(Personnage)

voler(Personnage)

class Magicien

String nom  
int energie  
int dureeVie

Baguette baguette

rencontrer(Personnage)

class Sorcier

String nom  
int energie  
int dureeVie

Baguette baguette

Baton baton

rencontrer(Personnage)

et il pourrait aussi rencontrer d'autres personnages, voire les voler, Le magicien lui aussi aurait son nom, son énergie, sa durée de vie, il pourrait rencontrer d'autres personnages, et il aurait peut-être une baguette au lieu d'avoir une arme. On pourrait également avoir des sorciers, comme des sortes de magiciens, qui en plus d'avoir une baguette, auraient aussi un bâton. Tout ceci dupliquerait beaucoup de codes, ça nous ferait perdre beaucoup de temps d'écrire tout cela,

notes

résumé

1m 1s





et surtout poserait de gros problèmes de maintenance, imaginons que l'on ne veuille plus que les personnages aient une durée de vie, mais qu'ils aient par exemple des points de vie, ou que l'énergie ne soit plus un int mais un double, il faudrait alors corriger l'ensemble des classes que nous avons développées, c'est certainement pas une bonne façon de faire.

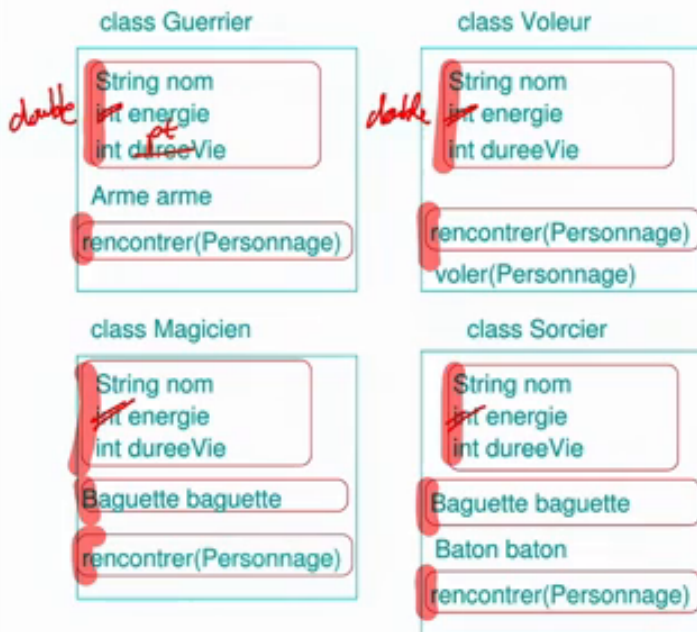
notes

résumé

1m 27s



## Exemple : classes pour les personnages



La notion d'héritage en programmation permet justement de répondre à ce genre de problèmes, en essayant de regrouper au maximum les parties communes, attributs et méthodes communes dans des classes qu'on va appeler des super-classes, qui regroupent des caractéristiques qui sont ensuite enrichies ou spécialisées pour d'autres classes plus spécifiques. Dans notre cas typiquement, ça serait une classe Personnage, un personnage générique qui d'ailleurs répondrait à un des problèmes de la conception présentée jusqu'ici,

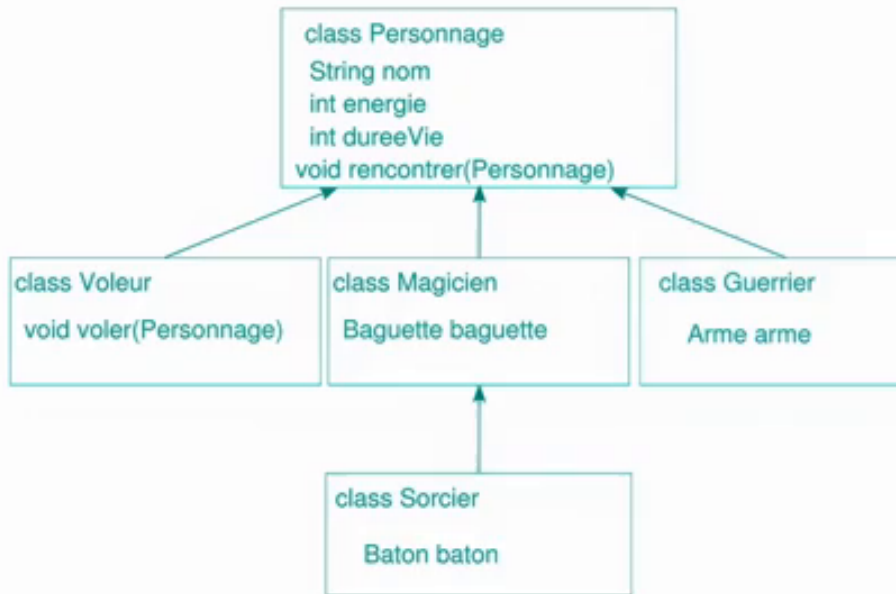
notes

résumé

1m 44s



## Exemple : héritage



c'est qu'on n'avait pas de classe de type Personnage.

notes

résumé

2m 13s

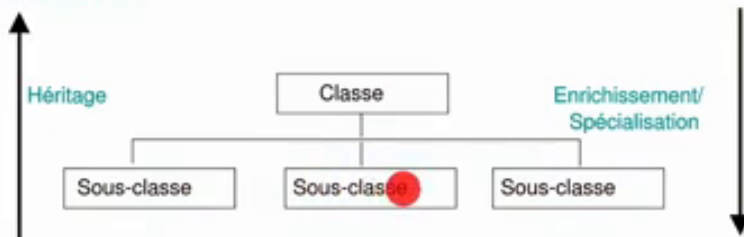




Après les notions d'*encapsulation* et d'*abstraction*, le troisième aspect essentiel de la « Programmation Orientée Objet » est la notion d'**héritage**.

L'héritage représente la relation «**est-un**».

Il permet de créer des classes *plus spécialisées*, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.



En anticipant bien sûr sur ce qui va suivre, avec l'héritage, nous pourrions faire quelque chose comme ceci, avoir une classe Personnage qui regroupe l'ensemble des caractéristiques communes, nom, énergie, durée de vie, mais aussi des méthodes, donc par exemple "rencontrer un personnage", donc une classe Personnage qui regroupe l'ensemble des caractéristiques de ces personnages, et dont hériterait le voleur, le magicien, le guerrier, ayant chacun ses spécificités, une arme pour le guerrier, une baguette pour le magicien, une spécialisation ici avec une méthode voler pour le voleur, et puis de la classe magicien, on pourrait aussi faire une extension supplémentaire, un héritage de plus, pour avoir un sorcier qui est une sorte de magicien, qui aurait la baguette et en plus un bâton. Voilà ce à quoi sert l'héritage en programmation orientée objet. Il représente la relation "est-un", et permet d'éviter la duplication de codes, d'attributs et de méthodes. L'héritage permet ainsi de créer des classes plus spécialisées, des classes enrichies. On va parler donc de super-classes, pour une classe à partir de laquelle on va dériver des sous-classes.

notes

résumé

2m 16s



## Héritage (2)

Lorsqu'une sous-classe **C1** est créée à partir d'une super-classe **C**,

- ▶ le type est *hérité* : un **C1** **est** (aussi) un **C**
- ▶ **C1** va *hériter* de l'ensemble :
  - ▶ des attributs de **C**
  - ▶ des méthodes de **C**  
(sauf les constructeurs)
- ☞ Les attributs et méthodes de **C** vont être disponibles pour **C1** sans que l'on ait besoin de les redéfinir explicitement dans **C1**.
- ▶ Par ailleurs :
  - ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **C1**
    - ☞ **enrichissement**
  - ▶ des méthodes héritées de **C** peuvent être redéfinies dans **C1**
    - ☞ **spécialisation**

On dira que les sous-classes héritent, et on mettra la flèche dans ce sens là, les sous-classes héritent de la super-classe, et ces sous-classes seront donc des extensions, soit par ajout d'attributs de méthodes, soit par spécialisation de méthodes, ce seront des extensions de la super-classe. Avant de les détailler chacune sur un exemple, voyons les implications de la relation d'héritage. Si on a une sous-classe C1 qui est créée à partir d'une classe C, qui hérite d'une classe C, on va dire que C1 "est un" C, c'est-à-dire que le type est hérité. Donc par exemple, si j'ai une variable x de type C, si j'ai une variable y de type C1, je peux tout à fait mettre y dans x, un C1 est aussi un C. Par ailleurs la classe C1 va hériter de l'ensemble des attributs et de l'ensemble des méthodes de la classe C. Quand on dit l'ensemble des attributs et l'ensemble des méthodes, c'est sauf les constructeurs, les constructeurs ne sont pas hérités. Les attributs et les méthodes de C seront donc disponibles dans C1 sans qu'on ait besoin de les redéfinir explicitement. Et enfin, des attributs et des méthodes supplémentaires

notes

résumé

3m 25s

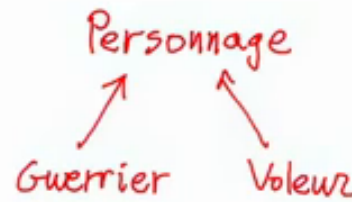


## Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- le type est *hérité* : un **Guerrier** **est** (aussi) **un** **Personnage** :

```
Personnage p;
Guerrier g;
// ...
p = g;
// ...
void afficher(Personnage);
// ...
afficher(g);
```



peuvent être ajoutées à C1, c'est ce qu'on appelle l'enrichissement, ou alors des méthodes héritées de C dans C1 peuvent être redéfinies, c'est ce qu'on appelle la spécialisation. Revoyons tout ceci en détails sur un exemple. Imaginons donc que l'on ait créé une super-classe **Personnage**, et deux sous-classes **guerrier** et **voleur**, par exemple. Donc on a dit que lorsqu'une sous-classe C1, par exemple ici **guerrier**, est créée à partir d'une super-classe C, donc par exemple ici **Personnage**, le type est hérité. En fait un **Guerrier** est aussi un **Personnage**, ce qui nous permet d'écrire le code suivant, supposons qu'on ait donc déclaré une variable **p** de type **Personnage**, une variable **g** de type **Guerrier**, on les manipule, et à un certain moment par exemple, on dit que le **Personnage p** est le **Guerrier g**, donc on met le **Guerrier g** dans le **Personnage p**, on peut tout à fait faire ceci, puisqu'un **Guerrier** est un **Personnage**. Evidemment on ne peut pas faire le contraire, on ne peut pas écrire **g = p**, un **personnage générique** n'est pas un **guerrier**, on peut avoir des **personnages** qui sont des **voleurs**, des **magiciens**, et on ne pourrait pas mettre un **voleur** dans un **guerrier**, ça n'aurait pas de sens.

### notes

### résumé

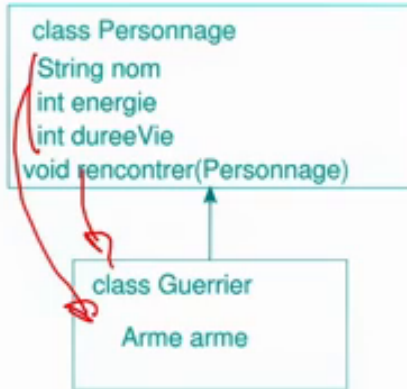
4m 37s



## Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- **Guerrier** va *hériter* de (i.e. recevoir) l'ensemble des attributs et des méthodes de **Personnage** (sauf les constructeurs)



```

Guerrier g = new Guerrier(...);
Voleur v = new Voleur(...);

g.rencontrer(v);
//...
// dans une méthode de Guerrier
energie = //...
  
```

Donc ici, la relation est bien orientée, c'est bien un guerrier qui est un personnage. De même, on pourrait imaginer donc avoir une fonction qui permet d'afficher un personnage, et on peut afficher `g`, qui est ici un guerrier, qui est passé comme paramètre à la fonction afficher en tant que personnage, la fonction afficher le verra en tant que personnage, on peut faire ceci parce que un **Guerrier** est un **Personnage**. Deuxième aspect de l'héritage : **Guerrier** va hériter, c'est-à-dire recevoir, contenir l'ensemble des attributs et des méthodes de **Personnage**. Donc par exemple, si **Personnage** avait un nom, une énergie, une durée de vie, et disons une méthode `rencontrer`, et que **Guerrier** hérite de la classe **Personnage**, alors dans **Guerrier** on aura aussi un nom une énergie, une durée de vie, sans qu'on ait besoin de les redéfinir, et aussi une méthode `rencontrer`, donc on pourrait faire des choses comme ceci, supposons qu'on ait donc ici un **Guerrier** `g` et un **Voleur** `v`, qui est aussi un **Personnage**, `g` ici, en tant que **Guerrier**, va hériter la méthode `rencontrer`, et donc on pourrait appeler, pourvu que cette méthode

### notes

### résumé

5m 49s



## Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- ▶ des attributs et/ou méthodes supplémentaires peuvent être **définis** par la sous-classe **Guerrier** : **arme**
- ▶ des méthodes héritées de **Personnage** peuvent être **redéfinies** dans **Voleur** : **rencontrer(Personnage)**

soit dans l'interface publique de **Personnage**, on va pouvoir appeler "g.rencontrer(v)", ici on passe le voleur v comme argument à la méthode rencontrer, mais ce qui m'importe ici c'est de voir que la méthode rencontrer est disponible, accessible dans g, Guerrier a hérité de la méthode rencontrer de Personnage, de même si on avait une méthode quelconque dans Guerrier, on pourrait directement comme ceci utiliser l'attribut énergie dans une méthode de guerrier, pourvu qu'il ne soit pas privé, l'attribut énergie qui est hérité de personnage dans la classe Guerrier, sans qu'on ait besoin de rajouter aucune ligne de code, c'est automatique, c'est une des conséquences de l'héritage. Troisième et dernier aspect enfin que nous voulions souligner, même si les attributs et les méthodes sont hérités, qu'on les a dans les sous-classes, on peut quand même ajouter des attributs et des méthodes supplémentaires, ce qu'on appelle l'enrichissement, par exemple dans la classe Guerrier, on pourrait rajouter une arme, et on peut aussi donc redéfinir des méthodes, ce qu'on appelle, les spécialiser, donc par exemple, on pourrait redéfinir la méthode rencontrer dans la classe Voleur,

### notes

### résumé

7m 1s



L'héritage permet donc :

- ▶ d'expliciter des relations structurelles et sémantiques entre classes
- ▶ de réduire les redondances de description et de stockage des propriétés



### Attention !

- ▶ l'héritage doit être utilisé pour décrire une relation « **est-un** » ("is-a")
- ▶ il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

en disant que par exemple un Voleur quand il rencontre un autre Personnage, il le vole, par exemple.

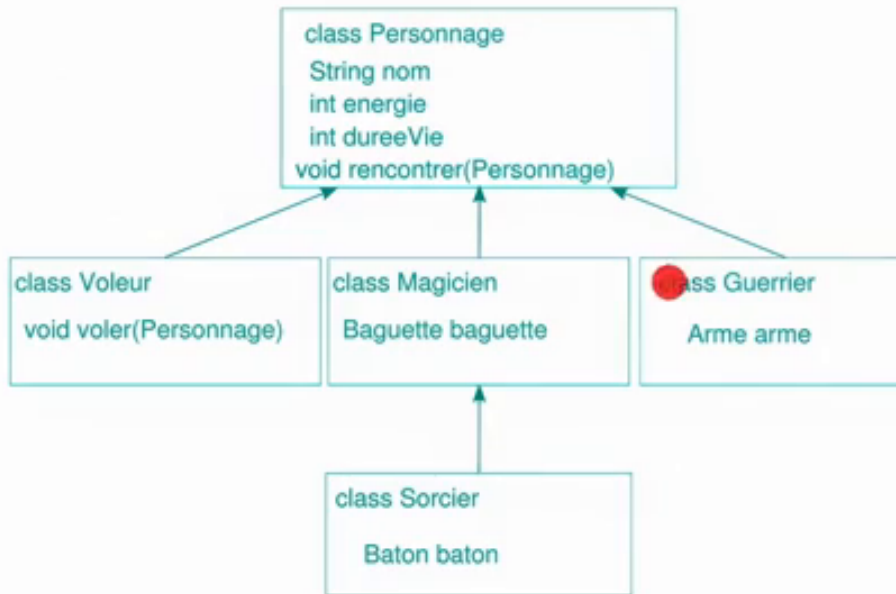
notes

résumé

8m 13s



## Exemple : héritage



L'héritage est donc une notion très importante en programmation orientée objet, parce qu'elle permet d'organiser le code, de rendre la conception beaucoup plus claire, elle permet d'explicitier les relations structurelles et sémantiques qui existent entre différentes classes,

notes

résumé

8m 19s



L'héritage permet donc :

ORGANISE  
LE  
CODE

▶ d'expliciter des relations structurelles et sémantiques entre classes

▶ de réduire les redondances de description et de stockage des propriétés



### Attention !

- ▶ l'héritage doit être utilisé pour décrire une relation « **est-un** » ("is-a")
- ▶ il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

par exemple, un Guerrier est un Personnage, un Sorcier est une sorte de Magicien.

notes

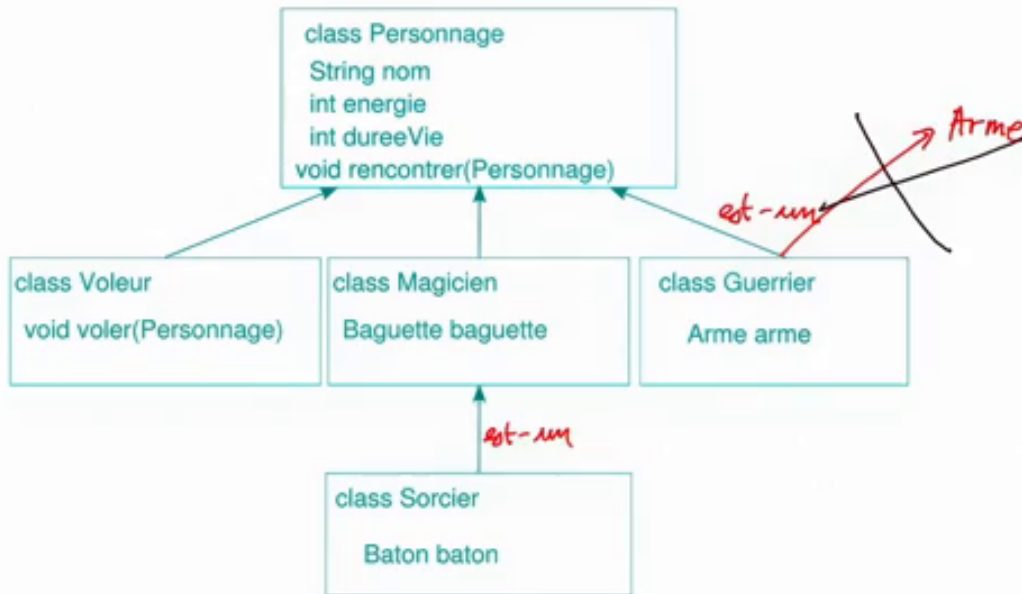
résumé

8m 34s





## Exemple : héritage



Elle permet aussi donc d'éviter d'avoir à réécrire de grandes portions de code, donc ce qu'on appelle de la redondance de code. Attention donc à bien utiliser l'héritage à bon escient, il faut utiliser l'héritage pour représenter la relation "est-un", un Guerrier est un Personnage, mais on ne va pas dire qu'un Guerrier est une Arme, on dit qu'un Guerrier a une Arme, et pour représenter donc la relation "a/possède", on utilisera ce qu'on avait vu jusqu'à maintenant, l'encapsulation. Et donc, pour résumer, un Guerrier héritera de la classe Personnage, mais encapsulera, aura comme attribut une arme. comme attribut une arme.

### notes

### résumé

8m 40s

