



Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W16-05-JAVA-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Définitions concrètes. Règle numéro. Classe guerrier. Défaut des méthodes. Interface cavalier. Défaut des méthodes d'une interface. Méthodes. Mage ultime. Nombre de méthodes. Nombreuses classes. Définition concrète des méthodes. Règles de base. Invocation d'une méthode statique. Interface. Fait possible.



[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/>

page 1/12

Interfaces : Nouveautés depuis Java 8

(Partie 2)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



Une classe n'est plus obligée de redéfinir les méthodes des interfaces qu'elle implémente si celles-ci ont une définition par défaut

- Si la classe Guerrier implémente l'interface Cavalier elle est instantiable en l'état

```
interface I {  
    void m1();  
    void m2();  
    void m3(); ←  
}  
  
class C1 implements I {  
    public void m1() { ... }  
    "    " m2() { ... }  
}  
  
class C2 implements I {  
    public void m1() { ... }  
    "    " m2() { ... }  
}
```

Faisons simple, ici, deux méthodes, m1 et m2 ; supposons que de nombreuses classes implémentent cette interface. On en prend deux ici, en guise d'exemple. Donc une classe C1 et une classe C2, qui toutes deux implémentent l'interface I. Ces deux classes sont déclarées si non abstraites, ce qui veut dire qu'on les souhaite instanciables ; pour qu'elles soient instanciables, elles doivent impérativement fournir une définition concrète des méthodes m1 et m2 ; donc si ces classes existent et qu'elles sont compilables, c'est qu'elles proposent une définition concrète des méthodes proposées par l'interface implémente. Donc idem ici, pour m2. Et on devrait faire la même chose dans l'interface, dans la classe pardon, finie. Supposons maintenant que, les besoins évoluant au cours du temps, on réalise que l'interface I1 devrait posséder en plus une méthode m3. Que se passe-t-il alors avec le code existant ? Eh bien tout simplement, si on l'écrit comme ceci, le code existant ne compile plus. Les classes C1 et C2 sont non abstraites, et donc,

notes

résumé

1m 13s



Une classe n'est plus obligée de redéfinir les méthodes des interfaces qu'elle implémente si celles-ci ont une définition par défaut

- Si la classe `Guerrier` implémente l'interface `Cavalier` elle est instantiable en l'état

Elle peut néanmoins le faire si nécessaire :

```
class Guerrier extends Personnage implements Cavalier
{
    //...
    @Override
    boolean peutDescendre() { return false; }
}
```

à l'instar de ce qu'on avait fait pour `m1` et `m2`, il faudrait impérativement retoucher ces classes pour donner une définition concrète à la méthode `m3`. Le fait de pouvoir désormais proposer une définition par défaut, pour la méthode `m3`, nous permet de contourner ce problème. Selon la règle que nous venons d'exposer, une classe qui implémente une interface n'est pas dans l'obligation de redéfinir une méthode ayant une définition par défaut, et dans ce cas-là il n'est plus nécessaire de définir `m3` à l'intérieur des classes, pour que ces classes continuent à être compilables et instanciables.

notes

résumé

2m 37s



Règle 3 : les classes ont la précedence (2)

La classe `Guerrier` peut bien sûr redéfinir la méthode `seDeplace` pour plutôt choisir celle de l'interface :

```
class Guerrier extends Personnage implements Cavalier
{
    public void seDeplace() {
        Cavalier.super.seDeplace();
    }
}
```

interface I {
 default void m1() {}
 static void m2() {}
}

notes

résumé

3m 12s



Règle 3 : les classes ont la précedence (2)

La classe `Guerrier` peut bien sûr redéfinir la méthode `seDeplace` pour plutôt choisir celle de l'interface :

```
class Guerrier extends Personnage implements Cavalier
{
    public void seDeplace() {
        Cavalier.super.seDeplace();
    }
}
```

interface I {
 default void m1() {}
 static void m2() {}
}

définie dans l'interface Cavalier. Pour invoquer une méthode avec définition par défaut dans une interface, on mettra le nom de l'interface, suivi d'un point, suivi du nom réservé, super, encore un point, et le nom de la méthode avec définition par défaut. Vous noterez ici l'usage du mot réservé, super. Ce mot réservé permet de faire la distinction entre l'invocation d'une méthode avec définition par défaut et celle de l'invocation d'une méthode statique, interface. Par exemple, il est possible de déclarer une interface comme ceci, d'y placer les méthodes avec définition par défaut, ce que nous savons faire désormais, mais également d'y placer des méthodes statiques, de la même façon que nous pouvons placer des méthodes statiques dans une classe.

notes

résumé

Règle 3 : les classes ont la précedence (2)

La classe **Guerrier** peut bien sûr redéfinir la méthode **seDeplace** pour plutôt choisir celle de l'interface :

```
class Guerrier extends Personnage implements Cavalier
{
    public void seDeplace() {
        Cavalier.super.seDeplace();
    }
}
```

```
interface I {
    default void m1() { }
    static void m2() { }
}

class C implements I {
    void m() {
        I.m2();
        I.super.m1();
    }
}
```

... Supposons qu'une classe C implémente l'interface I, si l'on souhaite dans une méthode de la classe C invoquer l'une ou l'autre des méthodes de l'interface, on procédera comme suit. Donc pour invoquer la méthode statique, on mettra le nom de l'interface, suivi d'un point, suivi du nom de la méthode statique. Pour invoquer la méthode avec définition par défaut, on utilisera le nom de l'interface, suivi de, super, suivi du nom de la méthode avec définition par défaut. Vous noterez donc la distinction, ici ; pour invoquer une méthode statique d'interface, on met le nom de l'interface, suivi du nom de la méthode ; c'est exactement la même syntaxe que l'on utilisait pour invoquer un membre statique d'une classe. Par contre, pour invoquer une méthode avec définition par défaut, on insère le mot réservé, super. Je rappelle que les méthodes statiques d'interface font l'objet d'un complément PDF sur le site du cours. Petite remarque, ici, avant de poursuivre, ces deux méthodes d'interface ne pourraient pas avoir le même nom. Il n'est pas toléré par le compilateur qu'une méthode d'interface de même entête

notes

résumé

6m 25s



Règle 4 : les classes doivent lever les ambiguïtés (1)

Les conflits entre définitions par défaut sont désormais possibles :

```
interface Dragonier extends Chevauchant
{
    default void seDeplace() { System.out.println("vole"); }
}

interface SeTeleporte
{
    default void seDeplace { System.out.println("plop !"); }
}

class MageUltime extends Magicien implements Dragonier, SeTeleporte {
    // conflit sur la définition de seDeplace !
}
```

- Les classes qui implémentent des interfaces conflictuelles doivent lever l'ambiguïté

soit à la fois déclarée comme statique et ait, en même temps, une définition par défaut.

notes

résumé

7m 49s



Règle 4 : les classes doivent lever les ambiguïtés (2)

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier, SeTeleporte
{
    // un MageUltime se déplace comme un Dragonier
    public void seDeplace { Dragonier.super.seDeplace(); }
}
```

Que se passe-t-il, finalement, si une classe implémente deux interfaces proposant toutes deux une définition par défaut pour une même méthode? Ici, par exemple, nous avons deux interfaces ; l'interface, `Dragonier`, et l'interface, `Se téléporte`. Chacune de ces deux interfaces propose une définition par défaut pour une méthode, `Se déplace`, ayant exactement le même entête dans les deux cas. Nous avons ici une classe, `Mage ultime`, qui implémente les deux interfaces ; lorsque l'on invoque la méthode, `Se déplace`, sur une instance de `Mage ultime`, laquelle des deux méthodes est-elle invoquée? Il y a donc ici une situation de conflit. La règle adoptée par Java pour faire face à ce genre de situations est que les classes qui implémentent les interfaces conflictuelles sont en charge de lever l'ambiguïté. Par exemple, la classe,

notes

résumé

7m 55s



Règle 4 : les classes doivent lever les ambiguïtés (2)

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier, SeTeleporte
{
    // un MageUltime se déplace comme un Dragonier
    public void seDeplace { Dragonier.super.seDeplace(); }
}
```

Mage ultime, qui implémente les deux interfaces conflictuelles, devra redéfinir la méthode, Se déplace, pour spécifier laquelle des deux elle veut utiliser. Laquelle des deux définitions par défaut, issues des interfaces, elle veut utiliser. Donc ici le choix est fait,

notes

résumé

8m 45s



le choix s'est porté de la définition par défaut issue de l'interface, Dragonnier. La méthode, Se déplace, de Mage ultime, est redéfinie de façon à invoquer la méthode, Se déplace, telle que définie par défaut dans l'interface, Dragonnier. Rien n'empêche, bien sûr, de faire le choix alternatif de la méthode, Se déplace, issue de, Se téléporte. Dans ce cas-ci, la méthode, Se déplace, est redéfinie dans, Mage ultime, de sorte à choisir la méthode, Se déplace, telle que définie par défaut dans l'interface, Se téléporte. Bien d'autres implémentations sont, évidemment, possibles ; et rien n'empêche la classe, Mage ultime, de redéfinir la méthode, Se déplace, de sorte à utiliser les définitions par défaut des deux interfaces. Ici vous avez un exemple de définition de, Se déplace, qui fait appel dans certaines conditions à la méthode, Se déplace, issue de, Se téléporte ; et dans d'autres conditions à la méthode, Se déplace, issue de Dragonnier. Si le mage ultime peut descendre de sa monture, alors il se déplace en se téléportant, sinon, eh bien, il volera comme un dragonnier. eh bien, il volera comme un dragonnier.

9m 2s

