**Université Paris Cité** **Programmation Orientée Objet**
**L2** **Année 2023-2024**

# TD - Séance n° 6 - Correction

## Interfaces et classes abstraites

## 1 Questions de cours

**Exercice 1** (*Interfaces vs classes abstraites*)

1. Peut-on instancier directement une interface ? une classe abstraite ?

   **Correction :** Non pour les deux, pas directement : il faut créer une classe concrète qui implémente l'interface, et une extension concrète de la classe abstraite.

2. Peut-on munir une interface d'un constructeur sans corps ? d'un constructeur avec un corps ? Mêmes questions pour une classe abstraite.

   **Correction :** Une interface ne peut être munie d'un constructeur, et encore moins d'un constructeur avec un corps.

   Les constructeurs d'une classe abstraite doivent tous être implémentés - seules ses méthodes peuvent être abstraites. Si la classe ne spécifie aucun constructeur, elle sera tout de même munie d'un constructeur par défaut – sans arguments, et de corps vide.

   Un constructeur d'une classe abstraite est invocable explicitement par `this(...)` dans un autre constructeur de cette classe, ou par `super(...)` dans un constructeur d'un héritier direct de la classe. Dans tous les cas, il ne peut s'agir que de la première instruction du constructeur, et cette invocation est non renouvelable.

3. Considérons l'instruction suivante : `A a = new B();`

   Quelles sont les deux conditions que doit vérifier la classe `B` pour que cette instruction soit valide :
   — lorsque `A` est une classe abstraite ?
   — lorsque `A` est une interface ?

   **Correction :** La classe `B` doit dans chaque cas être concrète. Dans le premier, `A` doit être ancêtre de `B`. Dans le second, `B` ou un ancêtre de `B` doit implémenter `A`.

4. Une interface peut-elle contenir des champs ? Avec quels modificateurs ? Doivent-ils être initialisés ? Mêmes questions pour une classe abstraite.

   **Correction :** Une interface peut avoir des champs, mais ils seront implicitement `public`, `static` et `final` : il s'agit donc toujours de constantes, qui doivent être initialisées explicitement dans le corps de l'interface. Il est inutile de mentionner ces modificateurs, puisque ce sont les seuls possibles.

   Une classe abstraite peut avoir tout type de champ, avec tout modificateur.

5. Une interface peut-elle contenir des méthodes statiques/non-statiques, abstraites/non-abstraites ? Même question pour une classe abstraite.

   **Correction :** Par défaut, les méthodes d'une interface sont non statiques, abstraites et publiques – les modificateurs `public` et `abstract` peuvent être ajoutés aux noms de ces méthodes, mais c'est inutile. Le modificateur `public` est en revanche nécessaire lorsqu'une classe implémentant cette interface fournit l'implémentation de l'une de ses méthodes. Depuis Java 8, d'autres possibilités sont offertes au programmeur :

   (a) On peut ajouter à une interface des méthodes `static`, qui doivent être implémentées.

   (b) On peut proposer pour une méthode non statique d'une interface une *implémentation par défaut*, en la faisant précéder du mot-clef `default`. L'implémentation peut mentionner les champs statiques de l'interface, ainsi que le mot-clef `this`, sur lequel sont invocables toutes les méthodes de l'interface.

   Une classe pouvant implémenter plusieurs interfaces, il est possible que deux de ces interfaces proposent deux implémentations par défaut (a priori distinctes) d'une même méthode. La classe doit alors lever cette ambiguïté en proposant une nouvelle implémentation de la méthode. Les anciennes implémentations restent accessibles via la syntaxe `I.super.m()` et `J.super.m()` si l'on est dans une classe qui implémente les deux interfaces `I` et `J` implémentant chacune par défaut la méthode `m`.

   Dans une classe abstraite on peut avoir la même chose, sauf des méthodes en `default` : méthodes abstraites, méthodes concrètes, méthodes statiques et concrètes. Noter tout de même une différence : les méthodes statiques d'interface ne sont pas héritées par les classes qui implémentent l'interface (dans ces classes, il faut les invoquer via `I.m()` où `I` est le nom de l'interface). En revanche les méthodes statiques d'une classe abstraite sont héritées par les classes qui étendent la classe abstraite (on peut les invoquer via `m()`).

6. Une interface peut-elle hériter d'une autre interface ? d'une classe abstraite ?

   **Correction :** Oui. Non.

7. Une classe abstraite peut-elle hériter d'une autre classe abstraite ? d'une interface ?

   **Correction :** Oui. Non, mais ce n'est qu'une question de terminologie : on dit qu'elle implémente l'interface, pas qu'elle en hérite.

**Exercice 2** (*Interfaces et héritage*) On considère les trois déclarations d'interfaces suivantes, que l'on supposera placées dans des fichiers distincts :

```
public interface I {
    default void m() {
        System.out.println("I, m");
    }
    void n();
}

public interface J {
```

```
    void p();
}

public interface K {
    void q();
}
```

Soient à présent les trois déclarations de classes suivantes, que l'on supposera encore dans des fichiers distincts :

```
public abstract class A implements I, J {
    public void n() {
        System.out.println("A, n");
    }
}
public class B extends A {
    public void p() {
        System.out.println("B, p");
    }
}
public class C extends B implements K {
    public void m() {
        System.out.println("C, m");
    }
    public void q() {
        System.out.println("C, q");
    }
}
```

Dans le code ci-dessous, quelles sont les instructions qui déclencheront un message d'erreur, et pourquoi ? En supposant ces instructions commentées, qu'affichera ce code ?

```
A a = new A(); a.m(); a.n(); a.p();
B b = new B(); b.m(); b.n(); b.p();
C c = new C(); c.m(); c.n(); c.p(); c.q();

A u = b; u.m(); u.n(); u.p();
A v = c; v.m(); v.n(); v.p(); v.q();
```

**Correction :**

```
// A est abstraite, donc non instantiable.
  // A a = new A(); a.m(); a.n(); a.p();

  // b.m() : I, m  (implementation par défaut)
  // b.n() : A, n  (implementation héritée de A)
  // b.p() : B, p  (implementation de B)
  B b = new B(); b.m(); b.n(); b.p();

  // c.m() : C, m  (nouvelle implementation dans C)
```

```
// c.n() : A, n  (héritée de A via B)
// c.p() : B, p  (héritée de B)
// c.q() : C, q  (implementation de C)
C c = new C(); c.m(); c.n(); c.p(); c.q();

// meme effet qu'avec b, par liaison dynamique
A u = b; u.m(); u.n(); u.p();

// meme effet qu'avec c, par liaison dynamique, sauf pour
// e.q() : le nom de méthode q n'est pas visible dans la
// classe A
A v = c; v.m(); v.n(); v.p(); // e.q();
```

## 2 Modélisation

**Exercice 3** (*Instruments de musique*) Dans cet exercice, nous allons construire une hiérarchie de classes et d'interfaces permettant de modéliser des instruments de musique. Plusieurs solutions existent pour classifier ces instruments.

Une première solution consiste à les différencier selon le procédé qui leur permet de produire un son. Certains instruments sont dits mécaniques, au sens où le son provient d'une vibration mécanique d'une pièce ou d'une masse d'air (tous les instruments traditionnels). Les instruments mécaniques se séparent en trois grandes familles :

— Les cordes, pincées (guitare), frappées (piano), frottées (violon).
— Les vents, divisés en bois (saxophone) et cuivres (trombone).
— Les percussions, à peau (timbale), en bois (xylophone), en métal (triangle).

D'autres instruments, dits électroniques, produisent un son à l'aide d'un générateur (tous les synthétiseurs).

Une seconde solution consiste à différencier les instruments selon la manière dont leur son est amplifié. L'amplification peut à nouveau être mécanique (par une caisse de résonance) ou bien électrique (à l'aide d'un microphone). Une guitare électrique, par exemple, est un instrument mécanique à amplification électrique : le son provient de la vibration d'une corde, puis il est amplifié á l'aide de microphones qui transforment cette vibration en signal électrique.

1. Trouver et représenter graphiquement une hiérarchie de classes et d'interfaces permettant de représenter les différentes sortes d'instruments, ainsi que la manière dont ils sont amplifiés. Insérer dans cette hiérarchie des classes représentant un piano, un saxophone, et une guitare électrique.

   Inutile de spécifier l'implémentation des classes : indiquez seulement visuellement quelles sont leurs liens de parenté, quelles sont les interfaces implémentées, et quelles sont les classes qui doivent être abstraites.

2. Donner la déclaration (l'en-tête) des classes représentant un piano, un saxophone, et une guitare électrique.

3. On aimerait bien aussi pouvoir dire qu'un piano, un orgue, un piano synthétiseur, appartiennent tous à la famille des claviers. Comment faire ?

**Correction :**

```
// hierachie des instruments
abstract class Instrument {}
abstract class InstrumentMecanique extends Instrument {}
abstract class InstrumentElectronique extends Instrument {}

abstract class Cordes extends InstrumentMecanique {}

abstract class CordesFrappees extends Cordes {}
abstract class CordesPincees extends Cordes {}
abstract class CordesFrottees extends Cordes {}

abstract class Vent extends InstrumentMecanique {}
abstract class Cuivre extends Vent {}
abstract class Bois extends Vent {}

abstract class Percussion extends InstrumentMecanique {}
abstract class PercussionPeau extends Percussion {}
abstract class PercussionBois extends Percussion {}
abstract class PercussionMetal extends Percussion {}

// hierarchie d'interfaces specifiant le type d'amplification
interface Amplification {}
interface AmplificationElectrique extends Amplification {}
interface AmplificationMecanique extends Amplification {}


// interface commune aux instruments a clavier
interface Clavier {}

// quelques exemples de classes concretes
class Orgue extends Vent
    implements AmplificationMecanique, Clavier {}
class Piano extends CordesFrappees
    implements AmplificationMecanique, Clavier {}
class GuitareElectrique extends CordesPincees
    implements AmplificationElectrique {}
class Saxophone extends Bois implements
   AmplificationMecanique {}
class PianoSynthetiseur extends InstrumentElectronique
    implements AmplificationElectrique, Clavier {}
```

## Si vous avez le temps

**Exercice 4** Implémenter une ou plusieurs interfaces peut être vu comme la capacité d'un objet à remplir certains rôles, indépendamment de sa position dans la hiérarchie des classes. Cet exercice peut être résolu sans utiliser de `if` ni de `instanceof`.

(1) *Une école de samouraïs dispense deux types de cours : des cours de combat et des cours de méditation. Chaque cours est assuré par un unique orateur, qui doit avoir été formé dans la spécialité correspondante (c'est-à-dire, formé à la méditation pour un cours de méditation, formé au combat pour un cours de combat, un orateur pouvant être formé aux deux).*

Construire et implémenter deux hiérarchies modélisant la situation décrite ci-dessus : une hiérarchie de classes pour les cours et une hiérarchie d'interfaces (vides) pour les orateurs. Les constructeurs des cours devront, par simple application des règles de typage, interdire la création d'un cours par un orateur non formé à sa spécialité.

(2) *L'école comporte deux types de membres : les élèves et les maîtres. Les élèves sont divisés en deux groupes : apprentis et disciples. Les maîtres se divisent en deux groupes : initiés et grands maîtres.*

*L'orateur d'un cours de méditation ne peut être qu'un initié ou un grand maître. L'orateur d'un cours de combat peut être un disciple, un initié ou un grand maître.*

Construire et implémenter une hiérarchie de classes supplémentaires (vides) pour les membres de l'école, en respectant cette dernière contrainte.

**Correction :**

```
interface Orateur {}

interface OrateurCombat extends Orateur {}

interface OrateurMeditation extends Orateur {}

abstract class Cours {
    Orateur orateur;
    Cours (Orateur orateur) {
        this.orateur = orateur;
    }
}

class CoursCombat extends Cours {
    CoursCombat (OrateurCombat orateurCombat) {
        super (orateurCombat);
    }
}

class CoursMeditation extends Cours {
    CoursMeditation (OrateurMeditation orateurMeditation) {
```

```
        super (orateurMeditation);
    }
}

abstract class Membre {}

abstract class Eleve extends Membre {}

class Apprenti extends Eleve {}

class Disciple extends Eleve implements OrateurCombat {}

abstract class Maitre extends Membre {}

class Initie extends Maitre
implements OrateurMeditation, OrateurCombat {}

class GrandMaitre extends Maitre
implements OrateurMeditation, OrateurCombat {}

public class Ecole {}
```