

Support de cours

Cours:

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

Vidéo:

### W16-05-JAVA-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Défaut des méthodes. Concept d'interface. Notion de définition. En-têtes de méthodes abstraites. Méthodes d'une interface. Fait possible. Fameux exemple des personnages. Nombre de méthodes statiques. Vidéo précédente. Définition concrète. Défaut d'une méthode. Règles d'utilisation des méthodes. Méthodes statiques. Classe personnage. Objectif de cette vidéo.**



[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/>

# Interfaces : Nouveautés depuis Java 8

## (Partie 1)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Comme esquissé dans une vidéo précédente,

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

---

---

---

---

Les interfaces jusqu'à Java 7 ne peuvent contenir que :

- ▶ des constantes ✓
- ▶ des méthodes abstraites ✓

Nouveautés depuis Java 8, elles peuvent aussi contenir :

1. des définitions par défaut pour les méthodes ✓
2. des méthodes statiques

le concept d'interface a fait l'objet depuis Java 8 d'ajouts importants. L'objectif de cette vidéo est de vous présenter ces ajouts et en particulier la notion de définition par défaut des méthodes dans une interface. Nous avons vu que, jusqu'à Java 7 compris, une interface pouvait contenir des constantes ainsi que des en-têtes de méthodes abstraites. Ce contenu est bien évidemment toujours également possible pour les interfaces en Java 8, mais en plus il est possible de donner un corps aux méthodes. On peut donner une définition par défaut à des méthodes d'une interface, c'est une nouveauté notable. Il est également possible de donner, de définir à l'intérieur d'une interface un certain nombre de méthodes statiques de la même façon que nous procédons pour définir des méthodes statiques dans une classe. Dans le cadre de cette vidéo, nous allons essentiellement exposer cette partie,

notes

résumé

0m 11s



Reprenons notre exemple avec les personnages :



Supposons maintenant que certains personnages puissent chevaucher des montures.

c'est-à-dire la définition par défaut pour les méthodes d'une interface. Le second volet est l'objet d'un complément PDF que vous pouvez retrouver sur le site du cours.

notes

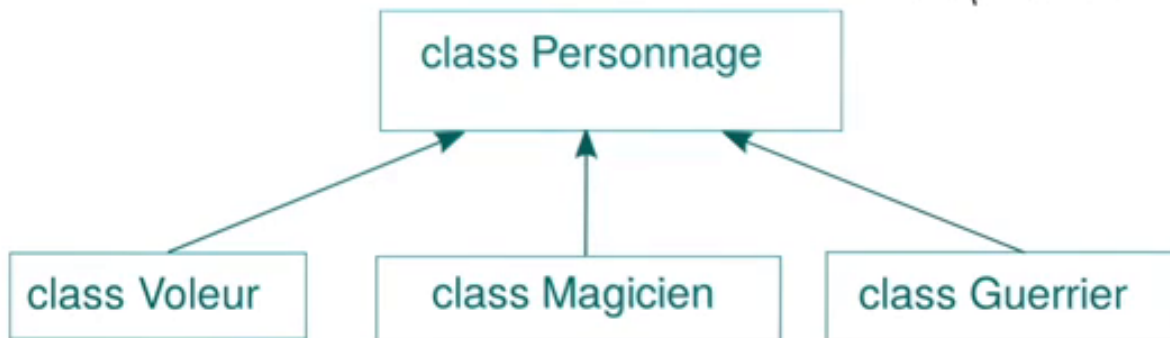
résumé

1m 1s



Reprenons notre exemple avec les personnages :

*se Déplace () { ... }*



Supposons maintenant que certains personnages puissent chevaucher des montures.

Pour illustrer la notion de définition par défaut d'une méthode dans une interface, partons d'un exemple. Nous utiliserons bien entendu notre fameux exemple des personnages dans un jeu. Supposons que nous souhaitions désormais modéliser le fait que certains personnages du jeu soient capables de chevaucher des montures. Ceci veut dire que ces personnages auraient désormais la possibilité d'avoir des fonctionnalités liées au fait qu'ils chevauchent une monture. Par exemple, est-ce que un personnage qui chevauche une monture a le droit d'en descendre durant le jeu? Comment va-t-il se déplacer en utilisant cette monture? Et ainsi de suite. Où placer ces fonctionnalités dans notre conception? Alors supposons par exemple que dans notre jeu, seuls les voleurs et les guerriers soient habilités à chevaucher des montures. Il ne fait pas de sens, dans ce cas, de placer les fonctionnalités liées au fait de chevaucher une monture dans la classe Personnage. Ici, il ne ferait pas sens de placer une méthode se déplace indiquant

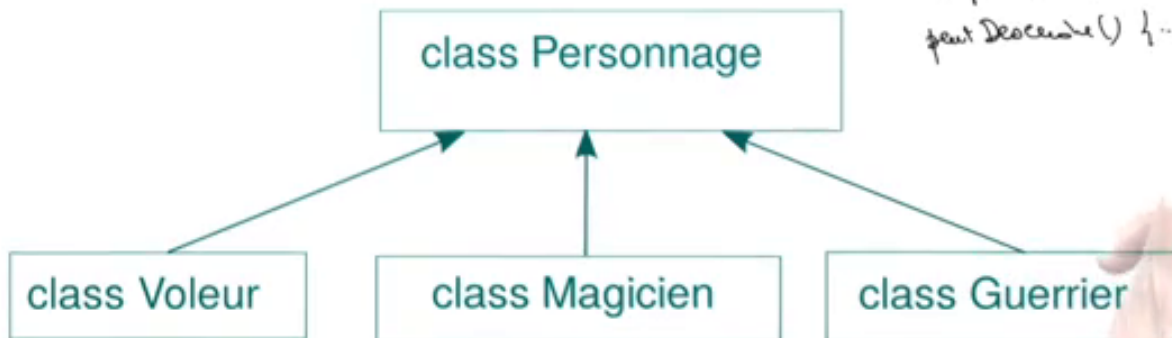
notes

résumé

1m 9s



Reprenons notre exemple avec les personnages :



*se Déplace () { ... }  
peut Descendre () { ... }*

Supposons maintenant que certains personnages puissent chevaucher des montures.

comment un personnage se déplace avec une monture dans la classe Personnage, pas plus qu'il ne ferait sens de placer dans Personnage une méthode peut descendre

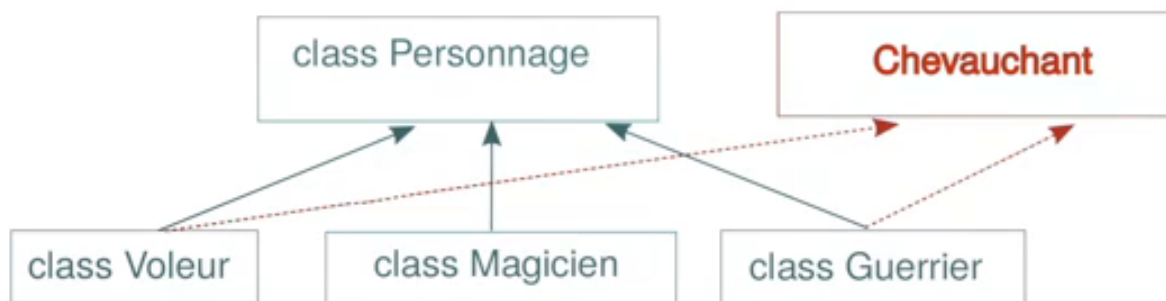
notes

résumé

2m 13s



On pourrait donc imaginer la conception suivante :



indiquant si le personnage a le droit de descendre de sa monture ou pas. Pourquoi? Eh bien tout simplement parce que le magicien n'est pas capable de chevaucher une monture, et donc il ne fait pas sens de lui faire hériter des méthodes se déplace et peut descendre.

notes

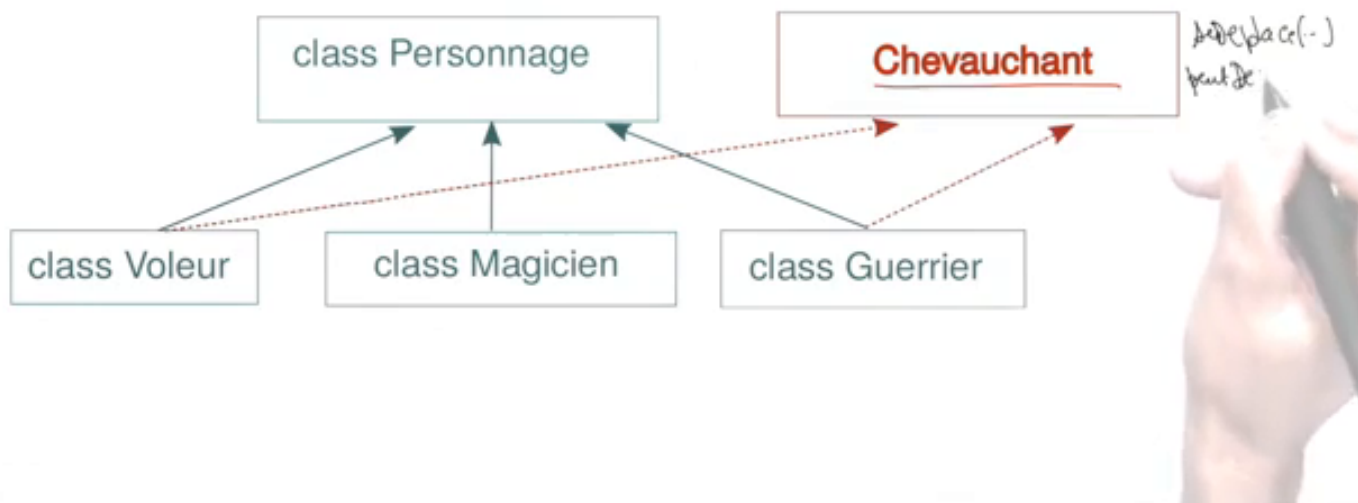
résumé

2m 24s





On pourrait donc imaginer la conception suivante :



Il est donc naturel ici de plutôt placer les méthodes en question dans une interface, ici l'interface Chevauchant. On mettrait donc dans cette interface des méthodes typiquement liées au fait de se déplacer au moyen d'une monture, ou pour interroger comment on peut

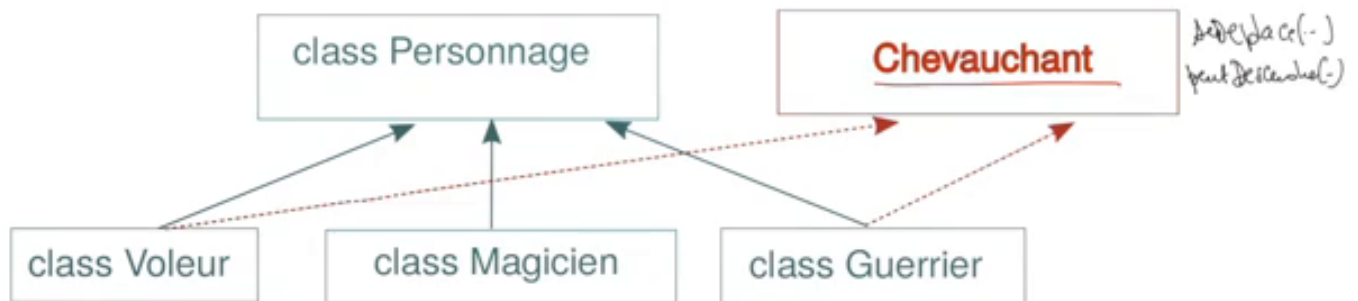
notes

résumé

2m 39s



On pourrait donc imaginer la conception suivante :



faire usage de cette monture, par exemple est-ce qu'on peut en descendre ou pas.

notes

résumé

2m 58s



Pour le contenu de l'interface `Chevauchant`, on pourrait avoir :

```
interface Chevauchant
{
    void seDeplace();           // effectue le déplacement en chevauchant
    boolean peutDescendre();    // peut-on descendre de la monture ou pas?
}
```

Supposons que dans la plupart des cas un personnage chevauchant une monture ne puisse pas en descendre ...

Voici donc à quoi pourrait ressembler le contenu de l'interface `Chevauchant`. Cette interface pourrait contenir l'en-tête d'une méthode `se déplace`, cette fonctionnalité aurait pour vocation d'indiquer comment s'effectue le déplacement en chevauchant, ainsi que d'une méthode `peut descendre` tournant un booléen et indiquant si l'on peut oui ou non descendre de la monture. Vous remarquerez que ce contenu est conforme à ce qui était décrit pour les

notes

résumé

3m 2s



Depuis Java 8, il est possible de donner une définition par défaut à certaines méthodes :

```
interface Chevauchant
{
    void seDeplace();
    default boolean peutDescendre() { return false; }
}
```

interfaces dans les versions antérieures à Java 8. Nous avons ici la déclaration de deux méthodes abstraites : toute classe que l'on souhaiterait instanciable et qui implémenterait l'interface Chevauchant devra impérativement fournir une définition concrète de ces deux méthodes justement pour pouvoir être instanciable. Voyons ce qu'il est possible de faire en plus depuis Java 8, et supposons que dans la plupart des cas un personnage qui chevauche une monture ne puisse pas en descendre. Eh bien depuis Java 8, il est possible de spécifier ce comportement au sein même de l'interface. Et ceci en donnant une définition par défaut à la méthode peut descendre. Concrètement, cela signifie que cette méthode peut désormais avoir un corps. Ce corps consiste ici simplement à retourner false, ce qui indique que l'on ne peut pas descendre de la monture. Toute méthode d'interface dotée d'un corps, c'est-à-dire d'une définition concrète comme ici, devra impérativement être précédée du modificateur default, faute de quoi le compilateur vous rappellera à l'ordre en vous indiquant qu'une méthode abstraite ne peut avoir de corps. Rappelez vous que tout en-tête de méthode est automatiquement abstrait dans une interface, même si ce modificateur,

notes

résumé

3m 25s



### Syntaxe :

```
interface UneInterface
{
    default définition par défaut de la méthode
}
```

le modificateur abstract, n'est pas explicitement présent.

### notes

---

---

---

---

---

---

---

---

---

---

### résumé

4m 37s



---

---

---

---

---

---

---

---

---

---

## Syntaxe :

```
interface UneInterface
{
    default définition par défaut de la méthode
}
```

Cette nouveauté soulève de nouvelles problématiques :

- ▶ quelles sont les règles d'utilisation des méthodes avec définition par défaut ?
- ▶ comment gérer les ambiguïtés pouvant se produire lors de définitions multiples (par des classes ou des interfaces) ?

La syntaxe formelle pour la déclaration d'une méthode avec définition par défaut dans une interface est donc simplement comme suit : vous avez donc le mot réservé `default` suivi de la définition concrète de la méthode, cette définition se fait de façon tout à fait analogue à ce que vous savez faire dans une classe, vous allez mettre un en-tête suivi d'un corps entre accolades. Bien entendu cette nouveauté va soulever de nouvelles problématiques. Nous allons voir dans ce qui suit quelles sont les règles d'utilisation des méthodes avec définition par défaut. A-t-on le droit de les redéfinir ? Comment les classes qui implémentent des interfaces avec méthodes avec définitions par défaut peuvent les utiliser ? Mais surtout nous allons voir comment gérer les ambiguïtés qui peuvent résulter de la présence de ces définitions par défaut. Par exemple, supposons que l'on ait une interface `I` proposant une définition par défaut pour une méthode donnée, et supposons qu'une classe `C` qui implémenterait l'interface `I` propose également une méthode de même en-tête avec potentiellement une autre définition. Nous aurions donc ici potentiellement une situation de conflit.

## notes

## résumé

4m 41s



## Syntaxe :

```
interface UneInterface
{
    default définition par défaut de la méthode
}
```

```
interface I1 {
    default void m() { ... }
}
interface I2 {
    default void m() { ... }
}
```

Cette nouveauté soulève de nouvelles problématiques :

- ▶ quelles sont les règles d'utilisation des méthodes avec définition par défaut ?
- ▶ comment gérer les ambiguïtés pouvant se produire lors de définitions multiples (par des classes ou des interfaces) ?

```
class C implements I1, I2 {
}
```

Autre exemple, supposons que nous ayons cette fois deux interfaces et que chacune de ces interfaces propose une définition par défaut pour une méthode de même en-tête. Donc ici l'interface I 1 propose une méthode M avec définition par défaut ... et pareil pour une seconde interface I 2 qui proposerait donc elle aussi une définition par défaut pour la méthode M. Que se passe-t-il si une classe implémente les deux interfaces?

## notes

## résumé

6m 10s





Donc là encore, on a potentiellement une situation conflictuelle.

notes

---

---

---

---

---

---

---

---

---

---

résumé

7m 1s



---

---

---

---

---



## Règle 1 : héritage et définition

Les définitions par défaut des méthodes s'héritent et peuvent être redéfinies plus bas dans les hiérarchies d'interfaces :

```
interface Cavalier extends Chevauchant
{
    default void seDeplace() { System.out.println("au trot"); }
}
```

➤ Inutile de redonner une définition par défaut à `peutDescendre` si on en est satisfait.

```
interface Chevauchant
{
    void seDeplace();
    default boolean peutDescendre() { return false; }
}
```

Quatre règles fondamentales régissent l'utilisation des méthodes avec définition par défaut dans les interfaces. Nous allons maintenant vous les exposer. La première règle est que les définitions par défaut des méthodes s'héritent. Rappelez vous que nous avons défini l'interface Chevauchant comme suit : cette interface proposait une méthode abstraite se déplace ainsi qu'une définition par défaut d'une méthode peut descendre retournant false pour indiquer que le comportement par défaut d'un personnage chevauchant était de ne pas descendre de sa monture. Il est tout à fait possible de définir une sous interface, ici la sous interface Cavalier de l'interface Chevauchant, ne redéfinissant pas explicitement la méthode peut descendre. Si un cavalier a pour comportement par défaut qu'il ne descend pas de sa monture, alors on peut parfaitement se contenter de la définition héritée de la super interface Chevauchant, et dans ce cas-là, ne pas redéfinir cette méthode et garder celle héritée de plus haut. Vous aurez noté qu'il est parfaitement possible de donner dans une sous interface, ici dans Cavalier, une définition par défaut pour

### notes

### résumé

7m 8s



