

Support de cours

Cours:

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

Vidéo:

### W17-05-copieprofonde-JAVA-pt3

Concepts (extraits des sous-titres générés automatiquement) :

**Valeur de base de l'objet. Copie polymorphique d'accessoires. Méthode de copie. Objet de type. Heure du mécanisme. Méthode abstraite. Sous-classe concrète d'accessoire. Constructeur de copie de la sous-classe. Méthode polymorphique de copie. Objet modifiable. Constructeur de copie de la classe. Biais de la méthode de copie. Copie. Moment de la création de l'objet. Objet courant.**



[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 3)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



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

Sinon, c'est le constructeur par défaut de la superclasse de « Montre »,

notes

résumé

0m 1s



```
ArrayList<Accessoire> accessoires;  
// ...  
accessoires.add(acc.copie());
```

```
abstract class Accessoire extends Produit {  
    //...  
    // copie polymorphique d'Accessoire  
    public abstract Accessoire copie();  
    //..  
}  
//-----  
class Bracelet extends Accessoire {  
    //..  
    public Bracelet(Bracelet autre) { super(autre); }  
    // copie polymorphique de Bracelet  
    @Override  
    public Bracelet copie(){  
        return new Bracelet(this);  
    }  
}
```

c'est-à-dire de la classe « produit », qui serait invoqué. Ce dernier initialise la valeur de base de l'objet au moyen de 0 ; ceci voudrait dire que toutes les montres obtenues par copie auraient une valeur de base nulle, ce qui n'est pas forcément souhaité. Voyons donc maintenant comment programmer une copie polymorphique d'accessoires. Un accessoire en tant que tel est une entité abstraite, on ne va pas définir de copie pour un accessoire à proprement parler, donc on se contente de définir la méthode de copie comme une méthode abstraite dans la classe « Accessoire », dont chaque sous-classe concrète d'accessoire, il va falloir définir de façon concrète la copie et la subtilité ici, est que la méthode polymorphique de copie peut utiliser dans la sous-classe le constructeur de copie de la sous-classe en question pour produire la copie.

notes

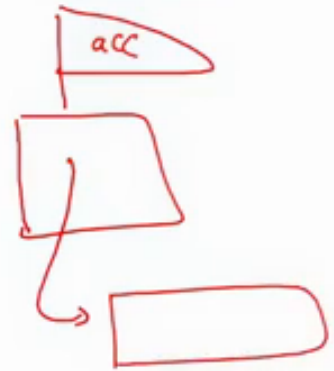
résumé

0m 6s



```
ArrayList<Accessoire> accessoires;
// ...
accessoires.add(acc.copie());
```

```
abstract class Accessoire extends Produit {
    //...
    // copie polymorphique d'Accessoire
    public abstract Accessoire copie();
    //...
}
//-----
class Bracelet extends Accessoire {
    //...
    public Bracelet(Bracelet autre) { super(autre); }
    // copie polymorphique de Bracelet
    @Override
    public Bracelet copie(){
        return new Bracelet(this);
    }
}
```



Supposons ici qu'un objet de type « Accessoire » soit copié par le biais de la méthode de copie, et qu'il se trouve qu'en réalité, cet objet correspond à un « Bracelet ». La situation en mémoire serait la suivante, nous avons l'objet que nous voulons copier,

notes

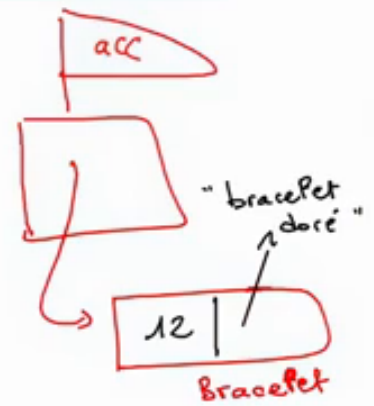
résumé

1m 1s



```
ArrayList<Accessoire> accessoires;
// ...
accessoires.add(acc.copie());
```

```
abstract class Accessoire extends Produit {
    //...
    // copie polymorphique d'Accessoire
    public abstract Accessoire copie();
    //...
}
//-----
class Bracelet extends Accessoire {
    //...
    public Bracelet(Bracelet autre) { super(autre); }
    // copie polymorphique de Bracelet
    @Override
    public Bracelet copie(){
        return new Bracelet(this);
    }
}
```



qui est une référence à un objet de type « Bracelet »... qui aurait donc par exemple une valeur de base, ainsi qu'un nom... alors lorsque nous appelons la méthode de copie l'objet courant est cet objet ; comme c'est un objet de type « Bracelet », nous allons chercher la méthode de copie dans la classe « Bracelet », laquelle va invoquer le constructeur de copie de la classe « Bracelet ». Ici nous copions un « Bracelet », donc cela ne pose aucun problème

notes

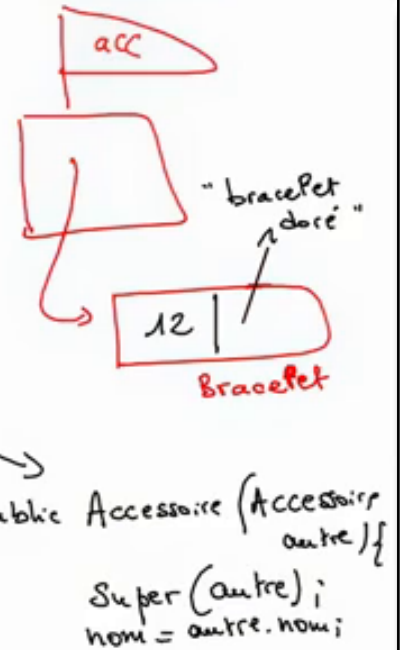
résumé

1m 21s



```
ArrayList<Accessoire> accessoires;
// ...
accessoires.add(acc.copie());
```

```
abstract class Accessoire extends Produit {
    //...
    // copie polymorphique d'Accessoire
    public abstract Accessoire copie();
    //...
}
//-----
class Bracelet extends Accessoire {
    //...
    public Bracelet(Bracelet autre) { super(autre); }
    // copie polymorphique de Bracelet
    @Override
    public Bracelet copie(){
        return new Bracelet(this);
    }
}
```



d'utiliser le constructeur de copie pour copier un « Bracelet » ; et ce dernier va donc simplement faire appel au constructeur des superclasses pour initialiser le nom hérité d'« Accessoire » et initialiser la valeur héritée de « Produit ». Ceci présuppose donc qu'il existe un constructeur de copies également dans la classe « Accessoire », qui peut se rédiger comme ceci par exemple : le constructeur de copies de la classe « Accessoire » prendrait en argument... évidemment, un autre accessoire, il ferait appel au constructeur de copies de la superclasse, qui serait donc le constructeur de copies de la classe « Produit » et il copierait des valeurs qui lui sont spécifiques...

notes

résumé

2m 1s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

sachant donc qu'un accessoire a comme attribut un nom. Ceci présuppose que dans la superclasse d'accessoires, c'est-à-dire la classe « Produit », il existe aussi un constructeur de copies. Ce dernier va simplement copier la valeur de base. On procéderait de façon analogue pour définir la copie polymorphique des « Mecanismes ». Enfin, un point qui aura peut-être attiré votre attention, si l'on examine l'en-tête de la méthode de copie dans la sous-classe « Bracelet », et son en-tête dans la superclasse « Accessoire », on se rend compte que les types de retours ne sont pas exactement les mêmes. Nous avons bel et bien affaire à une redéfinition de la méthode de copie héritée de la superclasse. Pourquoi ? Parce que les types de retours sont compatibles ; « Bracelet » est une sous-classe d'« Accessoire ». Techniquement, on appelle cela la « covariance des types de retours ». Pour terminer sur ce volet de la copie, revenons un peu sur le constructeur de la classe « Montre ». Nous l'avons initialement programmé de sorte à ce que...

#### notes

#### résumé

3m 1s





```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

il prenne en paramètre un « Mecanisme »

notes

résumé

4m 1s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

*coeur = depart;*

et que ce soit ce « Mecanisme » qu'on affecte directement au « coeur » de la montre. Si « Mecanisme » est un objet modifiable, ceci peut causer ce que l'on appelle des « failles d'encapsulation ».

notes

résumé

4m 3s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(--);

Imaginez par exemple que l'on ait un objet du type « Mecanisme »... proprement initialisé, comme ceci par exemple,

notes

résumé

4m 19s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Si l'on utilise ce « Mecanisme » pour créer une montre

notes

résumé

4m 37s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);

comme ceci, alors la variable « meca » et le « coeur » de la montre pointent sur le même objet en mémoire.

notes

résumé

4m 41s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);



Si « meca » est un objet modifiable, c'est-à-dire qu'il y a une méthode qu'on peut appliquer à cet objet pour le modifier, imaginez par exemple que j'aie une méthode de réglage qui permette de régler l'heure du mécanisme, alors en réglant l'heure du mécanisme, je vais aussi régler l'heure du cœur de la montre. En effet,

notes

résumé

5m 1s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);



la variable « meca » contient une référence vers un objet en mémoire

notes

résumé

5m 25s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);



qui serait un « MecanismeDouble »,

notes

résumé

5m 29s





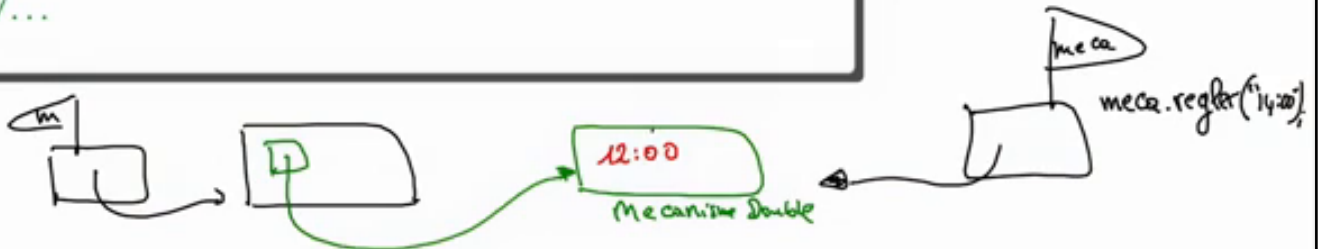
```

class Montre extends Produit {
    //...
    public Montre(Mecanisme depart)
    {
        coeur = depart.copie();
        accessoires = new ArrayList<Accessoire>();
    }
    //...
}

```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);



« m », qui est une montre, qui est une référence vers un objet de type « Montre », et dont le « coeur » serait la référence au même objet que celui pointé par « meca ». Supposons qu'au moment de la création de l'objet Montre, l'objet « meca » avait une heure qui était la suivante, et supposons que les « Mécanismes » soient dotés d'une méthode de réglage, donc qu'il soit possible par exemple de régler l'heure d'un « Mecanisme » en utilisant ce genre de tournures par exemple, alors cela voudrait dire que l'heure du « Mecanisme » est modifiée,

notes

résumé

5m 41s



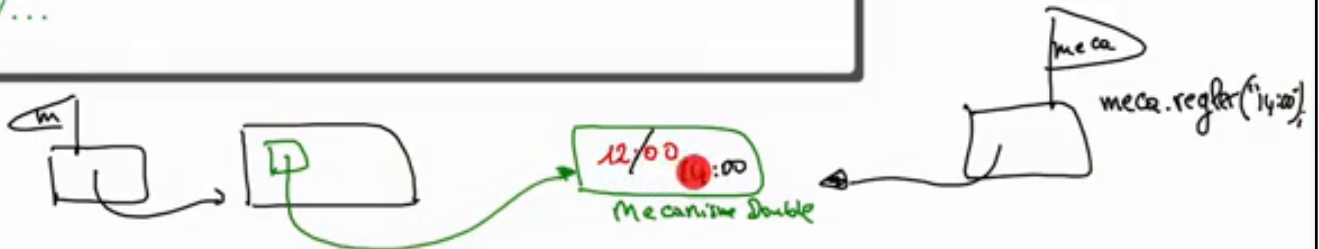
```

class Montre extends Produit {
    //...
    public Montre(Mecanisme depart)
    {
        coeur = depart.copie();
        accessoires = new ArrayList<Accessoire>();
    }
    //...
}

```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);



et cela voudrait dire aussi que l'heure de la montre est modifiée aussi.

notes

résumé

6m 23s



```
class Montre extends Produit {  
    //...  
    public Montre(Mecanisme depart)  
    {  
        coeur = depart.copie();  
        accessoires = new ArrayList<Accessoire>();  
    }  
    //...  
}
```

Mecanisme meca =  
new MecanismeDouble(...);

Montre m =  
new Montre(meca);

Il faut toujours être très attentif à ce point lorsqu'on écrit des constructeurs qui prennent en paramètre des objets, si ces objets sont modifiables, à moins qu'on ne veuille réellement les partager, c'est-à-dire que le comportement de partage soit réellement souhaité, alors il faut aussi penser à affecter une copie de l'objet passée en paramètre pour éviter des effets de bord indésirés, comme par exemple ici, le fait de pouvoir régler l'heure de la montre au travers d'un objet extérieur, ce qui constitue justement une faille d'encapsulation.

notes

résumé

6m 29s



```
class Montre extends Produit {  
  
    private Mecanisme coeur;  
    private ArrayList<Accessoire> accessoires;  
  
    public void ajouter(Accessoire accessoire) {  
        accessoires.add(accessoire); // nous reviendrons sur ce point  
    }  
}
```

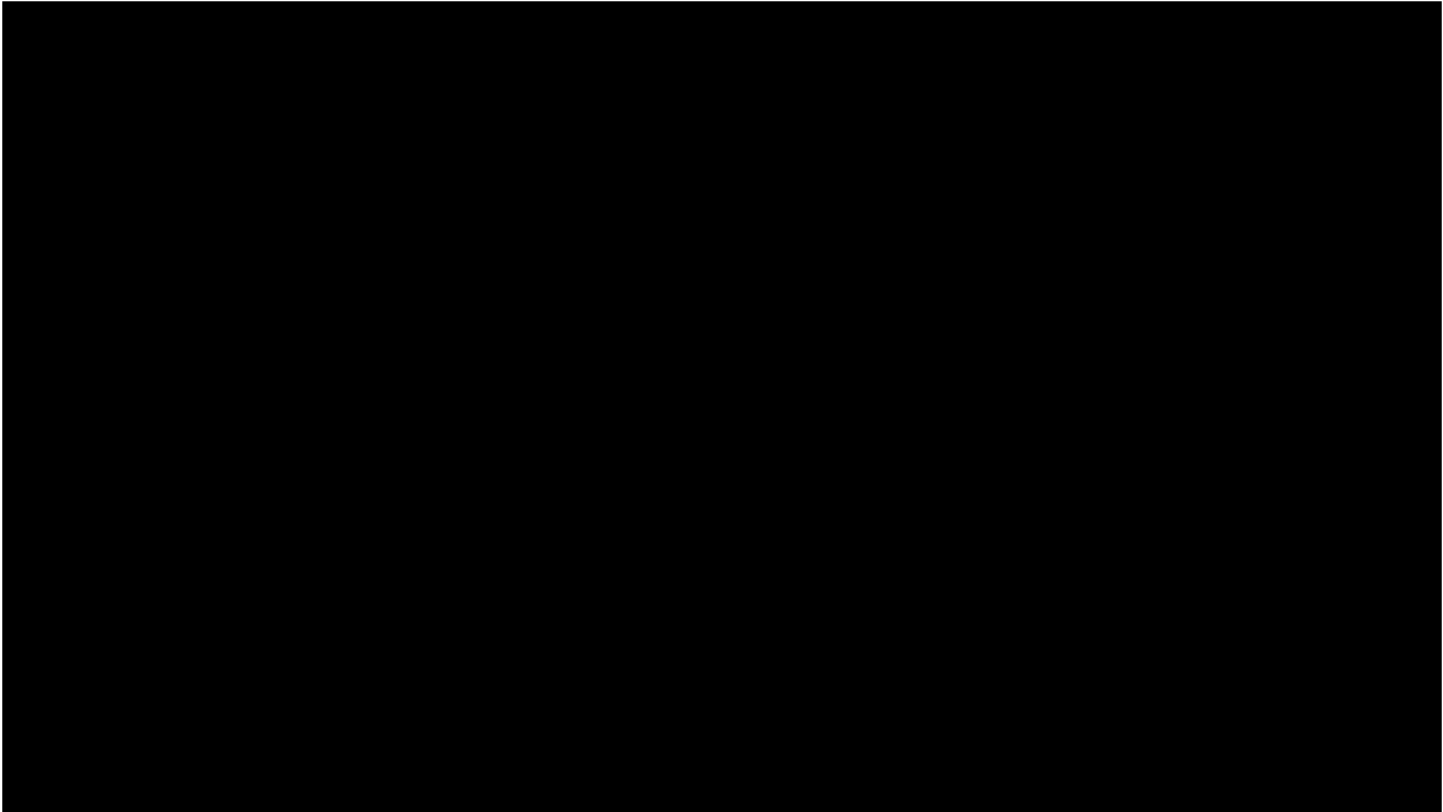
Donc ici, on utiliserait typiquement la copie polymorphe des « Mecanismes », pour obtenir un « coeur » indépendant du « Mecanisme » de départ et éviter ce genre d'effet de bord.

notes

résumé

7m 1s





Notez qu'un problème analogue se pose à nous au niveau de la méthode « ajouter() » permettant d'ajouter des accessoires à une montre. En effet, si la méthode « ajouter() » reçoit en paramètres la référence vers un accessoire, ici les accessoires sont modifiables, par exemple, il y aurait une méthode de réparation des « Accessoires » qui permettrait d'en modifier certaines valeurs, à ce moment-là, on aurait le même problème qui se pose à nous et il faudrait plutôt ajouter à notre collection d'accessoires... une copie de l'accessoire passée en paramètre de sorte à ce que les accessoires de notre montre ne dépendent pas d'un accessoire fourni depuis l'extérieur, et modifiable. depuis l'extérieur, et modifiable.

notes

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

résumé

7m 12s

