

Support de cours

Cours:

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

Vidéo:

### W17-04-interface-JAVA-pt2

Concepts (extraits des sous-titres générés automatiquement) :

**Méthode d'affichage générale. Classe mecanisme. Fonction des sous-classes. Affichage du cadran. Classe montre. Heure du réveil. Constructeur de mecanismeanalogique. Mécanismes digitaux. Méthode d'affichage du cadran. Sous-classe mecanismedigital. Construction des mécanismes. Super-classe. Classe mecanismedouble. Code java. Nombre d'éléments propres.**



[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 : modélisation des mécanismes

(Partie 2Z)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



Pour le moment, il n'y a qu'un constructeur par défaut, mais fixons la construction des `Mecanisme` :

- ▶ initialisation de la valeur de base (`Produit`)
- ▶ initialisation de l'heure

```
class Mecanisme extends Produit {  
    private String heure;  
  
    public Mecanisme(double valeurDeBase, String uneHeure) {  
        super(valeurDeBase);  
        heure = uneHeure;  
    }  
    public Mecanisme(double valeurDeBase){  
        super(valeurDeBase);  
        heure = "12:00";  
    }  
}
```

Le constructeur de `MecanismeAnalogique`

notes

résumé

0m 1s



Complétons maintenant nos constructeurs pour la classe `Montre` :

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

va en effet invoquer ce constructeur de la super-classe `Mecanisme` lequel va initialiser proprement l'heure par défaut.

notes

résumé

0m 6s



Supposons que l'on veuille :

- ▶ que tous les mécanismes s'affichent suivant le **même** schéma, **imposé** et non modifiable

Par exemple :

*méthodes* *polymorphe*  
**affichage du type de mécanisme**, suivi d'un **affichage du cadran** (heure, date, heure de réveil, ...), suivi du prix

- ▶ mais que chaque **partie** de ce schéma soit adaptable
- ▶ offrir une version par défaut, utilisable dans les sous-classes, de l'affichage du cadran (par exemple affichage de l'heure)
- ▶ imposer la redéfinition de l'affichage du type de mécanisme

🔊 Comment faire ?

notes

résumé

0m 15s



Supposons que l'on veuille :

- ▶ que tous les mécanismes s'affichent suivant le **même** schéma, **imposé** et non modifiable

Par exemple :

**affichage du type de mécanisme**, suivi d'un **affichage du cadran** (heure, date, heure de réveil, ...), suivi du prix

*méthodes* *polymorphe*

- ▶ mais que chaque **partie** de ce schéma soit adaptable
- ▶ offrir une version par défaut, utilisable dans les sous-classes, de l'affichage du cadran (par exemple affichage de l'heure)
- ▶ imposer la redéfinition de l'affichage du type de mécanisme

☞ Comment faire ?

s'adapter automatiquement à la nature réelle des objets auxquels elle s'applique. Le fait que le même schéma de base

notes

résumé

```
abstract class Mecanisme extends Produit {
    //...
    // Tous les mécanismes DOIVENT s'afficher comme ceci
    public final String toString() {
        String result = "mécanisme ";
        result += toStringType();
        result += " (affichage : ";
        result += toStringCadran();
        result += ")", prix : ";
        result += super.toString();
        return result;
    }
    // on veut offrir la version par défaut aux sous-classes et aux classes
    // du même paquetage. Par défaut, on affiche juste l'heure.
    protected String toStringCadran() {
        return heure;
    }
    // Un mécanisme, ici à ce niveau, est abstrait (= classe abstraite)
    protected abstract String toStringType();
}
```

soit imposé à tous pour les mécanismes sous-entend qu'une fois cette méthode adhérent à ce schéma fixé, il ne faut plus qu'elle soit modifiée. Ce qui devrait nous faire penser à des méthodes finales. Nous voulons également faire en sorte qu'il existe une version utilisable dans les sous-classes de l'affichage du cadran. Donc cette méthode devrait avoir une définition par défaut, typiquement tout en haut de la hiérarchie, dans la classe Mecanisme. Cette version par défaut pourrait par exemple simplement afficher l'heure et être utilisée dans les sous-classes pour justement afficher l'heure et éventuellement d'autres informations. Donc, ici nous nous dirigeons vers une méthode qui serait, pour l'affichage du cadran, polymorphique mais qui aurait une définition par défaut dans la super-classe. Par contre, pour l'affichage du type de mécanisme, nous voulons impérativement en imposer la redéfinition dans les sous-classes ce qui naturellement devrait nous faire penser ici aux méthodes abstraites. Voici comment on pourrait traduire ces contraintes en code Java. Notre super-classe Mecanisme offre une méthode d'affichage polymorphique qui redéfinit celle héritée de Produit laquelle affichait le prix ; notre super-classe Mecanisme offre donc une méthode d'affichage qui obéit à un schéma précis lequel affichera le type, le cadran et le prix par le biais de la méthode héritée de Produit. Pour faire en sorte que ce schéma soit fixé une fois pour toute et ne soit pas redéfinissable dans une sous-classe de la hiérarchie, nous marquons la méthode comme étant " finale ". Les sous-classes de Mecanisme n'auront donc plus la possibilité de redéfinir la méthode toString. Elle est déclarée comme final. En revanche, elles auront tout à fait le loisir de redéfinir des parties de ce schéma dont, par exemple, l'affichage du cadran ou l'affichage du type sont elles-mêmes des méthodes polymorphiques qui pourront s'adapter aux types des

## notes

## résumé

2m 25s



```
abstract class Mecanisme extends Produit {
    //...
    // Tous les mécanismes DOIVENT s'afficher comme ceci
    public final String toString() {
        String result = "mécanisme ";
        result += toStringType();
        result += " (affichage : ";
        result += toStringCadran();
        result += ")", prix : ";
        result += super.toString();
        return result;
    }
    // on veut offrir la version par défaut aux sous-classes et aux classes
    // du même paquetage. Par défaut, on affiche juste l'heure.
    protected String toStringCadran() {
        return heure;
    }
    // Un mécanisme, ici à ce niveau, est abstrait (= classe abstraite)
    protected abstract String toStringType();
}
```

mécanismes auxquels elles s'appliquent. La méthode permettant d'afficher le cadran conformément au souhait qui était formulé tout à l'heure a une définition par défaut dans la classe Mecanisme. Elle permet à ce moment-là juste d'afficher l'heure. Nous avons décidé de la déclarer en protégé pour permettre aux sous-classes d'utiliser cette méthode de la super-classe.

notes

résumé



```
class MecanismeDigital extends Mecanisme {  
    //..  
    @Override  
    protected String toStringType() {  
        return "digital";  
    }  
  
    @Override  
    protected String toStringCadran() {  
        // On affiche l'heure (façon de base)...  
        // ...et en plus l'heure de réveil.  
        return super.toStringCadran() + ", " + toStringReveil();  
    }  
  
    protected String toStringReveil() {  
        return " réveil " + reveil;  
    }  
}
```

Il n'y a pas de risque ici au niveau de l'encapsulation car la méthode `toStringCadran`, permettant de générer une représentation du cadran, ne modifie pas, n'altère pas les détails internes du mécanisme. On considère par contre que la méthode permettant d'afficher le type du mécanisme n'est pas vraiment définissable à ce stade de la hiérarchie. Il s'agit donc d'une méthode qui va être définie comme étant abstraite. Le fait de déclarer cette méthode abstraite va en imposer la définition concrète dans toutes les sous-classes de `Mecanisme` qu'on souhaite pouvoir instancier plus tard. Toute sous-classe de la classe `Mecanisme` que l'on souhaite pouvoir instancier, comme ici par exemple la sous-classe `MecanismeDigital`, devra impérativement donner une définition concrète de la méthode permettant d'afficher le type, ce qui est le cas ici. La sous-classe `MecanismeDigital` peut bien évidemment, aussi, redéfinir la méthode permettant l'affichage du cadran. Et elle peut redéfinir cette méthode en utilisant la méthode héritée de plus haut, c'est-à-dire celle permettant d'afficher l'heure. Donc, ici l'affichage du cadran d'un mécanisme digital affichera l'heure ainsi que le réveil. L'affichage de réveil se fait aussi par le biais d'une méthode

## notes

## résumé

4m 49s



```
class MecanismeDouble extends MecanismeAnalogique {
    //...
    @Override
    protected String toStringCadran() {
        // Par exemple...
        String result = "sur l'écran : ";
        result += super.toStringCadran();
        result += ", sous les aiguilles : ";
        result += toStringReveil();
        return result;
    }
    @Override
    protected String toStringType() {
        return "double";
    }
    // propres aux mécanismes digitaux
    protected String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

mais cette fois cette méthode est spécifique à la classe MecanismeDigital. Elle affiche simplement le libellé réveil suivi de l'heure du réveil. La classe MecanismeDouble,

notes

résumé

6m 1s



```
class MecanismeDouble extends MecanismeAnalogique {  
    //...  
    @Override  
    protected String toStringCadran() {  
        // Par exemple...  
        String result = "sur l'écran : ";  
        result += super.toStringCadran();  
        result += ", sous les aiguilles : ";  
        result += toStringReveil();  
        return result;  
    }  
    @Override  
    protected String toStringType() {  
        return "double";  
    }  
    // propres aux mécanismes digitaux  
    protected String toStringReveil() {  
        return " réveil " + reveil;  
    }  
}
```

si l'on veut pouvoir l'instancier, ce qui est le cas, doit elle aussi impérativement redéfinir la méthode permettant l'affichage multiple. Puisqu'elle dispose d'un réveil dont on veut probablement aussi

notes

résumé

6m 8s



```
class MecanismeDigital extends Mecanisme {
    //...
    @Override
    protected String toStringType() {
        return "digital";
    }

    @Override
    protected String toStringCadran() {
        // On affiche l'heure (façon de base)...
        // ...et en plus l'heure de réveil.
        return super.toStringCadran() + ", " + toStringReveil();
    }

    protected String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

vouloir afficher la valeur, il faut qu'on définisse également la méthode permettant d'afficher l'heure du réveil de la même manière qu'on l'a faite pour les mécanismes digitaux. Et on peut imaginer de redéfinir la méthode permettant l'affichage du cadran, permettant donc d'afficher les informations du cadran hérités de MecanismeAnalogique, c'est-à-dire concrètement ici l'heure et la date. Et l'on afficherait ensuite les informations relatives au réveil par le biais de cette méthode spécifique. On voit donc qu'aussi bien la classe mécanisme double

notes

résumé

6m 21s



Il serait bien :

- ▶ d'imposer aux mécanismes ayant (aussi) un comportement digital d'implémenter la méthode `toStringReveil`;
- ▶ d'expliciter le lien entre les mécanismes ayant un comportement digital.

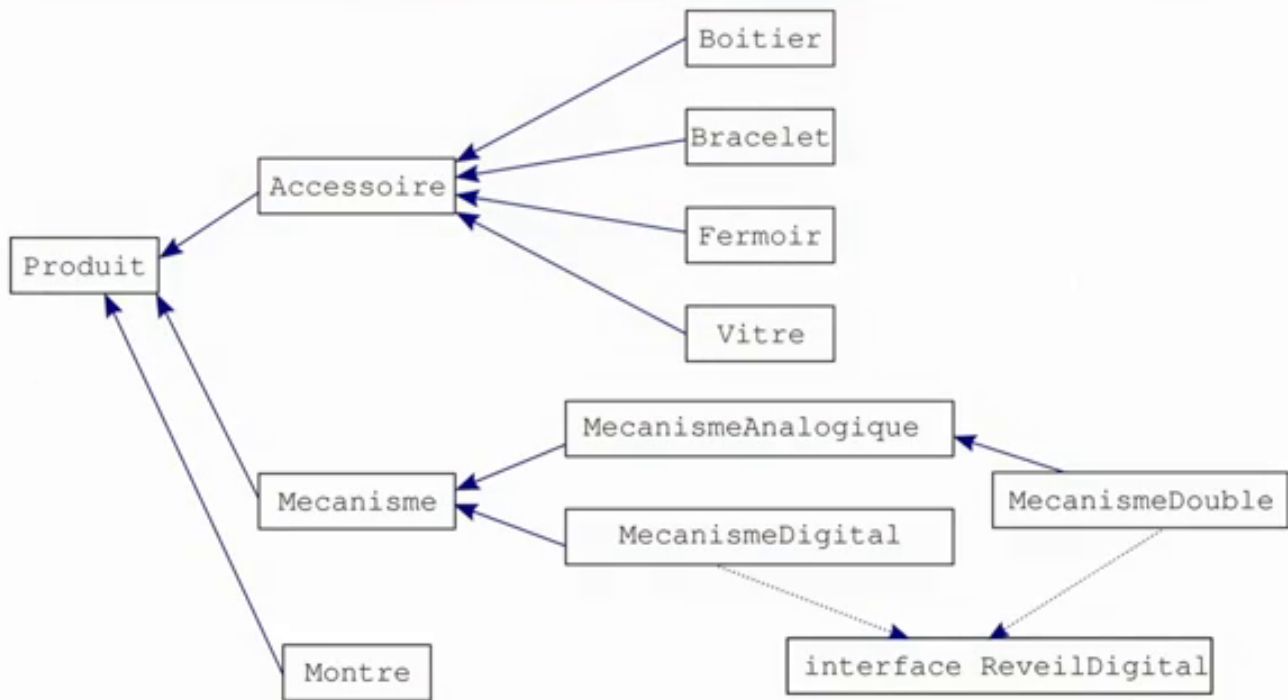
que la classe `MecanismeDigital` contiennent une méthode permettant d'afficher les informations relatives au réveil et le font de façon tout à fait analogue. Cependant, il n'existe aucun lien entre les deux. Il serait donc bien ici d'expliciter le lien entre les mécanismes ayant un comportement digital, c'est-à-dire `MecanismeDouble` et `MecanismeDigital` dans notre cas,

notes

résumé

6m 54s





et d'imposer aux mécanismes qui ont aussi un comportement digital d'implémenter les méthodes communes comme par exemple la méthode toStringRéveil permettant d'afficher les informations relatives au réveil.

notes

résumé

7m 13s



```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

Et ceci nous conduit à une nouvelle révision de notre hiérarchie de classe. Nous pouvons établir le lien entre les mécanismes digitaux et les mécanismes doubles par le biais d'une interface, laquelle imposerait aux sous-classes qui l'implémentent d'implémenter la méthode permettant d'afficher les informations relatives au réveil.

notes

résumé

7m 25s



```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

Nous pouvons donc introduire l'interface ReveilDigital et imposer le fait que toutes les classes qui implémenteraient des digital fournissent une méthode permettant d'afficher les informations relatives au réveil. Notre classe MecanismeDigital implémenterait cette interface.

notes

résumé

7m 43s





```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //..
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

Il en sera de même pour la classe MecanismeDouble et ces deux classes, si l'on veut pouvoir les instancier, doivent impérativement fournir une implémentation concrète de la même méthode, la méthode permettant d'afficher les informations relatives au réveil. Dans cette conception la classe MecanismeDouble est avant tout un mécanisme analogique qui contient un certain nombre d'éléments propres aux mécanismes digitaux, que l'on retrouve d'ailleurs dans la classe MecanismeDigital.

notes

résumé

7m 59s



```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

Mais elle est tenue d'implémenter ces éléments propres

notes

résumé

8m 25s



```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //..
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

aux mécanismes digitaux du fait qu'elle implémente l'interface ReveilDigital. Vous noterez ici que le fait d'avoir introduit une interface

notes

résumé

8m 31s



```
//=====
interface ReveilDigital
{
    String toStringReveil();
}
//=====
class MecanismeDigital extends Mecanisme implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
//=====
class MecanismeDouble extends MecanismeAnalogique implements ReveilDigital {
    //...
    // propres aux mécanismes digitaux
    public String toStringReveil() {
        return " réveil " + reveil;
    }
}
```

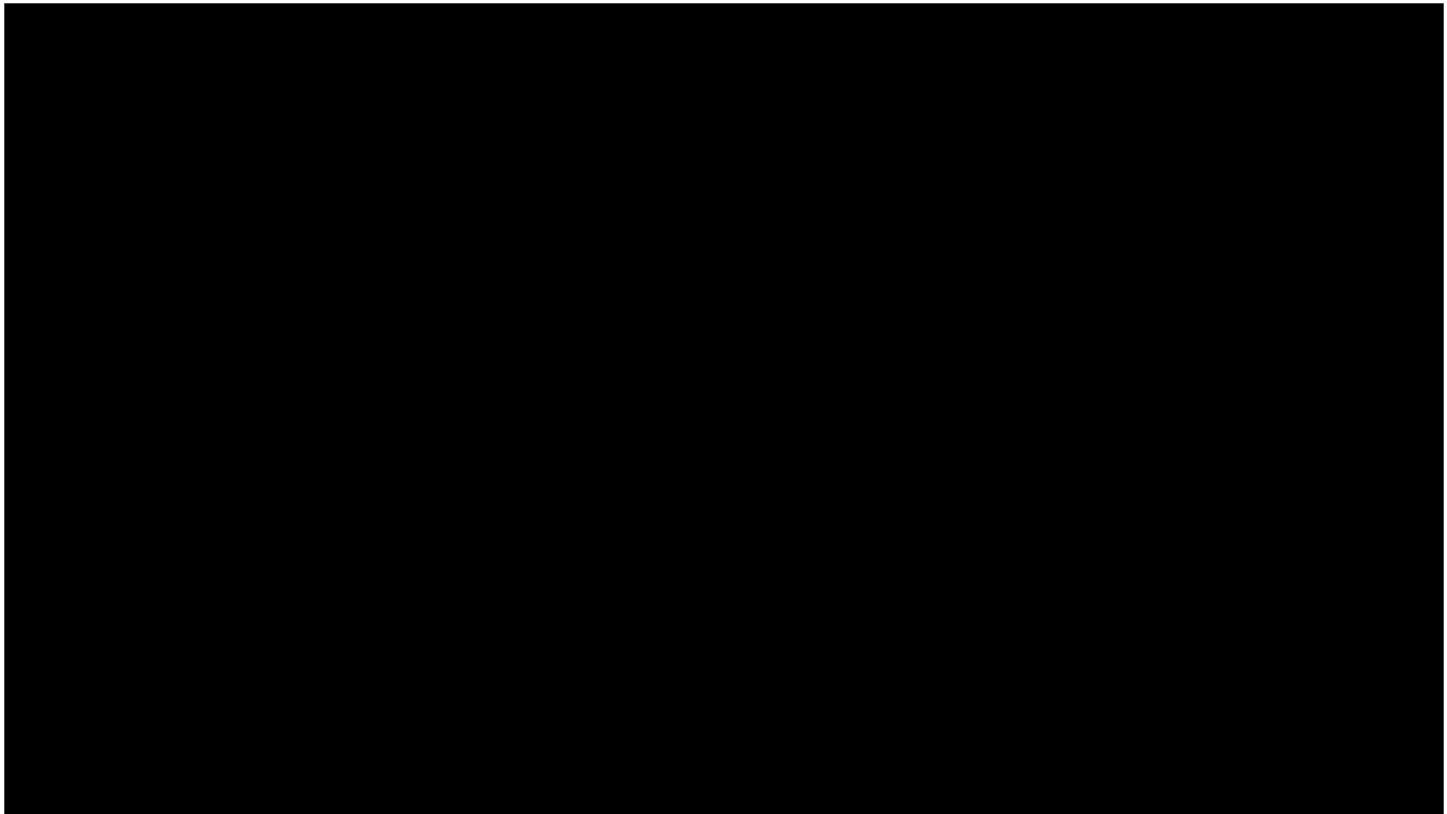
rend la modélisation plus claire, plus évidente mais ne nous épargne pas les duplications de code liés à l'absence d'héritage multiple à proprement parler en Java. Si ces duplications de code devaient être trop nombreuses,

notes

résumé

8m 38s





il est clair que la modélisation proposée sur la base de l'héritage devrait être discutée pour probablement se diriger vers une modélisation utilisant l'encapsulation. Par exemple, on pourrait imaginer l'introduction d'une classe `FonctionnaliteDigitale` regroupant les éléments communs aux fonctionnalités digitales justement, et dont chacune des classes `MecanismeDigital` et `MecanismeDouble` auraient une instance en guise d'attribut, par exemple. Si vous avez été attentifs à l'évolution du code dans les transparents qui ont précédé, vous aurez peut-être remarqué qu'il y a eu une modification des droits d'accès de la méthode `toStringRéveil`. Sauriez-vous dire pourquoi ? Sauriez-vous dire pourquoi ?

#### notes

---

---

---

---

---

---

---

---

---

---

#### résumé

8m 50s



---

---

---

---

---