

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W14-01-polymabstract-JAVA-pt1

Concepts (extraits des sous-titres générés automatiquement) :

Instances d'une sous-classe. Bonne méthode. Figures géométriques fermées. Résolution dynamique des liens. Fonction de sous-classes spécifiques. Première solution. Classe générale. Bon complément de l'abstraction. Méthodes. Séquence vidéo. Mauvaise idée. Premier argument. Propriétés propres. Certaine hauteur. Niveau de cette classe.



[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/>

Classes et méthodes abstraites

(Partie 1)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Dans cette séquence vidéo nous allons voir

notes

résumé

0m 1s

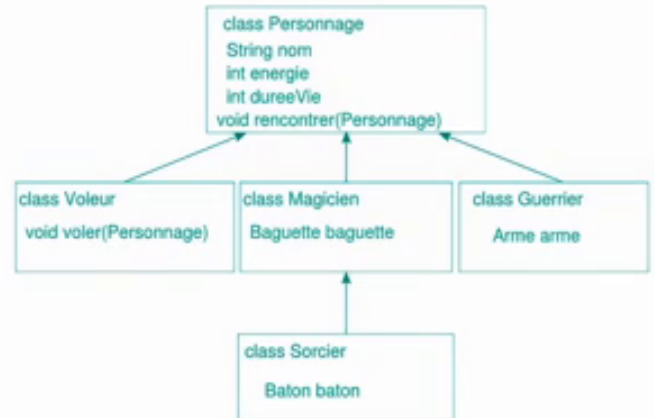


En POO, le **polymorphisme** (universel d'inclusion) est le fait que les instances d'une sous-classe, lesquelles sont *substituables* aux instances des classes de leur ascendance (en argument d'une méthode, lors d'affectations), **gardent leurs propriétés propres**.

- Le choix des méthodes à invoquer se fait *lors de l'exécution du programme* en fonction de la *nature réelle des instances* concernées.

La mise en œuvre se fait au travers de :

- l'héritage (hiérarchies de classes) ;
- la **résolution dynamique des liens**.



```
Personnage p1, p2;
// ...
p1.rencontrer(p2);
```

comment le polymorphisme est un très bon complément de l'abstraction et nous permet donc de mieux spécifier les classes au niveau le plus abstrait. Mais commençons par quelques rappels sur le polymorphisme. Le « polymorphisme universel d'inclusion » pour être précis, est le fait que des instances d'une sous-classe, qui sont « substituables » aux instances des super-classes, gardent leurs propriétés propres au lieu d'être réduites aux propriétés de la super-classe et en particulier leurs comportements, c'est à dire, les méthodes invoquées vont être déterminées lors de l'exécution du programme en fonction de la nature réelle des instances concernées. La définition peut paraître encore compliquée mais le principe est en fait assez simple par exemple, si j'ai une classe générale « Personnage » dont hérite « Voleur », un voleur est un personnage, « Magicien », « Guerrier », etc. Les magiciens, guerriers et voleurs vont,

notes

résumé

0m 5s



```
class Personnage {
    // ...
    public
    void rencontrer(Personnage p) {
        System.out.print("Bonjour !");
    }
}

class Guerrier {
    // ...
    public
    void rencontrer(Personnage p) {
        System.out.print("Boum !");
    }
}
```

```
class Rencontre
{
    public static void main(...) {
        Guerrier g = new Guerrier(...);
        Voleur v = new Voleur(...);
        uneRencontre(g, v);
    }
    static void uneRencontre(Personnage a,
                              Personnage b) {
        System.out.print(a.getNom());
        System.out.print(" rencontre ");
        System.out.print(b.getNom() + " : " );
        a.rencontrer(b);
    }
}
```

même si ils sont vus en tant que personnages, pouvoir se comporter, donc par exemple si ils rencontrent, en tant que chacun des instances propres. Par exemple, si un voleur rencontre un autre personnage il va peut-être le voler, un guerrier si il rencontre un autre personnage il va peut-être le frapper, etc. Donc par exemple si j'ai comme ça un personnage « p1 » qui rencontre un personnage « p2 » la méthode « rencontrer » ici, va s'adapter à la nature réelle de l'instance de « p1 » et donc va rencontrer « p2 » comme un voleur si « p1 » est un voleur, comme un magicien si « p1 » est un magicien, comme un guerrier si « p1 » est un guerrier. Et je vous rappelle que pour avoir du polymorphisme il faut bien sûr de l'héritage pour avoir cette classe générale par rapport à laquelle les traitements doivent s'adapter en fonction de sous-classes spécifiques donc de l'héritage et de la résolution dynamique des liens. Je vous rappelle aussi que la résolution dynamique des liens ça consiste à appeler la bonne méthode en fonction de l'instance, et donc pour reprendre un exemple qui vous avez été déjà donné dans une séquence vidéo précédente, imaginons qu'on organise une rencontre entre un guerrier et un voleur, cette méthode donc ici, organise une rencontre entre deux personnages reçu en argument en affichant un message et en appelant la méthode « rencontrer » du « Personnage a », reçu comme premier argument ici sur le « Personnage b » reçu comme deuxième argument. Et la résolution dynamique des liens va faire que lorsque l'on rencontre ici un guerrier « g » comme paramètre « a », c'est la méthode « rencontrer » de guerrier qui, dynamiquement, lors de l'exécution de ce bout de code, va regarder

notes

résumé

1m 1s



```
class Personnage {
    // ...
    public
    void rencontrer(Personnage p) {
        System.out.print("Bonjour !");
    }
}

class Guerrier {
    // ...
    public
    void rencontrer(Personnage p) {
        System.out.print("Boum !");
    }
}
```

```
class Rencontre
{
    public static void main(...) {
        Guerrier g = new Guerrier(...);
        Voleur v = new Voleur(...);
        uneRencontre(g, v);
    }
    static void uneRencontre(Personnage a,
                              Personnage b) {
        System.out.print(a.getNom());
        System.out.print(" rencontre ");
        System.out.print(b.getNom() + " : " );
        a.rencontrer(b);
    }
}
```

le fait que « a » est un guerrier pour appeler la méthode « rencontrer » de « Guerrier » et non pas la méthode « rencontrer » de « Personnage », en dépit du fait qu'on décrit ici « a » comme un personnage. Si on appelait ici la méthode « rencontrer » de « Personnage », on aurait une résolution statique qui ne dépend que de ce qui est écrit statiquement alors que ce qui se passe en java c'est une résolution dynamique où on va appeler la méthode « rencontrer » de l'instance réelle qui est passée, on regarde quelle est l'instance : c'est un « Guerrier », donc on va appeler la méthode ici « rencontrer » de « Guerrier ».

notes

résumé



Voilà donc pour les rappels.

notes

résumé

3m 13s



Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- ▶ donner une définition générale de certaines méthodes, *compatibles avec toutes les sous-classes*,
- ▶ ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes

Voyons maintenant comment ces ingrédients, comment le polymorphisme va nous permettre à mieux spécifier, à un niveau plus abstrait, nos programmes en introduisant ces notions de classes abstraites et de méthodes abstraites.

notes

résumé

3m 14s



Exemple :

```
class FigureFermee {  
    // ...  
  
    // difficile à définir à ce niveau !..  
    public double surface(...) { ??? }  
  
    // ...pourtant la méthode suivante en aurait besoin !  
    public double volume(double hauteur) {  
        return hauteur * surface();  
    }  
}
```



Figure Ferme
↑
Cercle

Définir **surface** de façon arbitraire sachant qu'on va la redéfinir plus tard n'est pas une bonne solution (source d'erreurs) !

Solution : déclarer la méthode **surface** comme **abstraite**

Commençons par présenter le problème auquel répondent ces méthodes abstraites justement. Au niveau le plus élevé d'une hiérarchie on ne sait pas forcément définir une méthode que l'on sait pourtant, qui va exister pour toutes les sous-classes par exemple, si l'on imagine avoir une classe très générale qui représente des figures géométriques fermées, quelconques. À ce moment là, au niveau de cette classe il va être assez difficile de définir comment calculer la surface. Calculer la surface d'une figure géométrique quelconque, est certainement quelque chose de difficile à définir ; on ne saurait pas très bien le faire. Pourtant on sait que toutes figures fermées, -- par exemple, un cercle est une figure fermée -- donc on sait qu'au niveau de toute les figures fermées spécifiques, on devrait quand même pouvoir définir la méthode « surface ». Donc, on imagine que toutes les figures fermées vont avoir une méthode « surface » même si on ne sait pas bien la définir au niveau le plus abstrait. Et pourtant, pour pousser le problème encore un peu plus loin

notes

résumé

3m 27s



Plusieurs équipes collaborent à la création d'un jeu.

Une équipe prend en charge les classes de base suivantes :

- ▶ **Jeu** :
 - ▶ Classe pour gérer le jeu
 - ▶ Se contente ici d'afficher les personnages
- ▶ **Personnage** :
 - ▶ Classe de base pour les personnages

Une autre équipe ajoutera des sous-classes de personnages spécifiques.

on peut même imaginer que cette méthode « surface » soit utilisée au niveau le plus élevé, par exemple, pour calculer un volume engendré sur une certaine hauteur par cette surface, à ce moment là ce volume engendré, suivant cette hauteur, par cette figure géométrique, serait défini par le produit de la hauteur multiplié par la surface de la figure fermée. Donc on pourrait même faire appel à cette méthode même si on ne sait pas très bien comment la définir au plus haut niveau, elle serait définie pour tous les exemples concrets de figures fermées. Et donc la bonne façon de faire ceci, une méthode que l'on sait qui doit exister mais que l'on ne sait pas forcément bien définir au niveau d'une super-classe, cette façon de faire c'est d'introduire la méthode comme une méthode abstraite. Illustrons encore ce concept de méthodes abstraites sur un autre exemple et revenons donc à nos jeux

notes

résumé

4m 25s



La classe `Jeu` développée par la première équipe :
☛ gère un tableau de personnages et les affiche

```
class Jeu {  
    private ArrayList<Personnage> persos;  
    // ...  
    public void afficher() {  
        for (Personnage unPerso : persos) {  
            unPerso.afficher();  
        }  
    }  
    public void ajouterPersonnage(...) { // ...  
    }  
    // ...  
}
```

avec des personnages où nous avons donc des guerriers, qui sont des personnages, des magiciens, qui sont des personnages, des voleurs etc.

notes

résumé

5m 25s



Plusieurs équipes collaborent à la création d'un jeu.

Une équipe prend en charge les classes de base suivantes :

- ▶ **Jeu :**
 - ▶ Classe pour gérer le jeu
 - ▶ Se contente ici d'afficher les personnages
- ▶ **Personnage :**
 - ▶ Classe de base pour les personnages

Personnage
↑
Guerrier ...

Une autre équipe ajoutera des sous-classes de personnages spécifiques.

La classe « Jeu » contiendrait bien sûr plusieurs personnages, on a décidé ici de les mettre dans un tableau dynamique de personnages et imaginons que dans le jeu on souhaite afficher l'ensemble des personnages donc, on ferait bien sûr une boucle qui parcourt l'ensemble des personnages qui parcourt ce tableau « persos » ici, et pour chacun des personnages on afficherait ce personnage. Le problème c'est comment afficher un personnage générique ?

notes

résumé

5m 34s



La classe `Jeu` développée par la première équipe :
▀ gère un tableau de personnages et les affiche

```
class Jeu {  
    private ArrayList<Personnage> persos;  
    // ...  
    public void afficher() {  
        for (Personnage unPerso : persos) {  
            unPerso.afficher();  
        }  
    }  
    public void ajouterPersonnage(...) { // ...  
    }  
    // ...  
}
```

On ne sait pas afficher un personnage générique on sait certainement afficher un guerrier on sait certainement afficher un magicien, afficher un voleur donc chacune des sous-classes spécifiques sait comment s'afficher mais par contre, au niveau général d'un personnage on ne sait pas forcément l'afficher,

notes

résumé

6m 3s



Si l'on ne met aucune méthode `afficher` dans `Personnage`, la classe `Jeu` ne compile pas :

```
class Jeu {  
    // ...  
    public void afficher() {  
        for (Personnage unPerso : persos) {  
            unPerso.afficher(); // ERREUR !  
        }  
    }  
    // ...  
}
```

☛ On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...

De plus, on aimerait :

- ▶ imposer aux sous-classes (`Guerrier`, ...) d'avoir leur méthode `afficher` spécifique

par contre, on sait qu'on en a besoin ici, au niveau très général du jeu, de pouvoir afficher les personnages. Donc comment faire ? Bien sûr si on ne fait rien au niveau de la classe « `Personnage` », si on ne définit pas la méthode « `afficher` » dans « `Personnage` » alors le code « `afficher` » de « `Jeu` » ne compilera pas

notes

résumé

6m 21s



On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...
...mais comment faire ?

```
class Personnage {  
    private String nom;  
    private int energie;  
    // ...  
    // constructeurs  
    // ...  
    public void afficher() {  
        // Tous les personnages doivent pouvoir s'afficher !...  
        // ...mais comment???  
    }  
}
```

puisque ici on appelle bien la méthode « afficher » d'un personnage et le compilateur ne saurait pas quoi faire il nous donnerait donc une erreur. On doit donc, pour pouvoir écrire ceci au niveau global du jeu, on doit mettre une méthode « afficher » dans chaque classe « Personnage ». De plus, on aimerait imposer à chacune des sous-classes, des guerriers, des magiciens de s'afficher avec leurs propres méthodes « afficher », d'avoir une méthode spécifique et que ce soit cette méthode spécifique qui soit appelée dans l'appel général ici au niveau du jeu. Donc on veut effectivement forcer d'avoir une méthode spécifique au niveau des classes et que cette méthode soit polymorphique.

notes

résumé

6m 37s



On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...
...mais comment faire ?

```
class Personnage {  
    private String nom;  
    private int energie;  
    // ...  
    // constructeurs  
    // ...  
    public void afficher() {  
        // Tous les personnages doivent pouvoir s'afficher !...  
        // ...mais comment???  
    }  
}
```



Et comment *imposer* que la méthode `afficher` soit redéfinie dans les sous-classes ?

Mais comment faire si on ne sait pas, supposons qu'on ne sache pas afficher un personnage générique,

notes

résumé

7m 12s



Première « solution » :

ajouter une méthode quelconque définie arbitrairement :

```
class Personnage {  
    // ...  
    // On n'affiche rien : corps de la méthode vide  
    public void afficher() { }  
    // ...  
}
```

et de plus, comment imposer que cette méthode « afficher », soit redéfinie ? Forcer la redéfinition dans les sous-classes ? Une première solution pour le premier problème : comment définir l'affichage pour un personnage générique serait d'avoir une méthode quelconque définie arbitrairement,

notes

résumé

7m 18s



Première « solution » :

ajouter une méthode quelconque définie arbitrairement :

```
class Personnage {  
    // ...  
    // On n'affiche rien : corps de la méthode vide  
    public void afficher() { }  
    // ...  
}
```

C'est une **très mauvaise idée**

- ❌ Mauvais modèle de la réalité
(affichage incorrect si une sous-classe ne redéfinit pas la méthode : personnages fantômes !)
- ❌ Cette solution n'impose pas que la méthode **afficher** soit redéfinie

par exemple, supposons ici qu'arbitrairement on définisse que l'affichage générique d'un personnage c'est de ne rien afficher. Cette solution c'est vraiment une très très mauvaise idée

notes

résumé

7m 37s



Bonne solution :

Signaler que la méthode doit exister dans *chaque* sous-classe sans qu'il soit nécessaire de la définir dans la super-classe

- ☛ Déclarer la méthode comme **abstraite**

parce que d'abord c'est un mauvais modèle de la réalité, les personnages ne s'affichent pas comme du vide donc ça ne correspond à rien, l'affichage est incorrect, et en plus l'affichage serait incorrect si une sous-classe oubliait de redéfinir la méthode, donc on aurait des personnages qui ne s'affichent pas, des personnages fantômes ce qui est un peu embêtant pour le jeu. De plus, cette solution ne répond pas au deuxième problème, elle n'oblige pas les sous-classes à redéfinir leurs propres méthodes « afficher ». La seule bonne solution donc

notes

résumé

7m 47s





c'est de signaler que la méthode « afficher » doit exister et doit être définie dans chacune des sous-classes c'est ce qu'on appelle une méthode abstraite. Voilà donc un deuxième exemple, j'espère qu'il vous a bien fait comprendre l'intérêt d'avoir des méthodes abstraites comme ça, définies au niveau de super-classes de classes abstraites. de super-classes de classes abstraites.

notes

résumé

8m 19s

