

Support de cours

Cours:

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

Vidéo:

### W17-05-copieprofonde-JAVA-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Copie profonde. Moyen d'un constructeur de copies. Objet de type. Étude de cas. Problématique particulière. Tableau dynamique. Objet m. Long du cours. Cœur de l'instance courante. Montre de toto. Nouveau mécanisme. Heure du mécanisme de m2. Heure du mécanisme. Constructeur de copie de montres. Objet.**



[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/>

# Etude de cas : copie profonde

## (Partie 1)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Pour conclure cette étude de cas sur les montres,

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

---

---

---

---

```
Montre maMontre = new Montre(...);  
  
// votre ami veut la même montre  
// et dans ce cas on sait qu'il ne faut pas juste affecter  
// les références  
Montre montreToto = new Montre(maMontre);
```

*montreToto = maMontre*

nous allons nous intéresser à une problématique particulière, à savoir : comment copier des montres ? Il ne s'agit pas ici de contrefaçons, mais cela va nous ouvrir à une problématique intéressante en programmation, celle de la copie profonde. Situons le problème concrètement. Vous avez une montre, et votre ami veut la même. Nous avons appris tout au long du cours que l'affectation en Java ne permet pas de réaliser la copie à proprement parler. Si j'écris « `montreToto = maMontre` »

notes

résumé

0m 6s

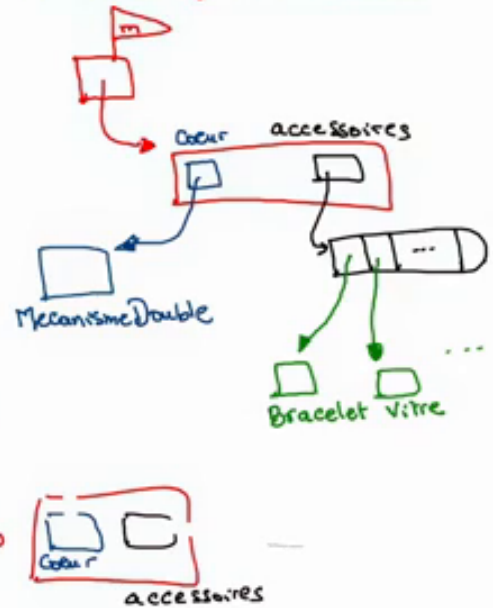


## Copie profonde

```
class Montre extends Produit {  
    private Mecanisme coeur;  
    private ArrayList<Accessoire> accessoires;  
    //...  
}
```

Si l'on veut faire des copies de **Montres**, on **doit** ici faire une **copie profonde** :  
copie de chaque constituant (mécanisme, accessoires)

Montre m = new Montre (... EPFL  
m.ajouter(new Bracelet(...  
...  
Montre m2 = new Montre(m);



je suis simplement en train d'affecter la référence de « maMontre » dans la variable montreToto. Donc en clair... Ma montre et la montre de Toto pointent vers le même objet en mémoire. Ceci signifie que si une montre offre un mécanisme permettant de régler l'heure, par exemple, alors le fait de régler ma montre va aussi régler celle de Toto ; et c'est un comportement plutôt inattendu, ici. Il nous faut donc ici un mécanisme permettant de réellement copier une montre. Essayons de voir comment nous pouvons réaliser ce traitement, au moyen d'un constructeur de copies. Une montre est un objet relativement complexe qui est constitué d'un attribut « coeur » de type « Mecanisme », et d'une liste « d'Accessoires ». Donc, lorsque je déclare un objet de type « Montre », que je l'initialise et que j'y ajoute, par exemple, des accessoires comme ceci, j'aboutis à une représentation en mémoire qui est la suivante, donc ma variable « m », de type « Montre », est une référence vers un objet type « Montre », lequel a comme attribut un « coeur » qui est une référence vers un objet, par exemple de type « MecanismeDouble », et dont le second attribut est une référence vers un tableau dynamique qui contient à son tour des références vers des accessoires. Lorsque je construis une montre par copie à partir de « m », à quelle situation est-ce que je souhaite aboutir ? Si je me contente de copier « m » dans « m2 » en copiant champ à champ les différents attributs... je vais avoir pour le cœur de « m2 » la même valeur que le cœur de « m », ce qui veut dire que je pointe vers le même objet en mémoire, et je vais avoir pour les accessoires de « m2 », une référence vers le même tableau. Nous nous trouvons ici confrontés

### notes

### résumé

0m 37s

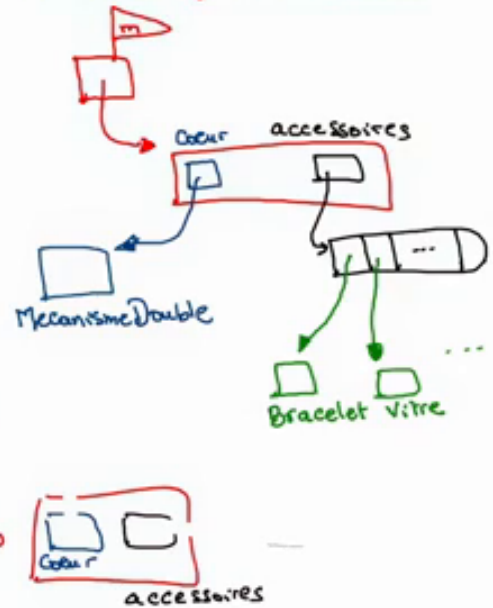


## Copie profonde

```
class Montre extends Produit {  
    private Mecanisme coeur;  
    private ArrayList<Accessoire> accessoires;  
    //...  
}
```

Si l'on veut faire des copies de **Montres**, on **doit** ici faire une **copie profonde** :  
copie de chaque constituant (mécanisme, accessoires)

Montre m = new Montre (... EPFL  
m.ajouter(new Bracelet(...  
Montre m2 = new Montre(m);



à des problématiques analogues à celles rencontrées précédemment, puisque l'objet m et l'objet m2 partagent le même tableau d'accessoires en mémoire, ajouter un accessoire au tableau de m revient à ajouter un accessoire au tableau de m2. De même, régler l'heure du mécanisme de m, revient à régler l'heure du mécanisme de m2. Pour avoir deux objets Montre qui ont, certes, les mêmes valeurs, mais qui sont totalement indépendants, il faut réaliser ce que l'on appelle la « copie profonde ». La situation à laquelle nous voulons arriver est plutôt la suivante :

notes

résumé

Copie de surface :

```
public Montre(Montre autre) {  
    super(autre);  
    coeur = autre.coeur;  
    accessoires = autre.accessoires;  
}
```

nous voulons que m2 ait un cœur bien à elle qui serait identique... à celui de m, mais distinct en mémoire, et un ensemble d'accessoires bien à elle aussi dont les valeurs pointeraient vers les objets identiques mais distincts en mémoire. C'est ce qu'on appelle donc, la copie profonde. Si l'on écrit le constructeur de copie de montres en se contentant d'affecter au cœur de l'instance courante le cœur de l'objet copié,

notes

résumé

3m 37s



```
public Montre(Montre autre)
{
    super(autre);
    coeur = new Mecanisme(autre.coeur);
    accessoires = new ArrayList<Accessoire>();
    for (Accessoire acc : autre.accessoires) {
        accessoires.add(new Accessoire(acc));
    }
}
```

et si l'on fait la même chose pour les accessoires, on réalise ce que l'on appelle la « copie de surface » qui, comme montré tout à l'heure, par opposition à la copie profonde, va simplement permettre aux deux montres de partager le même cœur et de partager la même liste d'accessoires. Donc cette façon de procéder n'est pas satisfaisante. Il faudrait donc réellement copier en profondeur chacune des entités mentionnées précédemment, c'est-à-dire mettre dans le cœur de l'objet courant un nouveau mécanisme construit à partir du cœur copié et construire une nouvelle liste d'accessoires où chaque accessoire, ajouté à la liste, serait une copie de l'accessoire correspondant de l'objet copié.

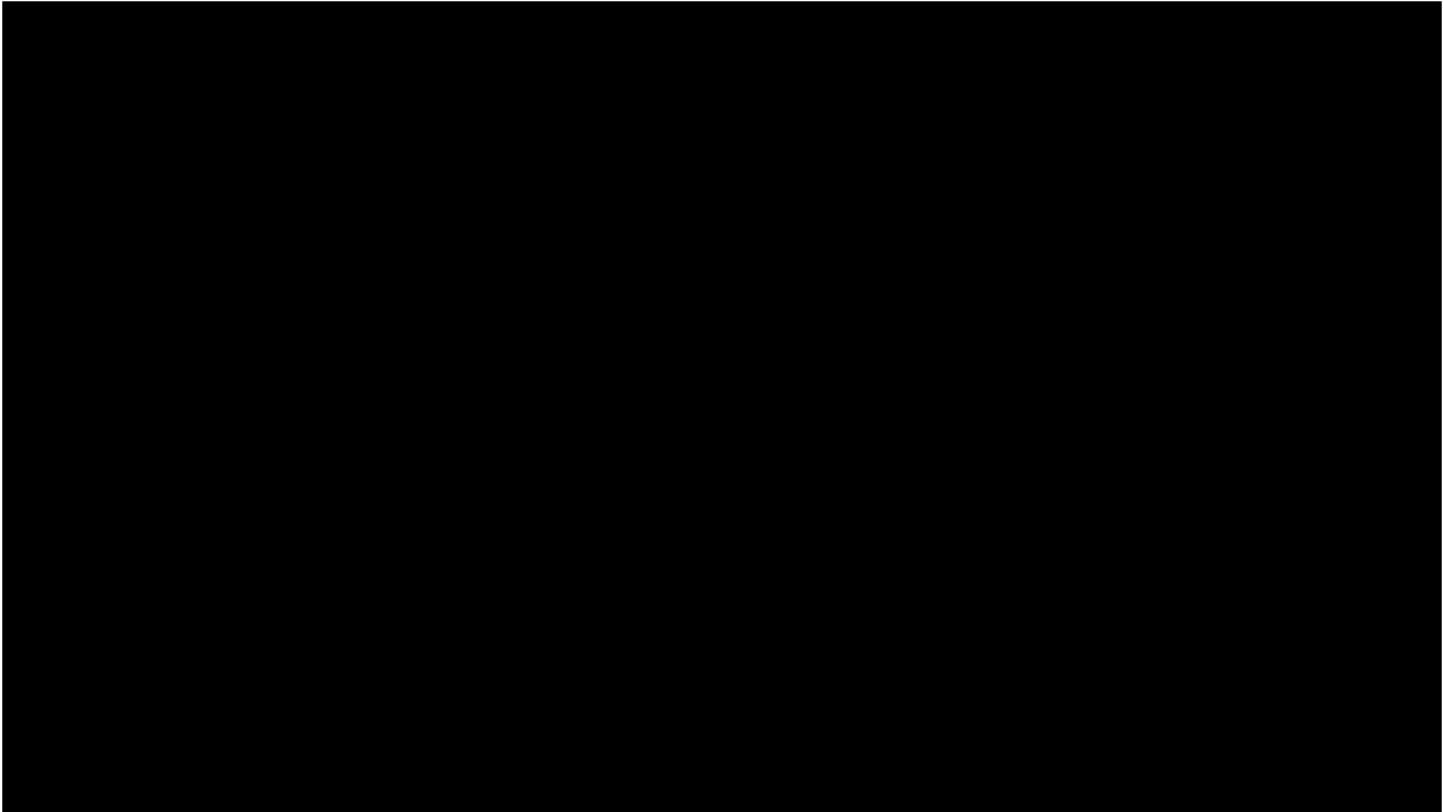
## notes

## résumé

4m 25s







Ici, on parcourrait la liste d'accessoires de l'objet copié et on créerait une copie de chacun de ces accessoires que l'on mettrait dans la nouvelle liste. Nous avons cependant un problème ici. En effet, pour obtenir une copie du « cœur », vu que « cœur » est de type « Mecanisme », nous avons utilisé le constructeur de copies de la classe « Mecanisme ». En réalité, « autre.cœur » correspond peut-être à un « MecanismeDouble ». Et dans ce cas, c'est bien un « MecanismeDouble » que nous souhaitons obtenir pour « cœur ». Le constructeur de copies de la classe « Mecanisme » peut-il produire un « MecanismeDouble » ? peut-il produire un « MecanismeDouble » ?

#### notes

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

#### résumé

5m 13s



.....

.....

.....

.....

.....