

Programmation Orientée Objet - Licence TIS CM4/9

Lancelot PECQUET

Lancelot.Pecquet@math.univ-poitiers.fr



Poitiers, le 23/01/2006

1 Classes abstraites

2 Interfaces

Rappel sur la séance précédente

La fois précédente, nous avons vu :

- ① l'héritage simple

Aujourd'hui, nous voyons :

- ① les classes abstraites
- ② l'héritage multiple et les interfaces

Méthode abstraite

- méthode abstraite = méthode partiellement définie :
 - nom de la méthode
 - signature de la méthode (types, nombres et ordres des arguments)
 - la réalisation (le corps) n'est pas spécifié à ce stade
 - il sera spécifié par héritage
- la méthode peut être invoquée comme une méthode normale
- objectif : structurer le code et le factoriser

Caricature du parisien

Hypothèse 1

Toute sa vie, voici la journée d'un parisien :

- métro
- boulot
- dodo

En Java

```
1 ... class parisien{
2   private int age_max;
3   ...
4   public void vit(){for(int i=0;i<age_max*365;i++){metro(); boulot(); dodo();}}
5 }
```

Caricature du parisien

Méthodes

- dans le métro, tous les parisiens ont le même comportement

```
1 private final void metro(){
2   System.out.println("beeeeeeep_clac_pshhh_VRRRRR...");
3 }
```

- dans leur lit, tous les parisiens ont le même comportement

```
1 private final void dodo(){System.out.println("Zzzzz...");}
```

- le boulot diffère pour chacun d'entre eux

Mot-clé abstract

Définition

- pour définir une méthode abstraite, on préfixe sa spécification du mot-clé **abstract**
- classe abstraite : contient au moins une méthode abstraite
- les classes abstraites sont préfixées du mot-clé **abstract**
 - non-instanciables (pas de `new...`)
 - constructeurs (utilisation de `super()` dans les héritières)

Exemple

```
1 abstract class parisien{...
2 abstract private void boulot(); // Methode abstraite, depend du parisien
3 ...}
```

Modélisation du parisien en Java

Fichier parisien.java

```
1 abstract class parisien{
2 private int age_max;
3 private void metro(){System.out.println("beeeeee_clac_pshhh_VRRRRR...");}
4 abstract private void boulot(); // Methode abstraite, depend du parisien
5 private void dodo(){System.out.println("Zzzzz...");}
6
7 // constructeur initialisant la duree de vie a environ 75 ans
8 public parisien(){
9 age_max = (int)(75.0 + new java.util.Random().nextGaussian()*10.0);
10 }
11
12 // On peut utiliser boulot() meme si on ne le connait pas, a ce stade:
13 public void vit(){
14 for(int i=0; i<age_max*365; i++){metro(); boulot(); dodo();}
15 }
16 }
```

Dérivation de deux instances de parisien

Fichier commercial.java

```
1 class commercial extends parisien{
2     private int pecule;
3     public commercial(){super(); pecule = 0;}
4     private void boulot(){System.out.println("blabla"); pecule += 1000;}
5 }
```

Fichier chercheur.java

```
1 class chercheur extends parisien{
2     private int questions, reponses;
3     public chercheur(){super(); questions = 0; reponses = 0;}
4     private void boulot(){
5         System.out.println("grat_grat"); reponses++; questions += 100;
6     }
7 }
```

Utilisation des classes

Fichier Main.java

```
1 ...
2 commercial A = new commercial(); A.vit();
3 chercheur B = new chercheur(); B.vit();
4 // parisien p = new parisien(); // erreur car la classe est abstraite!
5 ...
```

Sortie (console)

```
1 beeeep clac pshhh VRRRRR...
2 blabla
3 Zzzzz...
4 beeeep clac pshhh VRRRRR...
5 blabla
6 Zzzzz...
```

Questions ?



Petit exercice (sans le poly !)

Sachant que :

- la jeunesse de tous les parisiens est la même (cris de bébés, croissance, études)
- à la fin de sa vie, le parisien fait le bilan de sa vie (le commercial compte son pécule, le chercheur regarde combien il avait trouvé de réponses et de problèmes)
- la mort de tous les parisiens est la même (un dernier souffle)

Modifier les classes concernées en conséquence

Une correction

Fichier parisien.java

```
1 abstract class parisien{
2     private int age_max;
3     private final void jeunesse () {System.out.println("Ouin!!!<<grandit>>,<<etudes>>);}
4     private final void metro () {System.out.println("beeeeeeplac_pshhh_VRRRRR..");}
5     private abstract void boulot ();
6     private final void dodo () {System.out.println("Zzzzz...");}
7     private abstract void bilan ();
8     private final void meurt () {System.out.println("Argh!");}
9     public parisien () {
10         age_max=(int) (75.0 + new java.util.Random ().nextGaussian ()*10.0);
11     }
12     public void vit () {
13         jeunesse ();
14         for (int i=0; i<age_max*365; i++) {metro (); boulot (); dodo ();}
15         meurt (); bilan ();
16     }
17 }
```

Une correction (suite)

Fichier commercial.java

```
1 class commercial extends parisien{
2     private int pecule;
3     public commercial () {super (); pecule=0;}
4     private void boulot () {System.out.println("blabla"); pecule += 1000;}
5     private void bilan () {System.out.println("Le_defunt_avait_gagne_" + pecule);}
6 }
```

Fichier chercheur.java

```
1 class chercheur extends parisien{
2     private int questions, reponses;
3     public chercheur () {super (); questions=0; reponses=0;}
4     private void boulot () {
5         System.out.println("grat_grat"); reponses++; questions += 100;
6     }
7     private void bilan () {System.out.println("Le_defunt_avait_trouve_"
8         + (float)reponses/(float)questions*10 + "%des_reponses");}
9 }
```

Questions ?



Héritage multiple

En Java, l'héritage est **simple**

- plusieurs classes descendantes
- **une seule** ascendante (sauf Object qui n'en a pas)

Héritage **multiple**

- **plusieurs** classes ascendantes

Exemple d'animaux (interactif, sans le poly..)

Hypothèses

- tout animal mange
- les carnivores sont des animaux qui traquent le gibier
- les piscivores sont des animaux qui traquent le poisson
- les frugivores sont des animaux qui cueillent les fruits
- omnivore = carnivore + piscivore + frugivore.

Question 1

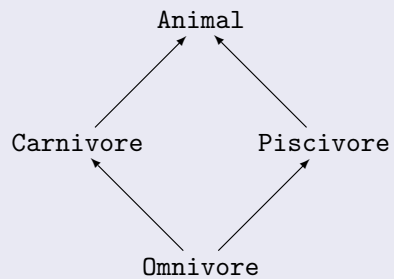
Comment modéliser cette faune ?

Question 2

Comment un omnivore traque-t-il ses proies ?

Le *Diamond of death* (*Dod*)

Le diagramme en diamant



Problème

Quelle traque ?

Solution de Java : les interfaces

Définition

Lorsqu'une classe ne contient que :

- des méthodes `public abstract`
- des champs `public final`

alors on peut la déclarer comme **interface**.

Intérêt

- toutes les méthodes sont abstraites
- pas de conflit dans le choix de réalisation d'une méthode
- héritage multiple possible et sans problème

Remarques

Limitations du mécanisme

- perte de puissance de l'héritage
- pas de factorisation de code possible
- on pourrait être plus fin dans l'analyse des conflits
- C++ autorise l'héritage multiple de classes (délicat)

Finalité du mécanisme d'interface

- structurer le code
- **indiquer** que certaines fonctionnalités vont être implémentées. . .

Mot-clé interface

Structure d'une interface

- tous les champs sont implicitement `public final`
- toutes les méthodes sont implicitement `public abstract`
- le nom et le type des arguments des méthodes doivent être explicités

Définition d'une interface

```
1 interface Animal{  
2     void Mange();  
3 }
```

Héritage d'interfaces

Héritage simple

```
1 interface Carnivore extends Animal{  
2     void Traque();  
3 }  
4 interface Piscivore extends Animal{  
5     void Traque();  
6 }  
7 interface Frugivore extends Animal{  
8     void Cueille();  
9 }
```

Héritage multiple

```
1 interface Omnivore extends Carnivore, Piscivore, Frugivore{}
```

Implémentation d'une interface

Définition

Plutôt de dire qu'une classe "hérite" d'une interface, on dit qu'elle **l'implémente**.

Remarque

Une classe peut implémenter directement plusieurs interfaces (sans forcément devoir créer une interface héritière de toutes ces interfaces)

Exemples d'implémentation

Fichier guepard.java

```
1 class guepard implements Carnivore{
2     /* Les 4 lignes suivantes sont identiques ou presque dans humain.java
3     mais ne peuvent être factorisées dans Animal car c'est une interface:
4     // perte de puissance par rapport aux classes (même abstraites)
5     private String nom;
6     private setNom(String nom){this.nom=nom;}
7     public getNom(){return nom;}
8     public guepard(String nom){setNom(nom);}
9
10    public void Traque(){
11        System.out.println(getNom() + " court après la proie,"
12        + " lui saute dessus et la mord à la gorge"
13        + " jusqu'à ce que mort s'ensuive");
14    }
15    public void Mange(){
16        System.out.println(getNom() + " déchiquète goulument la proie");
17    }
18 }
```

Exemples d'implémentation (suite)

Fichier humain.java

```
1 class humain implements Omnivore{
2     private String nom;
3     private setNom(String nom){this.nom=nom;}
4     public getNom(){return nom;}
5     public humain(String nom){setNom(nom);}
6     public void Traque(){System.out.println(getNom() + "_va_au_supermarche");};
7     public void Cueille(){Traque();};
8     public void Mange(){
9         System.out.println(getNom() + "_deplace_la_fourchette_en_alternance_"
10            + "entre_sa_bouche_et_son_assiette");}
11 }
```

Alternative (moins propre ici car Omnivore existe)

```
1 class humain implements Carnivore, Piscivore, Frugivore{...}
```

Résultat

Fichier Main.java

```
1 humain alfred = new humain("alfred"); alfred.Traque(); alfred.Mange();
2 guepard groarr = new guepard("groarr"); groarr.Traque(); groarr.Mange();
```

Sortie (console)

```
1 alfred va au supermarche
2 alfred deplace la fourchette en alternance entre sa bouche et son assiette
3 groarr court apres la proie, lui saute dessus et la mord a la gorge
4 jusqu'a ce que mort s'ensuive
5 groarr dechiquete goulument la proie
```

Remarque sur l'implémentation dans l'héritage multiple

Attention : le type de retour n'est pas discriminant

```
1 interface I1{int f(int i, int j);}
2 interface I2{int f(int i, int j);}
3 interface I3{double f(int i, int j);}
4 class C implements I1,I2; // OK
5 class D implements I1,I3; // erreur de compilation
```

Interfaces indicatrices, opérateur instanceof

Principe

- Mécanisme de balisage avec `M instanceof I`
- `C.getInterfaces()` renvoie les interfaces (`Class[]`) supportées par `C`.

Ex, dans Main.java

```
1 System.out.println(alfred instanceof Omnivore); // true
2 System.out.println(alfred instanceof Carnivore); // true
3 System.out.println(groarr instanceof Piscivore); // false
```

Utilisation de instanceof dans clone()

- `M.clone()` est toujours possible (défini dans `Object`)
- si `C` n'implémente pas `Cloneable` alors il y a une `CloneNotSupportedException`

Remarque de conception

Attention

Ne pas remplacer l'héritage par des tests de `instanceof`.

Ex à ne pas faire : l'héritage est anticipé (mal : insectivore?)

```
1 abstract class Animal {
2     public void mange() {
3         if (this instanceof Carnivore) {
4             System.out.println(getNom() + "_dechiquete_goulument_la_proie");
5         } else if (this instanceof Omnivore) {
6             System.out.println(getNom() + "_deplace_la_fourchette_en_alternance_"
7                 + "entre_sa_bouche_et_son_assiette");
8         } else { ... }
9     }
10 }
```

Questions ?



Conclusion

Aujourd'hui, nous avons vu :

- 1 Classes abstraites
- 2 Interfaces

La séance prochaine, nous verrons :

- 4 exceptions
- 2 package
- 3 javadoc
- 4 conteneurs et itérateurs