


# Concept. et prog. objet


## \1 TD n°3 : abstraction

### Détails

Écrit par stéphane Domas

Catégorie : M3105 - Concept. et prog, objet avancée (/index.php/menu-cours-s3/menu-mmi3-test)

 Publication : 21 octobre 2014

 Affichages : 25

### 1°/ Principe

- L'abstraction est un concept qui peut avoir plusieurs significations et noms selon le langage et le contexte dans laquelle elle est utilisée.
- Par exemple, en Java, on peut déclarer des classes et des méthodes abstraites, alors qu'en C++, il n'y a que les méthodes, que l'on qualifie de virtuelles pures plutôt que abstraites. Pour la suite, on se basera sur Java.
- En Java, on utilise le mot-clé `abstract` que ce soit pour une classe ou une méthode. Même s'il y a une relation entre les deux cas, l'impact de son utilisation n'est pas le même.

#### 1.1°/ Classe abstraite

- Déclarer une méthode comme abstraite empêche son instanciation. Elle ne fait que représenter un concept.
- Pour autant, elle n'est pas vide. On la définit avec des attributs et méthodes qui seront hérités dans des sous-classes. Par exemple, un véhicule est un concept mais il a, entre autres, un attribut nombre de roues, une méthode `avancer()`, ...
- La raison pour déclarer une méthode abstraite est souvent de l'ordre de la modélisation ET applicative. Par exemple, la classe `Humain` représente un concept qui regroupe les hommes et les femmes. Dans la vie réelle, on ne trouve pas d'instance d'humain mais juste des hommes et des femmes. Il est donc logique que la classe `Humain` soit abstraite dans un programme qui imite la vie réelle. Pour l'implémenter ainsi, on ajoute le mot-clé `abstract` devant `class` :

```
1 | abstract class Humain {  
2 |     ...  
3 | }  
4 |  
5 | Humain h = new Humain(...); // ERREUR COMPILATION
```

- Autre exemple, les classes `Polygone` et `PolygoneRegulier`. Les deux représentent des concepts géométriques donc abstraits. Mais dans une application de dessin vectoriel, on dessine réellement des polygones, donc on a besoin de pouvoir instancier les deux classes. Cela dit, si l'application est basique et ne permet de dessiner que des polygones réguliers, on pourrait également avoir `Polygone` comme abstraite.
- Dans les fait, on déclare une classe abstraite quand :
  - cette classe ne sera jamais instanciée lors de l'exécution de l'application,
  - on veut rendre impossible son instanciation par d'autre programmeurs (notamment pour éviter es transtypages invalides)
  - elle contient au moins une méthode abstraite (cf. section 2)

#### 1.2°/ Méthode abstraite

- Comme pour les classes, une méthode abstraite représente juste un concept.
- Contrairement à une classe abstraite qui peut contenir des membres, une méthode abstraite n'a pas de code.
- Bien souvent, on déclare une méthode abstraite car il est impossible (ou incorrect en terme POO) de créer du code pour cette méthode, alors que c'est possible dans les sous-classes.
- Prenons par exemple la méthode `rencontre()` entre deux humains. D'un point de vue implémentation, il existe plusieurs solutions, plus ou moins bonnes et respectant la POO.

- Solution la pire : définir `rencontre()` dans `Humain` et pas dans `Homme` et `Femme`. Cela impose de tester si l'humain courant et celui passé en paramètre sont de sexe opposés, de les transtyper correctement lorsque l'on veut accéder à des attributs propres à `Homme` et `Femme`, ... Bref, cela rend le code long, peu lisible, fastidieux à écrire et surtout qui n'exploite absolument pas les principes POO.
- Solution intermédiaire : définir `rencontre()` dans `Humain` avec un code vide et la redéfinir dans `Homme` et `Femme`. On a donc une méthode vide, ce qui n'est pas très productif.
- Solution top : déclarer `rencontre()` comme abstraite dans `Humain` et la définir (pas re-définir puisqu'elle n'a pas encore de code) dans `Homme` et `Femme`.

```

1 | abstract class Humain {
2 |     ...
3 |     public abstract Humain rencontre(Humain h);
4 | }
5 |
6 | class Homme extends Humain {
7 |     ...
8 |     public Humain rencontre(Humain h) {
9 |         ...
10 |    }
11 | }

```

- On remarque que la méthode abstraite n'utilise pas les `{}` puisqu'elle n'a pas de code.

## 2°/ Contraintes liées à l'abstraction

- Une classe contenant au moins une méthode abstraite **DOIT** être déclarée comme abstraite. Par exemple, la méthode `rencontre()` de `Humain` est abstraite, donc la classe `Humain` est abstraite.

**ATTENTION !** Ce n'est pas courant mais une classe abstraite peut ne contenir aucune méthode abstraite.

- Si une classe `A` déclare des méthodes abstraites, elles doivent obligatoirement être définies (normalement, par héritage ou bien redéfinie) dans **tous les descendant de `A` non abstraits**. Si ce n'est pas le cas, le compilateur signalera une erreur.
- En effet, quand une classe hérite d'une méthode abstraite sans la définir, c'est comme si elle-même déclarait une méthode abstraite. La 1ère contrainte s'applique et on doit donc rendre la classe abstraite.

```

1 | abstract class A {
2 |     public abstract void toto();
3 | }
4 | abstract class B extends A {
5 |     // ne définit pas toto donc hérite
6 |     // d'une méthode abstraite, donc classe abstraite
7 | }
8 | class C extends B {
9 |     public void toto { ... } // définition de toto()
10 | }
11 |
12 | class D extends A {
13 |     public void toto { ... } // définition de toto()
14 | }

```

- Dans l'exemple ci-dessus, il n'y a aucune erreur car `toto()` est définie dans `C` et `D` qui ne sont pas abstraites.
- Si on ajoute le code suivant :

```

1 | class E extends A {
2 |     // pas de redéfinition de toto()
3 | }

```

Le compilateur signale une erreur :

```

1 | E.java:1: error: E is not abstract and does not override abstract method toto() in A
2 | public class E extends A {
3 |     ^
4 | 1 error

```

- En effet, E n'est pas déclarée abstraite et ne définit pas `toto()`. Il existe deux solutions pour régler le problème :
  - définir `toto()` dans E,
  - rendre E abstraite (comme B).

### 3°/ Bizarreries

- Dans une arborescence, l'abstraction peut apparaître et disparaître plusieurs fois.
- 1er cas : une classe est abstraite parce qu'elle déclare une nouvelle méthode qui est abstraite, alors que sa super classe ne l'est pas. Par exemple :

```

1 | abstract class A {
2 |     public abstract void toto();
3 | }
4 | class B extends A {
5 |     public void toto();
6 | }
7 | abstract class C extends B {
8 |     public abstract void salut();
9 | }

```

- 2ème cas, sur une même méthode. Par exemple :

```

1 | abstract class A {
2 |     public abstract void toto();
3 | }
4 | class B extends A {
5 |     public void toto();
6 | }
7 | abstract class C extends B {
8 |     public abstract void toto();
9 | }

```

**ATTENTION !** ces cas sont généralement le signe d'une mauvaise modélisation (mais pas forcément).