Chap.1-2 – Paradigmes de Conception (suite)

V. Deslandres ©

S1-10 - Qualité de Développement-1

BUT Informatique - ASPE

IUT de Lyon - Université Lyon 1

Principes élémentaires

Bonne écriture de code

Préambule : exercice

Quelles sont les **3 mauvaises habitudes de code bien connues,** qu'il vous faut absolument éviter dorénavant (en BUT2) ?

1. Mal _____ ses variables, classes, fonctions

   Les _____ choisis aident à comprendre le code en plus des commentaires.

2. Ecrire des _____ __ _____ trop longs·longues

   Comme un article de presse qui utilise les _____, nos _____ doivent soigner nos yeux : on ne peut pas lire une _____ _____ de code.

3. Avoir de grosses _____ ou _____

   Se limiter à _____ en général par _____

Importance du nommage

A éviter	Mieux
struct Point {	typedef double **Coord**;
double x, y;	typedef double **Reel_0_1**;
double poids; // entre 0 et 1	struct Point {
};	**Coord** x, y;
	Reel_0_1 poids;
	};

Découpage d'une ligne

Si une ligne **dépasse 80 symboles**, il faut la découper, plusieurs possibilités :

- Si l'indentation est trop grande, il faut subdiviser la fonction.

- Si l'expression est trop compliquée, il faut utiliser des variables intermédiaires qui rendront le code facilement compréhensible si les noms sont bien choisis.

- Sinon, découper aux endroits logiques (opérateurs de faible priorité) en tentant de faire apparaître des alignements verticaux si c'est possible.

https://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TRANS_COMP_IMAGE/prog.html#performance-d-execution

Avantage d'un code court

Un code court :

- Comporte moins d'erreur.
- Est plus facilement lisible.
- Est plus facilement modifiable.
- Est plus vite écrit.

https://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TRANS_COMP_IMAGE/prog.html#performance-d-execution

Autres règles d'une bonne conception

1. **Ne pas mettre des accesseurs / mutateurs** pour tous les attributs systématiquement, sans conditions (règles Métier)

   On perdrait les bénéfices de l'encapsulation

Rappel : pourquoi encapsuler ?

Au sein d'une classe en POO, on encapsule pour mieux **contrôler** les attributs et méthodes :

- Rendre certains attributs en **lecture seule** (seul get() accessible en public), ou en **écriture seule** (seul set() accessible)
- **Plus de flexibilité** : le développeur peut changer une partie du code sans que cela affecte les autres parties (puisque pas d'accès direct)
- Améliorer la **sécurité** des données

C'est **exactement les mêmes raisons** qui font qu'on va encapsuler une classe dans une autre, un composant dans un autre, une application dans une autre

- Le but est de contrôler l'accès à l'élément encapsulé, cacher les détails de son implémentation

Autres règles d'une bonne conception (2)

2. **Principe DRY** - **Don't Repeat Yourself** : jamais de copier/coller de code

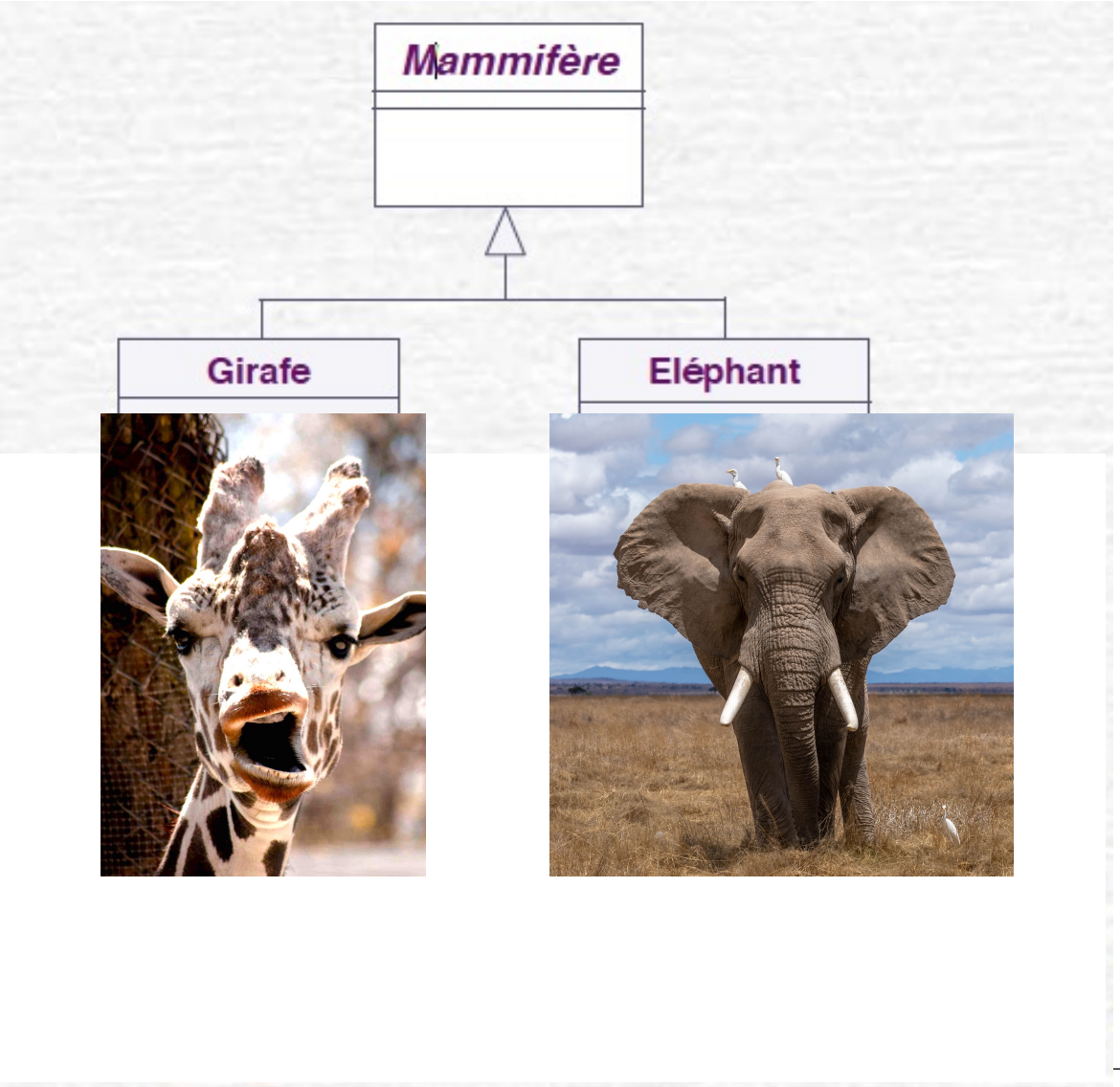
3. Ne jamais dériver une classe en n'exploitant que **certains attributs** et **méthodes**

4. Préférer la **composition** à **l'héritage**

illustrés ci-après…

Règle#3 : « Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »

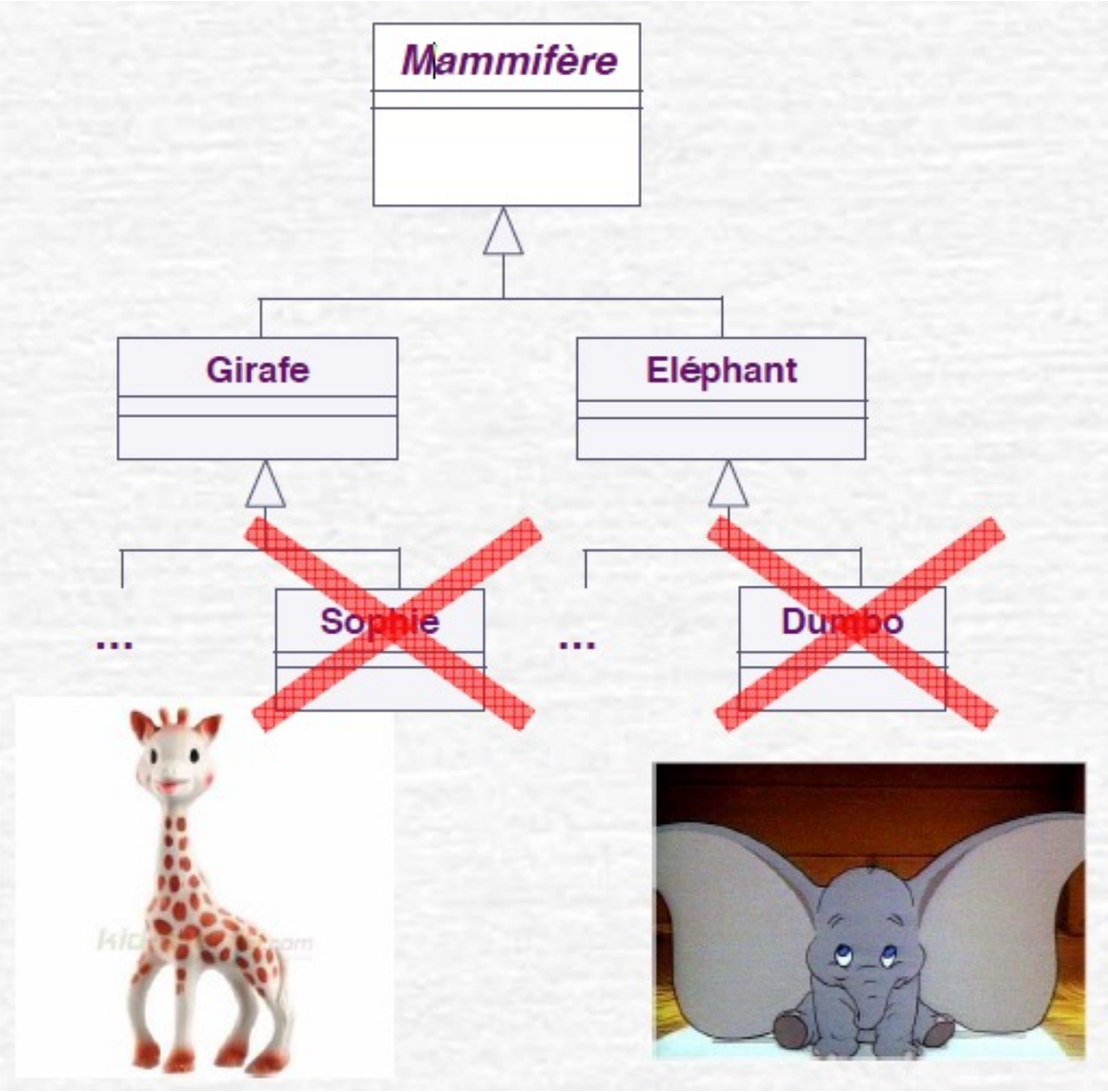
Imaginons un logiciel de simulation d'une **réserve animalière**, avec des girafes et des éléphants.



« Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »

Pour un logiciel de moulage du jouet *Girafe Sophie,* on décide de reprendre la classe *Girafe* **pour dessiner sa robe** : on crée la sous-classe Sophie uniquement pour cela…

Que se passe-t-il si une classe Client applique le **comportement** de la classe Girafe (reproduction, alimentation) à la *Girafe Sophie* ??



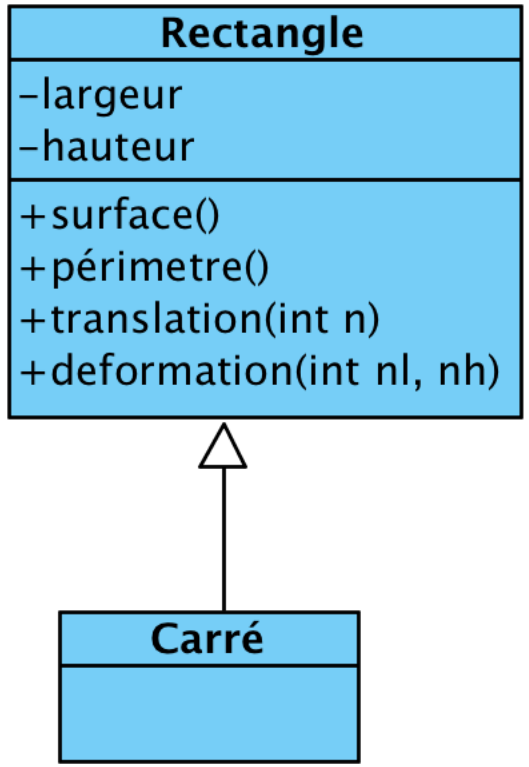
Règles en cas d'héritage

- Les **pré-conditions** définies par les sous-classes ne doivent pas être plus restrictives que celles héritées.

- Les **post-conditions** définies par les sous-classes ne doivent pas être moins larges que celles héritées

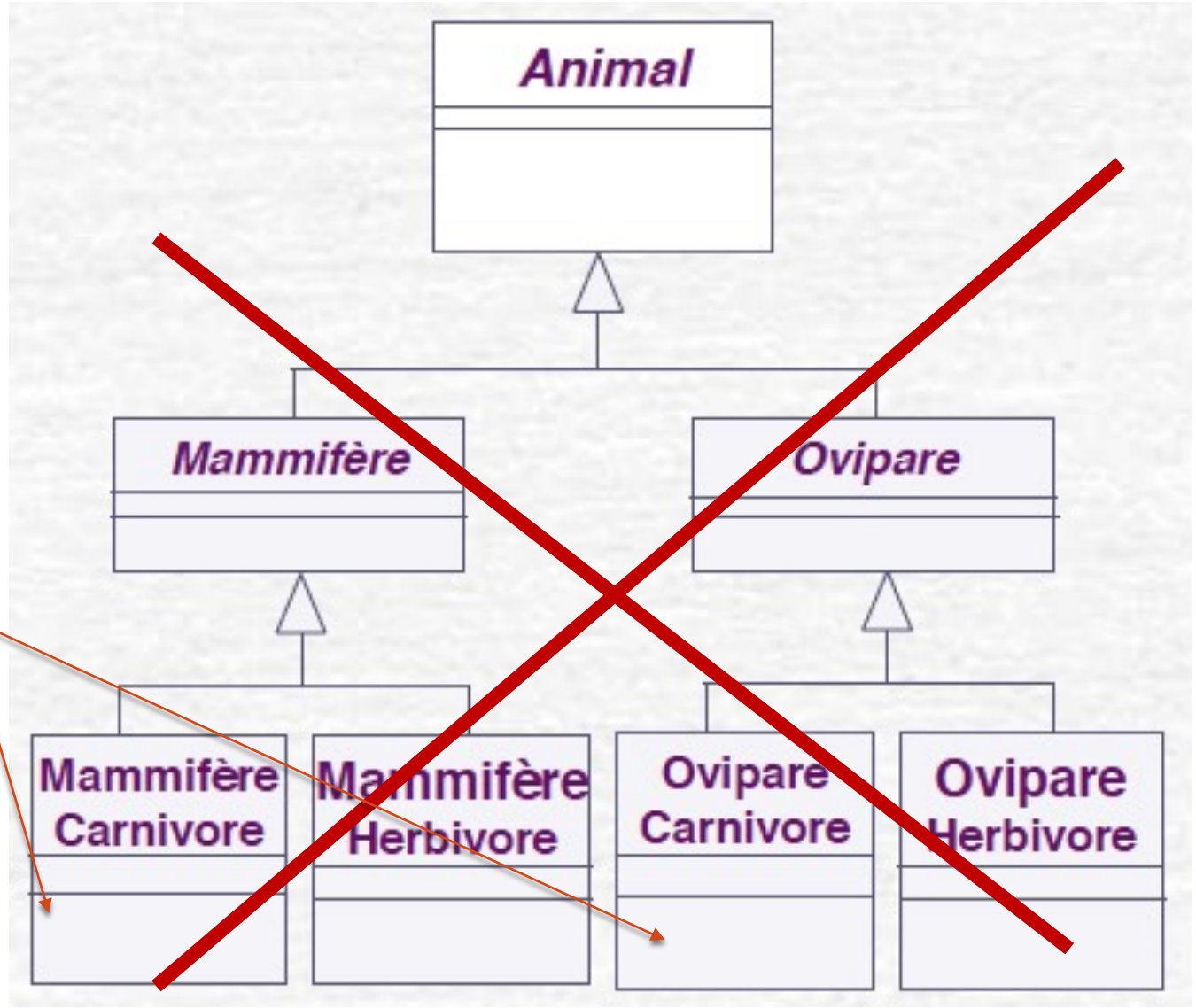
Exercice pré/post conditions

- Que penser de cette conception ?



Dans le constructeur d'un carré, on vérifie que les côtés ont la même taille

Règle#4 :« Préférer la composition à l'héritage »



code de Carnivore dupliqué

- Explosion combinatoire
- Duplication de code

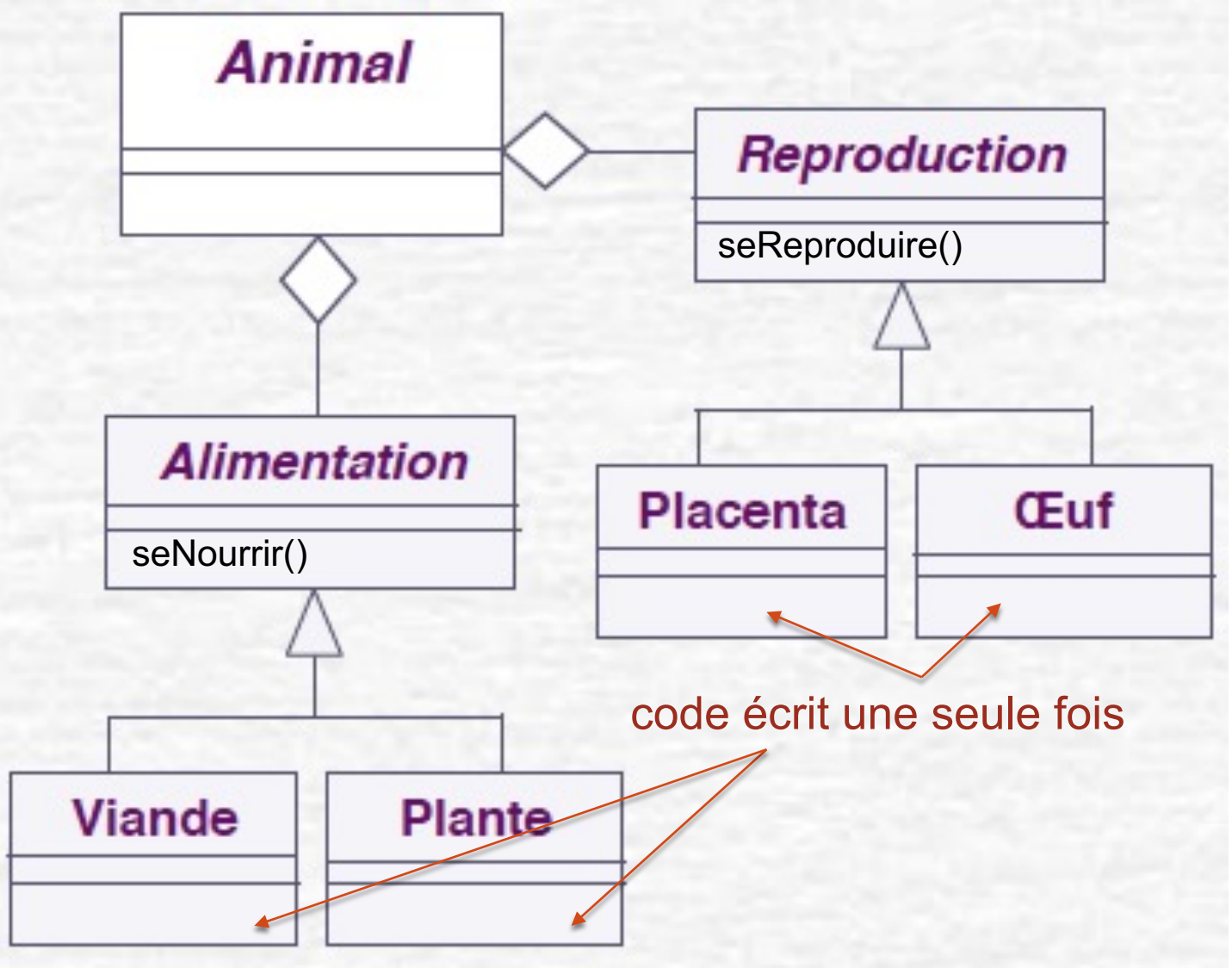
Règle#4 :« Préférer la composition à l'héritage »

On considère qu'un Animal possède 2 caractéristiques intrinsèques, qui varient d'un animal à l'autre :

- son mode d'Alimentation
- son mode de Reproduction

On définit ces modes à chaque instanciation d'Animal :

```
Animal garfield = new Animal();
garfield.setModeReproduction( new Placenta() );
garfield.setModeAlimentation( new Viande() );
…
garfield.getModeAlimentation().seNourrir();
```

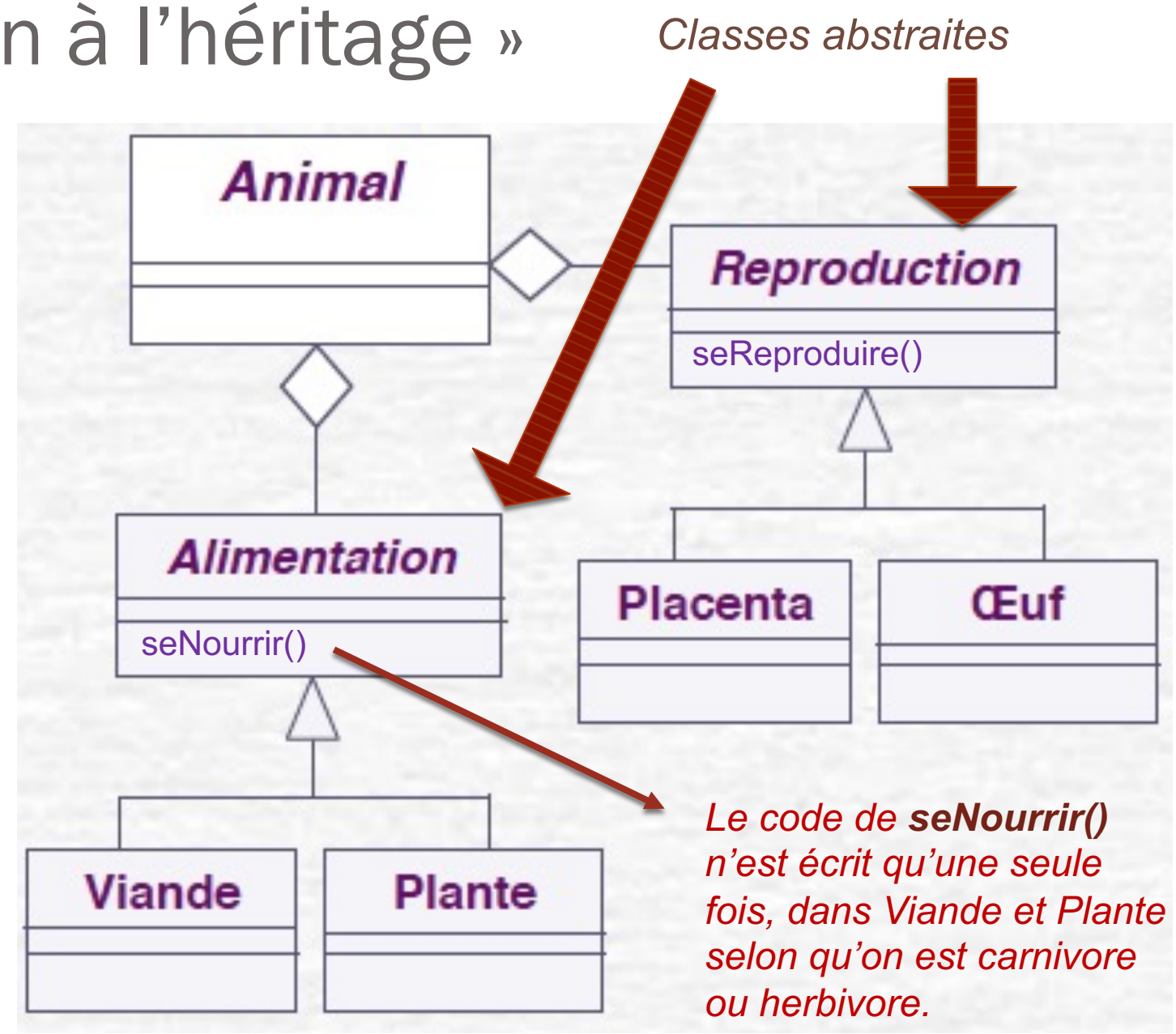


Préférer la composition à l'héritage »

Encore appelé principe d'indirection ou de délégation :

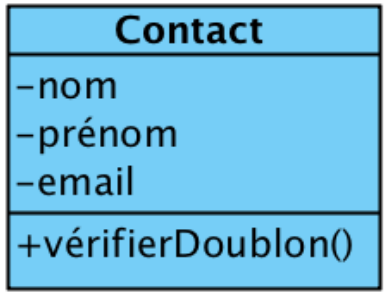
on délègue à la classe **Alimentation** le comportement seNourrir() de l'Animal

La classe **Alimentation** est dite *encapsulée* dans Animal.



Conception / Codage : importance des structures de données

- Certaines structures de données peuvent traduire et **simplifier les méthodes imaginées en phase d'analyse**
- Ex. : une classe d'analyse **Contact** avec une méthode **vérifierDoublon()**
  - si l'ID est défini par nom+prénom : il suffit de placer les objets Contact dans un conteneur *set* d (*set* ne tolère pas les doublons)
- Ex.: pour un **contrôle d'accès de salles**, on choisira une *HashTable* avec le numéros de salle (*clef*) et le code d'accès (valeur: *true/false*).



- Il est donc important de **bien connaître** les structures de données évoluées : *collections* Java, *arbres bicolores*, *skip-list*, etc.