

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W17-03-premiereversion-JAVA-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Classe produit. Séquences vidéos précédentes. Classe montre. Valeur de base du prix. Méthode toString. Tableau dynamique d'accessoires. Instance de la classe. Nouveau bracelet. Séquence vidéo. Nom d'un fermoir. Collection d'accessoires. Second paramètre. Accessoires. Prix des accessoires. Prix d'un produit usuel.



[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 : première version

(Partie 1)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Dans les séquences vidéos précédentes,

notes

résumé

0m 1s



Rappel :

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

On souhaite ajouter des accessoires .

Par exemple :

```
montre.ajouter(new Bracelet...));
```

nous vous avons présenté le problème général de l'étude de cas à laquelle nous nous intéressons ici, à savoir modéliser des montres avec différents mécanismes, différents accessoires. Puis nous avons abordé le problème plus spécifique de l'affichage polymorphique. Nous avons aussi complété la classe Produit. Dans cette séquence vidéo-ci, nous allons construire une première version opérationnelle de notre code, en ajoutant des accessoires aux montres et en définissant donc, justement, plusieurs de ces accessoires. Commençons donc par l'ajout d'accessoires à la classe Montre qui, je vous le rappelle, héritait de Produit et puis avait un Mécanisme et surtout, ce qui va nous intéresser ici, un tableau dynamique d'accessoires que l'on a appelé ici « Accessoire ». Et l'idée c'est donc que l'on puisse rajouter des éléments à ce tableau dynamique en faisant par exemple pour une montre, instance de la classe montre Montre.ajouter, par exemple, « new Bracelet »

notes

résumé

0m 6s



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

où je vous rappelle que « Bracelet » est une sous-classe d'Accessoire. Et donc on voudrait que ceci rajoute un nouveau bracelet à la collection d'accessoires, un « Bracelet » étant un « Accessoire » qui peut être mis dans une « collection d'Accessoires ».

notes

résumé

1m 1s





Voyons maintenant comment définir cette méthode « ajouter ». Ici, nous allons faire une solution très très simple sur laquelle nous reviendrons plus tard, dans la dernière vidéo de cette série de séquences vidéos, qui abordera donc la notion de « copie profonde ». Mais pour l'instant, on fait simplement une copie de surface, ici, en ajoutant simplement la référence vers un accessoire que l'on a reçu au tableau dynamique d'accessoires qui s'appelle « accessoires », au pluriel. Essayons maintenant de faire

notes

résumé

1m 16s



Essayons maintenant d'avoir une première version opérationnelle de notre code, pour le moment :

- ▶ *sans* tous les mécanismes
- ▶ *sans* copie des montres

Pour cela, il nous faut encore :

- ▶ quelques *accessoires*
- ▶ terminer la classe `Montre`
- ▶ un exemple d'utilisation dans une méthode `main()`

une première version opérationnelle du programme. Essayons donc de finaliser ce que nous avons écrit jusqu'ici. Et pour le moment, sans absolument aucun mécanisme et sans faire de copie de montre, nous laissons ces deux points pour les deux prochaines séquences vidéos.

notes

résumé

1m 45s



Décidons par exemple que
les accessoires :

```
abstract class Accessoire extends Produit {  
  
}
```

Donc, pour pouvoir finaliser le programme que nous avons jusqu'à maintenant, il nous faudrait quand même quelques accessoires un petit plus concrets. Il faudrait terminer la classe Montre qui pour l'instant n'est pas encore tout à fait opérationnelle et puis, bien sûr, utiliser tout ceci dans un « main » typique. Nous allons reprendre ces points les uns après les autres à commencer par quelques accessoires.

notes

résumé

2m 1s



Décidons par exemple que les accessoires :

- ▶ ont un nom et une valeur de base fixés au départ (sans valeur par défaut)
- ▶ s'affichent en indiquant leur nom et leur prix

```
abstract class Accessoire extends Produit {
    private final String nom;

    public Accessoire(String unNom,
                      double valeurDeBase) {
        super(valeurDeBase);
        nom = unNom;
    }

    @Override
    public String toString() {
        String result = nom + " coûtant ";
        result += super.toString();
        return result;
    }
}
```

Pour rappel, nous avons une classe « Accessoire » qui est un « Produit ». Et décidons, par exemple, que nous ayons au niveau des accessoires un nom pour pouvoir les désigner. Pour ceci, on va dire que l'on a un nom qui ne changera pas une fois qu'il a été fixé par le constructeur. Et nous allons donc ajouter un constructeur à la classe Accessoire qui prendra donc un nom pour pouvoir nommer cet accessoire. Et puis, il ne faut pas oublier que tout produit peut être vendu et a donc un prix et recevra un second paramètre ici, pour pouvoir initialiser la valeur de base du prix du produit. Le constructeur d'Accessoire va commencer par appeler le constructeur de la super-classe Produit, en passant ici, le paramètre valeurDeBase que l'on a reçu. Et puis ensuite, on initialisera l'attribut « nom » avec le nom reçu. Voilà donc pour le constructeur que l'on a décidé de donner à tous les accessoires. Et décidons maintenant d'une façon d'afficher les accessoires, car je vous rappelle que tous les produits sont capables de s'afficher de façon polymorphique. Nous allons donc, ici, redéfinir la méthode toString que nous avons hérité de Produit. On dit qu'on l'a « redéfini » en ajoutant la notation, ici, « override » et décidons que, par exemple, il s'affiche en indiquant le nom et leur coût. Donc on va afficher simplement le nom puis on va indiquer qu'il coûte et nous savons déjà afficher le prix en appelant directement la méthode de la super-classe Produit. Donc ici on utilise le mot-clé « super » pour appeler la méthode héritée de la super-classe Produit. Parce que, bien sûr, si l'on écrit toString tout seul sans démasquage ici, ce sera la méthode toString de la classe en question, et on aurait une récursion infinie : ToString s'appellerait elle-même et donc pour,

notes

résumé

2m 19s



Décidons par exemple que les accessoires :

- ▶ ont un nom et une valeur de base fixés au départ (sans valeur par défaut)
- ▶ s'affichent en indiquant leur nom et leur prix

```
abstract class Accessoire extends Produit {  
    private final String nom;  
  
    public Accessoire(String unNom,  
                      double valeurDeBase) {  
        super(valeurDeBase);  
        nom = unNom;  
    }  
  
    @Override  
    public String toString() {  
        String result = nom + " coûtant ";  
        result += super.toString();  
        return result;  
    }  
}
```

dans la méthode toString

notes

résumé

```
class Bracelet extends Accessoire {
    public Bracelet(String unNom, double valeurDeBase) {
        super("bracelet " + unNom, valeurDeBase);
    }
}

//-----

class Fermeture extends Accessoire {
    public Fermeture(String unNom, double valeurDeBase) {
        super("fermeture " + unNom, valeurDeBase);
    }
}

//...
```

de la classe Accessoire, appeler la méthode toString, héritée de la super-classe Produit, il faut ici donc démasquer cette méthode toString. Décidons enfin que le prix des accessoires est le « même » que le prix d'un produit usuel, c'est-à-dire un produit général tel que nous l'avons conçu tout en haut de la hiérarchie, et donc à ce moment-là, là, nous n'avons rien du tout à faire puisque la méthode Prix que nous avons hérité de la classe Produit suffit ici, et donc on n'a pas besoin de la redéfinir dans la classe Accessoire. Nous avons donc ici une classe Accessoire tout à fait opérationnelle et qui correspond à ce que nous souhaitons. Définissons donc maintenant quelques accessoires, donc, par exemple, un « Bracelet ». Un « Bracelet » est un « Accessoire » donc on va avoir ici une relation d'héritage. Au niveau des accessoires, admettons que l'on souhaite que leur nom -- je vous rappelle que les accessoires ont un nom, donc tous les accessoires vont hériter ce nom -- admettons que l'on souhaite

notes

résumé

4m 13s





que chacun des accessoires ait, comme ça, un nom qui marque effectivement ce que c'est, donc par exemple, le nom d'un bracelet va commencer par « bracelet », le nom d'un fermoir va, par exemple, commencer par « fermoir ». Et donc on va forcer ceci au niveau du constructeur donc, par exemple, pour la sous-classe Bracelet de la classe Accessoire, on recevra donc le nom, qui sera le complément qu'on va rajouter ici, derrière le bracelet. Donc, par exemple, « en cuir » pour que le nom complet soit « bracelet en cuir » et puis comme deuxième paramètre le prix, que nous passons aussi au constructeur de la super-classe. Donc, de même, on pourrait définir un Fermoir qui hérite d'Accessoire dans lequel on définit un constructeur qui prend un complément du nom « fermoir » et qui prend un prix pour pouvoir appeler le constructeur de la super-classe Accessoire où le nom complet sera donc « fermoir » avec ce complément que l'on a reçu comme premier paramètre et comme deuxième paramètre le prix que nous avons reçu avec toujours le destructeur virtuel.

notes

résumé

5m 13s



Pour finaliser la classe `Montre`
(mais sans mécanisme) :

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

Voilà, nous en avons donc ici terminé avec les accessoires. Essayons aussi de rendre la classe `Montre` utilisable sans les mécanismes et sans la copie toujours, que nous laissons de côté pour le moment. Nous avons à ce stade donc simplement défini le contenu de la classe `Montre`

notes

résumé

6m 12s



Pour finaliser la classe `Montre` (mais sans mécanisme) :

- ▶ contentons nous pour l'heure d'un constructeur par défaut
- ▶ décidons d'un calcul de prix : somme des prix des accessoires

```
// ...
@Override
public double prix() {
    // Au départ, le prix est le prix de base
    double prixFinal = super.prix();

    for (Accessoire acc : accessoires) {
        prixFinal += acc.prix();
    }
    return prixFinal;
}
//..
}
```

dans laquelle nous avons donc cette méthode « ajouter », qui nous permet de rajouter des accessoires à nos montres. Commençons donc par munir notre classe `Montre` d'un constructeur. Pour l'instant, contentons-nous d'un constructeur par défaut que nous redéfinissons quand même car le constructeur « par défaut par défaut » rendrait nulle cette référence, alors que nous, nous préférons avoir ici un tableau dynamique vide que nous pourrions ensuite remplir avec notre méthode `ajouter`. Voilà pour la construction des montres. Décidons ensuite du calcul du prix des montres et admettons que ce soit la somme des prix de ses accessoires. Donc pour ceci nous allons redéfinir la méthode « `prix` » : « `override` » cette méthode « `prix` » héritée de la super-classe `Produit`. Donc nous allons décider que, au départ, le prix d'une montre c'est le prix de base que nous récupérerons au travers de la méthode « `prix` » d'origine, héritée de la super-classe `Produit`.

notes

résumé

6m 26s



Puis, à ce prix de départ, nous allons rajouter l'ensemble des prix des différents accessoires. Donc, pour ceci, nous parcourons la liste des accessoires, et nous rajoutons au prix que nous avons au départ initialisé à chaque fois le prix, c'est-à-dire l'appel à la méthode « prix », et enfin donc, on retourne le prix que l'on vient de calculer. Décidons enfin comment afficher les montres. Par exemple, nous décidons d'avoir un message pour dire qu'une montre est composée, ensuite de la liste de ses différents accessoires, etc. et puis, à la fin, on affiche le prix total de la montre. Pour cela, on va par exemple ajouter une méthode afficher, que l'on va définir en commençant donc l'affichage par : « une montre composée de » et ensuite en parcourant l'ensemble des accessoires de la montre. Pour chacun des accessoires, on affiche cette petite étoile et puis on affiche, ici, l'accessoire en question. l'accessoire en question.

7m 25s

