

Programmation Orientée Objet -

Chapitre 4 – Classes/Méthodes Abstraites, Interfaces


Jean-Marie Normand
Bâtiment E - Bureau 211
`jean-marie.normand@ec-nantes.fr`

Plan du cours I

- 8 Classes et méthodes abstraites, Interfaces
 - Méthodes et classes abstraites
 - Interfaces



Plan

- 
- 8 Classes et méthodes abstraites, Interfaces
 - Méthodes et classes abstraites
 - Interfaces

Classes abstraites : exemple introductif

Exemple archi-classique des formes géométriques :

- on veut définir une application permettant de manipuler des formes géométriques (triangles, rectangles, cercles...)
- chaque forme est définie par sa position dans le plan
- chaque forme peut être déplacée (translation), on peut calculer son périmètre, sa surface...

Classe *Forme*

La classe *Forme*

```
public class Forme {  
  
    /**  
     * abscisse du "centre"  
     */  
    protected double x;  
  
    /**  
     * ordonnee du "centre"  
     */  
    protected double y;  
  
    public void deplacer(double dx, double dy) {  
        x += dx;  
        y += dy;  
    }  
}
```

Classe *Cercle*

La classe Cercle

```
public class Cercle extends Forme {  
    /// rayon du cercle  
    protected double r;  
    /**  
     * surface du cercle  
     * @return la surface du cercle (dans la meme unite que le rayon)  
     */  
    public double surface() {  
        return Math.PI * r * r;  
    }  
    /**  
     * perimetre du cercle  
     * @return le perimetre du cercle (dans la meme unite que le rayon)  
     */  
    public double perimetre() {  
        return 2d * Math.PI * r;  
    }  
}
```

Diagramme de classes

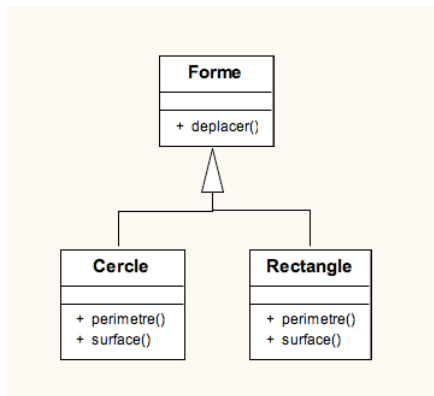


Figure: Diagramme de classes de nos *Formes*.

1. Gérer des listes de formes

- on veut pouvoir gérer des listes de formes ; on va exploiter le **polymorphisme** et la **généricité**
- rappel : le polymorphisme est le fait qu'un même code puisse s'adapter au type des données auquel il s'applique

Une liste de **Forme**

```
public class ListeFormes {  
    /**  
     * la liste (utilise LinkedList<E>)  
     */  
    protected LinkedList<Forme> listeFormes;  
  
    /**  
     * constructeur  
     */  
    public ListeFormes() {  
        listeFormes = new LinkedList<Forme>();  
    }  
}
```


1. Gérer des listes de formes II

Une liste de **Forme**

```
/**
 * ajoute une forme a la liste de forme
 * @param f reference sur la forme a ajouter
 */
public void ajouter(Forme f) {
    listeFormes.add(f);
}

/**
 * traduire l'ensemble des formes de la liste
 * @param dx abscisse de la translation
 * @param dy ordonnee de la translation
 */
public void toutDeplacer(double dx, double dy) {
    for (Forme f : listeFormes) {
        f.deplacer(dx, dy);
    }
}
```

2. Et calculer la surface totale ?

- Toutes nos classes ont une méthode `surface()`

Calcul de la surface totale de la liste de formes 

```
/* calcul de la surface totale : iteration sur les surfaces
 * @return la surface totale
 */
public double surfaceTotale() {
    double s=0d;
    for (Forme f : listeFormes) {
        s += f.surface();
    }
    return s;
}
```

2. Et calculer la surface totale ?

- Toutes nos classes ont une méthode `surface()`

Calcul de la surface totale de la liste de formes 

```
/* calcul de la surface totale : iteration sur les surfaces
 * @return la surface totale
 */
public double surfaceTotale() {
    double s=0d;
    for (Forme f : listeFormes) {
        s += f.surface();    Méthode « surface » n'est pas dans classe Forme
    }
    return s;
}
```

- mais cela ne compile pas !

2. Et calculer la surface totale ?

- Toutes nos classes ont une méthode `surface()`

Calcul de la surface totale de la liste de formes 

```
/* calcul de la surface totale : iteration sur les surfaces
 * @return la surface totale
 */
public double surfaceTotale() {
    double s=0d;
    for (Forme f : listeFormes) {
        s += f.surface();
    }
    return s;
}
```

- mais cela ne compile pas !
- mauvaise solution (peu élégante) :

Dans la classe `Forme`

```
double surface() {
    return 0d;
}
```

2. Et calculer la surface totale ?

Pourquoi cette solution est-elle mauvaise ?

Rajouter une méthode `surface()` dans la classe `Forme` est une très mauvaise idée :

- cette solution n'impose pas que la méthode `surface()` soit redéfinie dans les sous-classes de `Forme` !
- c'est un mauvais modèle de la réalité : le résultat du calcul des surfaces totales d'une liste de Formes est **incorrect** si une classe ne redéfinit pas cette méthode !

Bonne solution :

Signaler que la méthode **DOIT** exister dans toutes les sous-classes sans qu'il soit nécessaire de lui donner une définition incorrecte dans la super-classe \Rightarrow **déclarer la méthode comme abstraite** !

Les méthodes et classes abstraites

Caractéristiques :

- elles servent à définir des **concepts incomplets qui devront être définis dans les sous-classes**
- permettent la factorisation du code
- écriture/définition en Java
 - ▶ méthode abstraite : la déclaration est précédée du mot-clé **abstract**, mais avec un “;” à la place du code (i.e. sans code)
 - ▶ une classe abstraite : toute classe qui contient au moins une méthode abstraite ! Déclaration précédée du mot clé **abstract**
- permettent d'**imposer** aux sous-classes (non-abstraites) de redéfinir les méthodes abstraites héritées

Classe `Forme` abstraite

La classe `Forme` abstraite

```
public abstract class Forme {  
    // ...  
  
    /**  
     * methode asbtraite : chaque Forme devra etre capable de determiner  
     * sa surface  
     */  
    public abstract double surface();  
  
    /**  
     * methode abstraite : chaque Forme devra etre capable de determiner  
     * son perimetre  
     */  
    public abstract double perimetre();  
}
```

- les autres classes ne sont pas modifiées
- la classe `ListeFormes` compile !

Test de la classe ListeForme

Une nouvelle liste de **Forme**

```
/**
 * un petit main de test qui se contente de creer 3 formes
 * de les ajouter dans une liste et de calculer la surface
 * totale de ces trois formes
 * @param args
 */
public static void main(String[] args) {
    ListeFormes lf = new ListeFormes();
    Cercle c1 = new Cercle(4);
    Rectangle r1,r2;
    r1 = new Rectangle(2d,3d);
    r2 = new Rectangle(4d,4d);
    lf.ajouter(r1);
    lf.ajouter(r2);
    lf.ajouter(c1);

    System.out.println("surface totale : "+lf.surfaceTotale());
}
```


Classes abstraites

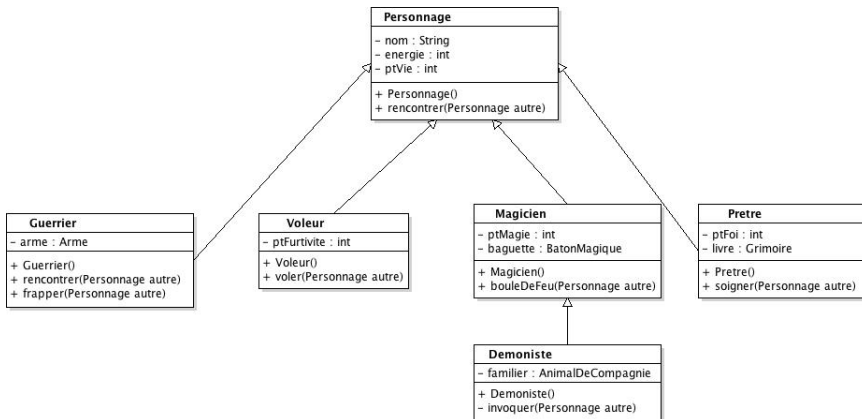
- **classe abstraite** : classe **non instanciable**, c'est-à-dire qu'il n'est pas possible de créer une instance de la classe déclarée abstraite
- **méthode abstraite** : méthode n'admettant pas d'implémentation
 - ▶ au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser
- une classe qui contient **au moins une méthode déclarée abstraite** devient **abstraite** (l'inverse n'est pas vrai) :
 - ▶ ses sous-classes **restent abstraites tant qu'elles ne fournissent pas les définitions de toutes les méthodes abstraites dont elles héritent**
- les **méthodes abstraites** sont particulièrement utiles pour mettre en œuvre le **polymorphisme**

Classes abstraites

- une classe abstraite est une **description d'objets** destinée à être **héritée** par des classes plus spécialisées
- pour être utile, elle doit admettre des sous-classes **concrètes**
 - ▶ une **classe concrète doit implémenter toutes les opérations (soit directement soit par la hiérarchie d'héritage) de la classe abstraite mère**
- une classe abstraite **permet de regrouper certaines caractéristiques communes** à ses sous-classes et définit un **comportement minimal commun**

Et pour notre *World of ECN* ?

Rappel de la hiérarchie actuelle :



Remarques - Ajout de fonctionnalités

Bilan

Si on réfléchit bien :

- Un **Personnage** représente t'il vraiment une classe que l'on souhaite pouvoir instancier ?

Remarques - Ajout de fonctionnalités

Bilan

Si on réfléchit bien :

- Un **Personnage** représente t'il vraiment une classe que l'on souhaite pouvoir instancier ?
 - ▶ pas vraiment : un de nos **Personnages** du jeu aura forcément une classe autre que **Personnage**
 - ▶ on devrait donc la **déclarer comme abstraite pour interdire de pouvoir l'instancier** directement
- Si je souhaite ajouter une méthode **afficher()** (d'affichage 3D de mon personnage) doit on avoir une implémentation dans la classe **Personnage** ?

Remarques - Ajout de fonctionnalités

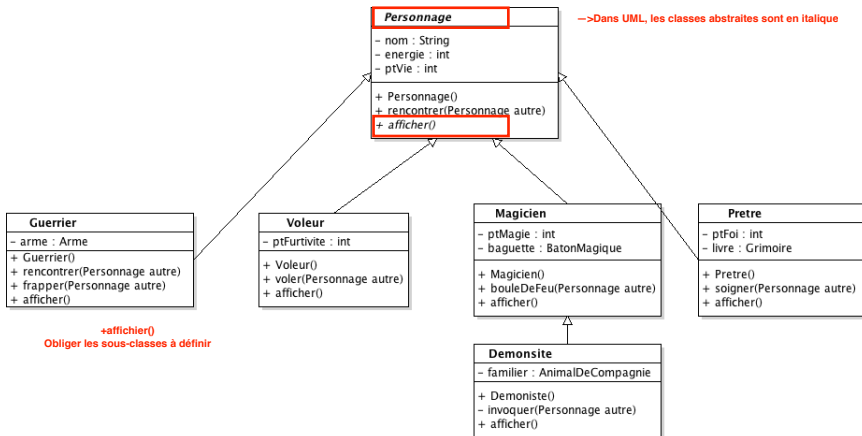
Bilan

Si on réfléchit bien :

- Un **Personnage** représente t'il vraiment une classe que l'on souhaite pouvoir instancier ?
 - ▶ pas vraiment : un de nos **Personnages** du jeu aura forcément une classe autre que **Personnage**
 - ▶ on devrait donc la **déclarer comme abstraite pour interdire de pouvoir l'instancier** directement
- Si je souhaite ajouter une méthode **afficher()** (d'affichage 3D de mon personnage) doit on avoir une implémentation dans la classe **Personnage** ?
 - ▶ pas vraiment non plus : l'affichage des caractéristiques et potentiellement du modèle 3D devrait être à la charge des classes que l'on peut effectivement instancier
 - ▶ on doit rajouter une méthode abstraite **afficher()** dans **Personnage** afin d'obliger les sous-classes à la définir

Et pour notre *World of ECN* ?

Nouvelle hiérarchie :



Et pour notre *World of ECN* ?

Nouvelle hiérarchie

Précisons qu'en UML les méthodes et les classes abstraites sont représentées en *italique*

Classes abstraites : problème potentiel

- la factorisation optimale des comportements communs à plusieurs classes nécessite le plus souvent plusieurs classes abstraites
- solutions possibles
 - ▶ une hiérarchie qui ne concrétise qu'une partie des méthodes avant d'arriver aux classes concrètes
 - ▶ lorsque plusieurs blocs de comportements sans rapport entre eux ont été identifiés, on utilise les **interfaces**

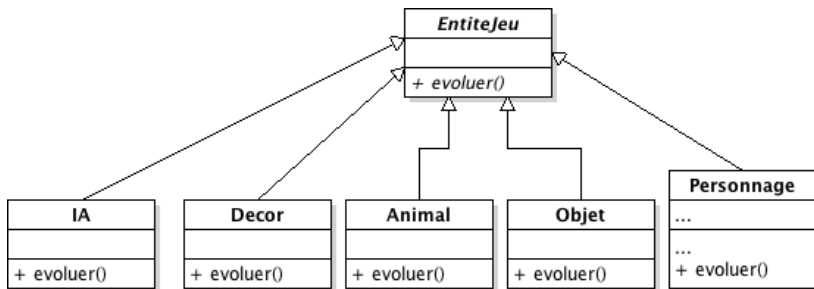
Exemple d'introduction aux interfaces

Supposons que l'on veuille rajouter des éléments à notre *World of ECN* :

- des éléments de décor (murs, haies, etc.)
- des animaux
- des objets que l'on peut posséder
- une intelligence artificielle basique contrôlant des personnages non joueurs (PnJ), etc.

Exemple d'introduction aux interfaces

Chaque entité en plus des **Personnages** sera principalement dotée d'une méthode **evoluer()** qui permet de gérer l'évolution de l'entité au cours du jeu. On aurait ainsi un diagramme de ce style :



Première ébauche de conception

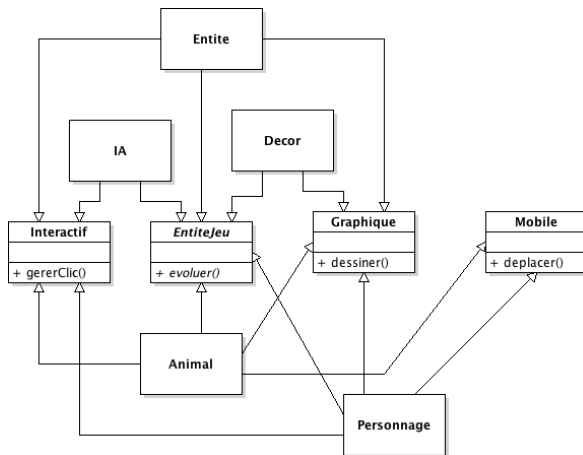
Si l'on analyse de plus près les besoins du jeu, on réalise que :

- certaines entités doivent avoir une **représentation graphique** (en supposant notre jeu 2D ou 3D), p. ex. les animaux, les décors, les personnages
- ... et d'autres non (IA)
- certaines entités doivent **être interactives** (contrôlables par le joueur ou réagir à un clic du joueur), p. ex. les personnages, les animaux
- ... et d'autres non (décors)
- certaines entités doivent **pouvoir se déplacer** (les personnages, les animaux)
- ... et d'autres non (objets, décors, IA)
- etc.

Comment organiser tout cela ?

Solution impossible (en Java)

Idéalement, on voudrait pouvoir mettre en place une hiérarchie de classes comme celle la :



Héritage simple/multiple

Mais ...

Java ne permet que l'héritage simple : chaque sous-classe ne peut avoir qu'une seule super-classe directe !

NB : ce n'est pas vrai dans tous les langages de programmation (p. ex. C++ admet l'héritage multiple)

Pourquoi pas d'héritage multiple en Java ?

- parce qu'il est parfois difficile à comprendre (quel sens lui donner ?)
- y compris pour le compilateur (si une sous-classe hérite de la même super-classe par plusieurs chemins différents)
- si une variable/méthode est déclarée dans plusieurs super-classes :
 - ambiguïté : laquelle utiliser ? comment y accéder ?

Analyse

En fait, souhaitait-on **vraiment** utiliser de l'héritage multiple ?

Nous voulions en fait :

Imposer à certaines classes de mettre en œuvre des méthodes communes !

Par exemple :

- `Personnage`, `Objet`, `Animal`, `IA` doivent avoir une méthode `gererClic()`
- mais `gererClic()` ne peut être une méthode de leur super-classe (car ça n'a aucun sens pour un décor)

Analyse

En fait, souhaitait-on **vraiment** utiliser de l'héritage multiple ?

Nous voulions en fait :

Imposer à certaines classes de mettre en œuvre des méthodes communes !

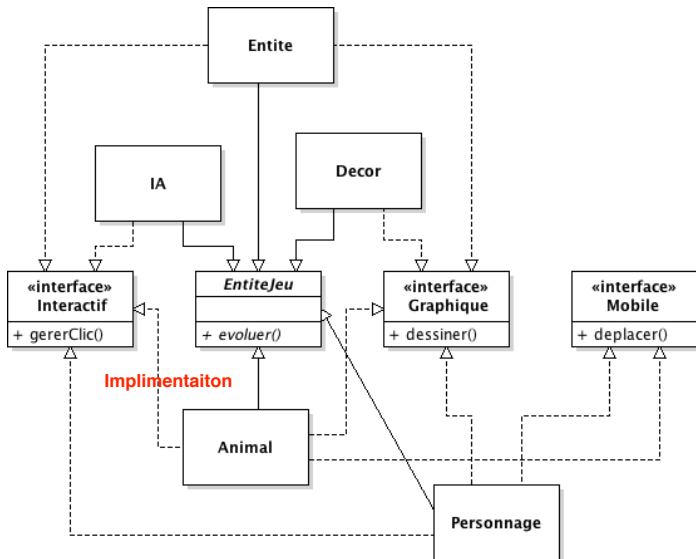
Par exemple :

- `Personnage`, `Objet`, `Animal`, `IA` doivent avoir une méthode `gererClic()`
- mais `gererClic()` ne peut être une méthode de leur super-classe (car ça n'a aucun sens pour un décor)

Solution :

Imposer un contenu commun à des sous-classes en dehors d'une relation d'héritage est possible grâce à la notion d'**interface** en Java !

Alternative possible



Alternative possible II

Solution avec interfaces

- **Interface** \neq **Classe**
- Une interface permet d'imposer à certaines classes d'avoir un contenu particulier (un ensemble de méthodes) sans que ce contenu ne fasse partie d'une classe
- Une interface peut être vue comme une classe abstraite :
 - ▶ **sans attributs**
 - ▶ dont les variables sont **nécessairement** `public`, `final` et `static` (i.e. des **constantes**)
 - ▶ dont **toutes** les méthodes sont abstraites (`public abstract`)
 - ▶ pour laquelle il est **interdit de déclarer un constructeur** !!
- Une interface fonctionne comme un **contrat** passé avec une classe : cette dernière s'**engage** à définir les méthodes déclarées dans l'interface !
- Une classe peut **implémenter** **plusieurs** interfaces

Interfaces I

Syntaxe :

```
public interface UneInterface {  
    constantes ou methodes abstraites  
}
```

Interface Graphique

```
public interface  
    Graphique {  
    public void dessiner();  
}
```

pas attribuer
pas constructor

Interface Interactif

```
public interface  
    Interactif {  
    public void gererClic();  
}
```

Il ne peut y avoir de constructeur dans une interface !

Impossible de faire appel à `new`

Interfaces II

Attribution d'une interface à une classe :

```
public class UneClasse implements UneInterface {  
    ...  
}
```

Exemple avec `Animal` et l'interface `Graphique`

```
public class Animal implements Graphique {  
    public void dessiner() { ... }  
}
```

Attention !

Pour être **instanciable**, une classe **doit** redéfinir **toutes** les méthodes des interfaces !

Variables de type interface

Une interface attribue un type supplémentaire à une classe d'objets, on peut donc :

- déclarer une variable de type interface
- y affecter un objet d'une classe implémentant cette interface
- éventuellement faire un transtypage explicite vers l'interface si besoin est

Variables de type interface

```
Graphique graph;  
Animal = new Animal(...);  
graph = animal;  
Entite entite = new Decor(...);  
graph = (Graphique)entite; // transtypage obligatoire  
!  
// en effet Entite implemente Graphique !
```

Plusieurs interfaces

Une classe peut implémenter plusieurs interfaces :

- **mais étendre une seule classe !** (Java = héritage simple)
- il faut alors séparer les interfaces par des virgules après le mot clé `implements`

Implémentation de plusieurs interfaces

```
public class Personnage implements Graphique, Interactif,  
    Mobile                                ajouter 3 interfaces  
{  
    // code de la classe  
}
```

Héritage d'interfaces

Hiérarchie d'interfaces

- de la même manière qu'une classe, une interface peut avoir des sous-interfaces (via le mot-clé **extends**)
- une sous-interface hérite de ses "super-interfaces" :
 - ▶ leurs **types**
 - ▶ **toutes leurs méthodes abstraites**
 - ▶ **toutes leurs constantes**
- peut définir de nouvelles constantes et de nouvelles méthodes abstraites
- une classe qui implémente une interface doit définir toutes les méthodes de l'interface et des interfaces dont elle hérite

Hiérarchie d'interfaces

```
public interface Interactif{ ... }  
public interface GestionSouris extends Interactif { ... }  
public interface GestionClavier extends Interactif { ... }
```

Quelques interfaces Java

Interfaces utiles :

- `Collection` : méthode abstraite `iterator()`
- `Iterable<E>` : méthodes abstraites `add(E e)`, `clear()`, `size()`, etc.
- `List` : méthodes abstraites (en plus de celles de `Collection` et `Iterable`) `get(int index)`, `remove(int index)`, etc.
- `Comparable<T>` : méthode abstraite `compareTo(T o)`
- `Runnable` : méthode abstraite `run()`

Allez voir la JavaDoc !!!!

Interfaces en UML

Représentation des interfaces en UML

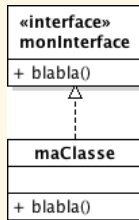


Figure: Représentation d'une interface et de son implémentation par une classe.

Interfaces – Bilan

Une interface :

Est un moyen d'attribuer des composants communs à des classes non-liées par des relations d'héritage \Rightarrow les composants de l'interface seront disponibles dans chaque classe qui l'implémente

Composants possibles d'une interface :

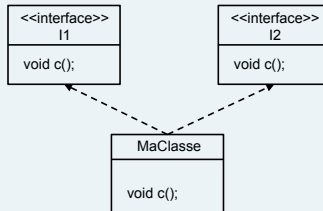
- ❶ Constantes (i.e. variables statiques finales, **assez rare**) \Rightarrow
Ambiguïté possible, nom unique exigé
- ❷ Méthodes abstraites (**très courant**) \Rightarrow
 - ▶ chaque classe qui implémente l'interface est obligée d'implémenter chaque méthode abstraite déclarée dans l'interface si elle veut pouvoir être instanciée
 - ▶ une façon de garantir que certaines classes ont certaines méthodes, sans passer par des classes abstraites
 - ▶ aucune ambiguïté car sans instructions !

Interfaces – Bilan

Une interface :

Est un moyen d'attribuer des composants communs à des classes non-liées par des relations d'héritage \Rightarrow les composants de l'interface seront disponibles dans chaque classe qui l'implémente

Exemple illustratif



OK car l'implémentation n'est pas dans l'interface

Figure: Implémentation de deux interfaces fournissant la même méthode abstraite par un objet.

Interfaces – Bilan II

Rappel héritage :

Nous avons vu que l'héritage permet de mettre en place une relation de type **“est-un”** entre deux classes

Rappel délégation :

Lorsqu'une classe a pour attribut un objet d'une autre classe, il s'établit entre les deux classes une relation de type **“a-un”** moins forte que l'héritage (on parle de **délégation**)

Une interface :

- permet de s'assurer qu'une classe respecte un **contrat**
- elle met en place une relation de type **“se-comporte-comme”** :
un **Personnage** **“est-une”** entité du jeu, elle **“se-comporte-comme”** un objet graphique, interactif et mobile