

# Module Programmation Objet

## Chapitre 3 - Typage Simple et Héritage

Pascal André, Gilles Ardourel

Université de Nantes

2010

Projet DVD Miage



Rappels

Sous typage et Héritage

Syntaxe

Encapsulation, l'héritage de propriétés

Redéfinition de méthodes

Redéfinition et typage

Résumé

## Rappels

Le typage

Protection

Sous typage et Héritage

Syntaxe

Encapsulation, l'héritage de propriétés

Redéfinition de méthodes

Redéfinition et typage

# Le typage statique

## Type

Un type détermine un ensemble de messages applicables.

# Le typage statique

## Type

Un type détermine un ensemble de messages applicables.

Lorsque

- ▶ une variable
- ▶ un retour de méthode
- ▶ un attribut

est déclaré d'un type  $T$ , l'objet référencé doit pouvoir répondre aux messages déclarés pour  $T$ .

# Le typage statique

## Type

Un type détermine un ensemble de messages applicables.

Lorsque

- ▶ une variable
- ▶ un retour de méthode
- ▶ un attribut

est déclaré d'un type  $T$ , l'objet référencé doit pouvoir répondre aux messages déclarés pour  $T$ .

La rupture du contrat donne lieu à une erreur à la compilation ou à l'exécution.

# Types en Java

- ▶ primitifs
- ▶ classes
- ▶ interfaces

Identifiés par un nom et caractérisés par un ensemble de signatures

## Exemples : typage et protection

### Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
v1.accueillir(v1);  
p12.accueillir(p2);  
p12.taille=1;  
p12.accueillir(v2);
```

# Exemples : typage et protection

## Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

## Exemples : typage et protection

### Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
v1.accueillir(v1);  
p12.accueillir(p2);  
p12.taille=1;  
p12.accueillir(v2);  
}
```

# Exemples : typage et protection

## La protection

permet d'éviter les envois de messages que l'émetteur n'a pas le droit d'envoyer au receveur.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
v1.accueillir(v1);  
p12.accueillir(p2);  
p12.taille=1;  
p12.accueillir(v2);
```

# Exemples : typage et protection

## La protection

permet d'éviter les envois de messages que l'émetteur n'a pas le droit d'envoyer au receveur.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
v1.accueillir(v1);  
p12.accueillir(p2);  
p12.taille=1;  
p12.accueillir(v2);
```

# Protection

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
- ▶ Respect des spécifications
- ▶ Respect de règles de génie logiciel

# Protection

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité  
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications
- ▶ Respect de règles de génie logiciel

# Protection

## Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité  
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications  
→ parking est le seul à pouvoir "rendre disponible" une place
- ▶ Respect de règles de génie logiciel

# Protection

## Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité  
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications  
→ parking est le seul à pouvoir "rendre disponible" une place
- ▶ Respect de règles de génie logiciel  
→ masquage de l'implémentation

# Protection

## Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité  
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications  
→ parking est le seul à pouvoir "rendre disponible" une place
- ▶ Respect de règles de génie logiciel  
→ masquage de l'implémentation  
→ réduction de couplage

# Le contrat de sous typage

## Sous Type

Si  $T$  est un sous-type de  $U$  alors l'ensemble de messages accepté par le type  $T$  inclus les messages acceptés par  $U$ .

# Le contrat de sous typage

## Sous Type

Si  $T$  est un sous-type de  $U$  alors l'ensemble de messages accepté par le type  $T$  inclus les messages acceptés par  $U$ .

## Substituabilité

Si  $T$  est un sous-type de  $U$  et  $t$  un objet de type  $T$ ,  $t$  doit pouvoir être substitué à un objet de type  $U$ .

# Le contrat de sous typage

## Sous Type

Si  $T$  est un sous-type de  $U$  alors l'ensemble de messages accepté par le type  $T$  inclus les messages acceptés par  $U$ .

## Substituabilité

Si  $T$  est un sous-type de  $U$  et  $t$  un objet de type  $T$ ,  $t$  doit pouvoir être substitué à un objet de type  $U$ .

Avantages : Une code (méthode ou constructeur) utilisant un type  $U$  fonctionnera si on lui passe des sous types de  $U$ . Extensibilité.

# Côté client

Les clients classiques sont

# Côté client

Les clients classiques sont

- ▶ le déclarant

# Côté client

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers

# Côté client

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs

# Côté client

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

# Héritage

## Objectif

- ▶ factoriser du code
- ▶ éviter les répétitions.

# Héritage

## Objectif

- ▶ factoriser du code
- ▶ éviter les répétitions.

Lorsqu'une classe hérite d'une autre, elle

# Héritage

## Objectif

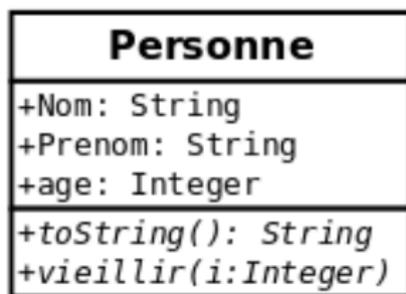
- ▶ factoriser du code
- ▶ éviter les répétitions.

Lorsqu'une classe hérite d'une autre, elle Permettre une modification incrémentale.

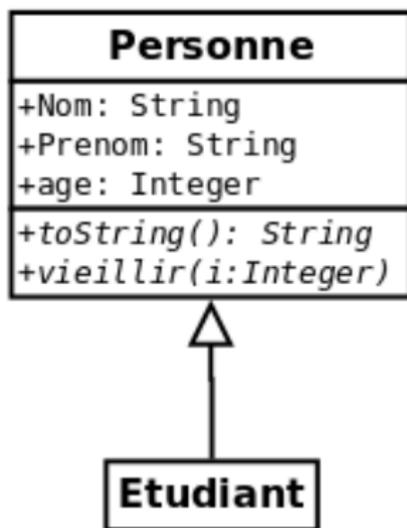
- ▶ Ajout d'attributs
- ▶ Ajout de méthodes

Les nouvelles méthodes peuvent utiliser les anciennes.

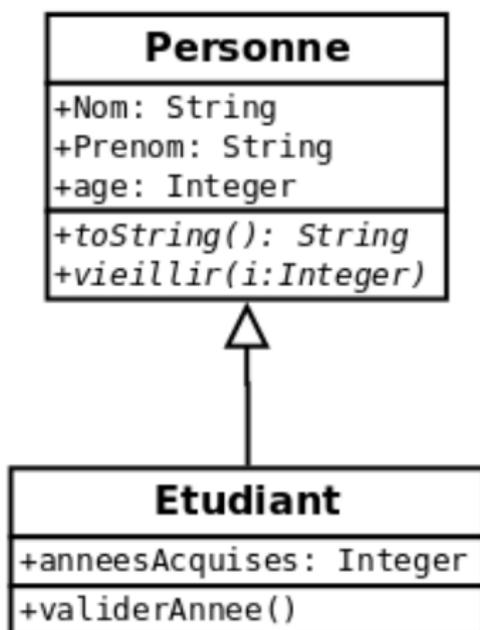
# Héritage - UML



## Héritage - UML



## Héritage - UML



Rappels

Sous typage et Héritage

**Syntaxe**

Encapsulation, l'héritage de propriétés

Redéfinition de méthodes

Redéfinition et typage

Résumé

# Héritage en Java

## Le mot-clé *extends*

- ▶ Pour indiquer que la classe **Fille** hérite de la classe **Mere**, la déclaration de la classe **Fille** doit être :  
`class Fille extends Mere {...}`
- ▶ On dit alors que **Mere** est la super-classe et que **Fille** est la sous-classe.
- ▶ Une classe a toujours une super-classe, même si celle-ci n'est pas déclarée. Dans ce cas, il s'agit de la classe **Object**, la super-classe commune de toutes les classes Java.
- ▶ En java, une sous classe est un sous type. Le sous typage est automatiquement respecté grâce à l'héritage.

# Héritage en C++

- ▶ Pour indiquer que la classe **Fille** hérite de la classe **Mere**, la déclaration de la classe **Fille** doit être :  
`class Fille : public Mere {...};`
- ▶ Dans le jargon C++, les super-classes sont aussi appelées les classes de base et les sous-classes sont appelées les classes dérivées.
- ▶ Bien que déconseillé, il est possible d'hériter de manière autre que **public** : e.g. **protected** et **private**. Dans ces cas, il y aura héritage, mais pas sous-typage.

# Héritage en Smalltalk

Par un *envoi de message*

- ▶ Pour indiquer que la classe **Fille** hérite de la classe **Mere**, la déclaration de la classe **Fille** doit être effectuée par un envoi de message à la classe **Mere!** e.g. :

```
ArithmeticValue subclass: #Point
instanceVariableNames: 'x y '
classVariableNames: ''
poolDictionaries: ''
category: 'Graphics-Geometry'
```

- ▶ On dit alors que **Mere** est la super-classe et que **Fille** est la sous-classe.
- ▶ Une classe a toujours une super-classe, par défaut on envoie le message à la classe **Object**, la super-classe commune de toutes les classes Smalltalk.
- ▶ En Smalltalk, une sous classe est un sous type. Le sous typage est automatiquement respecté grâce à l'héritage.

Rappels

Sous typage et Héritage

Syntaxe

Encapsulation, l'héritage de propriétés  
Type Statique et Dynamique

Redéfinition de méthodes

Redéfinition et typage

Résumé

## Héritage de propriétés privées

- ▶ Toutes les propriétés privées sont héritées par la sous-classe.
- ▶ Cependant, elles ne sont pas accessibles.

```
class Personne {  
    private int age;  
    public int getAge() {  
        return age;  
    }  
}  
class Student extends Personne {  
    public String toString() {  
        return "Student, "+age+  
            " years old";  
    }  
}
```

## Héritage de propriétés privées

- ▶ L'attribut **age** n'est pas accessible à l'intérieur de la classe étudiant.
- ▶ La méthode **getAge()** fonctionne correctement, car bien qu'inaccessible, l'attribut **age** a bien été hérité.
- ▶ La méthode **toString()** ne marche pas.

```
class Personne {  
    private int age;  
    public int getAge() {  
        return age;  
    }  
}  
class Student extends Personne {  
    public String toString() {  
        return "Student, "+age+  
            " years old";  
    }  
}
```

# Héritage de propriétés publiques

- ▶ Les propriétés publiques sont accessibles par les sous-classes.

```
class Personne {  
    public String getName() {  
        return name;  
    }  
    (...)  
}  
  
class Student extends Personne {  
    public String toString() {  
        return "Student: " +  
            this.getName();  
    }  
}
```

## Héritage de propriétés protégées

- ▶ L'héritage engendre l'utilisation d'un troisième type de visibilité, les propriétés protégées (`protected`).
- ▶ Ces propriétés sont accessibles seulement à l'intérieur de la classe et de ses sous-classes.

```
class Personne {  
    protected int age;  
    public int getAge() {  
        return age;  
    }  
}  
class Student extends Personne {  
    public String toString() {  
        return "Student, "+age+  
            " years old";  
    }  
}
```

# Héritage et Sous-typage

Toutes les méthodes d'une classe  $U$  sont héritées par ses sous-classes  $T$ .

# Héritage et Sous-typage

Toutes les méthodes d'une classe  $U$  sont héritées par ses sous-classes  $T$ .

Les méthodes de  $T$  incluent celles de  $U$

Si elles sont au moins aussi accessibles qu'elles l'étaient dans  $T$ , alors il y a sous-typage.

# Héritage et Sous-typage

Toutes les méthodes d'une classe  $U$  sont héritées par ses sous-classes  $T$ .

Les méthodes de  $T$  incluent celles de  $U$

Si elles sont au moins aussi accessibles qu'elles l'étaient dans  $T$ , alors il y a sous-typage.

C'est le cas en Java et en C++ avec un héritage Public

## Exemple

```
Personne pers= new Personne();  
Etudiant etu = new Etudiant();  
  
pers.seGarer(); // erreur de type  
pers.vieillir(2); // ok, défini dans Personne  
  
etu.validerAnnee(); // ok, défini dans Etudiant  
etu.vieillir(1); // ok, défini dans Personne  
  
pers.validerAnnee(); // non défini dans Personne
```

## Exemple

```
Personne pers= new Personne();
Etudiant etu = new Etudiant();

pers.seGarer(); // erreur de type
pers.vieillir(2); // ok, défini dans Personne

etu.validerAnnee(); // ok, défini dans Etudiant
etu.vieillir(1); // ok, défini dans Personne

pers.validerAnnee(); // non défini dans Personne

pers=etu; // Ok, la variable pers est de type Personne
// et etu est d'un sous-type de Personne
```

# Type Dynamique

Un objet est typé.

- ▶ Statiquement,  
*en fonction du type de sa référence.*
- ▶ Dynamiquement,  
*en fonction de la classe dont le constructeur a créé l'objet.*

# Type Dynamique

Un objet est typé.

- ▶ Statiquement,  
*en fonction du type de sa référence.*
- ▶ Dynamiquement,  
*en fonction de la classe dont le constructeur a créé l'objet.*
- ▶ Le typage statique permet au compilateur  
*de vérifier que les messages envoyés sont corrects et autorisés.*
- ▶ Le typage dynamique permet la liaison dynamique :  
*l'appel de la méthode la plus spécifique à l'objet. A voir dans la section redéfinition*

# Type Dynamique

```
Personne etu; //type statique
etu = new Etudiant(); //type dynamique

etu.vieillir(1); // ok, défini dans Personne
etu.validerAnnee(); // erreur : non défini dans Personne
```

Le compilateur ne peut se fier qu'au type statique. Pour s'en convaincre, considérer le code suivant :

```
Personne etu=null;
if (Math.random()>0.5) etu = new Personne();
else etu = new Etudiant();
```

# Cast

Le compilateur ne veut pas prendre de risque, mais il tolère que vous en preniez.

## Coercition de type

Forcer le compilateur à considérer une expression comme étant d'un sous-type de son type statique.

Syntaxe java : ((TYPE)expression)

Peut échouer à l'exécution !

# Cast

Le compilateur ne veut pas prendre de risque, mais il tolère que vous en preniez.

## Coercition de type

Forcer le compilateur à considérer une expression comme étant d'un sous-type de son type statique.

Syntaxe java : ((TYPE)expression)

Peut échouer à l'exécution !

## Rappel : Des messages aux méthodes

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant
- ▶ Le code de la méthode est interprété **dans le contexte de l'objet receveur**

## Rappel : Des messages aux méthodes

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant  
Lookup : **il cherche dans sa classe puis dans une super-classe**
- ▶ Le code de la méthode est interprété **dans le contexte de l'objet receveur**

## Rappel : Des messages aux méthodes

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant  
Lookup : il cherche dans sa classe puis dans une super-classe
- ▶ Le code de la méthode est interprété **dans le contexte de l'objet receveur**  
quelle que soit la classe qui l'a défini



## Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe Personne

```
public String toString(){
    String s;
    s= "nom " +this.nom;
    s=s+" prenom "+this.prenom;

    return s;
}
```



# Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe **Etudiant**

```
public String toString(){  
    String s;  
  
    return s;  
}
```

## Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe **Etudiant**

```
public String toString(){
    String s;
    s= "nom " +this.nom;
    s=s+" prenom "+this.prenom;

    return s;
}
```

## Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe **Etudiant**

```
public String toString(){
    String s;
    s= "nom " +this.nom;
    s=s+" prenom "+this.prenom;

    s=s+" formation "+this.formation;
    return s;
}
```

## Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe **Etudiant**

```
public String toString(){
    String s;

    s=super.toString();
    s=s+" formation "+this.formation;
    return s;
}
```

On peut faire appel à l'ancienne méthode avec  
`super.anciennemethode.`

## Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.  
Exemple dans la classe **Etudiant**

```
public String toString(){
    String s;

    s=super.toString();
    s=s+" formation "+this.formation;
    return s;
}
```

et à un super constructeur avec `super`.

## Liaison Tardive

On envoie les messages aux objets au travers de références.

Résolution :

- ▶ L'objet doit posséder une méthode correspondant
- ▶ Lookup : il cherche dans la classe avec laquelle il a été créé puis dans une super-classe
- ▶ La classe de création n'étant connue avec certitude qu'à l'exécution, on parle de liaison tardive ou dynamique

Execution :

- ▶ Le code de la méthode est interprété **dans le contexte de l'objet receveur**
- ▶ Au sein de ce code, un appel au travers de **this** suit le même principe (lookup à partir de la classe de création)
- ▶ Exceptions : les appels au travers de **super** sont résolus statiquement, ainsi que les appels aux constructeurs.

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
type statique Etudiant  
type statique Personne
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
type statique Etudiant  
type statique Personne
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
type statique Etudiant  
type statique Personne
```

```
type dynamique Etudiant  
type dynamique Etudiant
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
type statique Etudiant  
type statique Personne
```

```
type dynamique Etudiant  
type dynamique Etudiant
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
type statique Etudiant  
type statique Personne
```

```
type dynamique Etudiant  
type dynamique Etudiant
```

```
nom prenom formation
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

type statique Etudiant  
type statique Personne

type dynamique Etudiant  
type dynamique Etudiant

nom prenom formation  
nom prenom formation

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud(" AA001");  
ungars.setNumEtud(" AA001");
```

```
type statique Etudiant  
type statique Personne
```

```
type dynamique Etudiant  
type dynamique Etudiant
```

```
nom prenom formation  
nom prenom formation
```

# Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud(" AA001");  
ungars.setNumEtud(" AA001");
```

```
type statique Etudiant  
type statique Personne
```

```
type dynamique Etudiant  
type dynamique Etudiant
```

```
nom prenom formation  
nom prenom formation
```

```
ok  
Erreur de type
```

## Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud(" AA001");  
ungars.setNumEtud(" AA001");  
((Etudiant)ungars).setNumEtud(" AA001");
```

type statique Etudiant  
type statique Personne

type dynamique Etudiant  
type dynamique Etudiant

nom prenom formation  
nom prenom formation

ok  
Erreur de type

## Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud(" AA001");  
ungars.setNumEtud(" AA001");  
((Etudiant)ungars).setNumEtud(" AA001");
```

type statique Etudiant  
type statique Personne

type dynamique Etudiant  
type dynamique Etudiant

nom prenom formation  
nom prenom formation

ok  
Erreur de type  
possible, test à l'exécution

## Compléments

- ▶ La technique de résolution et de recherche indiquée précédemment, est la plus courante, elle varie selon les caractéristiques du langage
  - ▶ en cas d'héritage multiple, on doit déterminer un ordre de parcours des super-classes
  - ▶ en cas de sélection multiple, on détermine la méthode en calculant sur plusieurs classes de receveurs *e.g.* CLOS.
- ▶ La liaison tardive est le seul mode possible en Smalltalk, alors que C++ ou Java essaient de réduire le nombre de cas en proposant des mécanismes qui fixent statiquement la méthode (plus efficace).
- ▶ En Smalltalk, la recherche échoue si on ne trouve pas le sélecteur dans la classe **Object**, dans ce cas, on envoie le message **doesNotUnderstand:** au receveur **self** avec le nom du sélecteur. Cette méthode est redéfinissable, ce qui propose un mécanisme original de traitement d'exceptions.

## Redéfinition et typage

On peut modifier le code d'une méthode héritée.  
Et sa protection ? sa signature ?

## Redéfinition et typage

On peut modifier le code d'une méthode héritée.

Et sa protection ? sa signature ?

Si on veut respecter le typage, il faut respecter les contraintes de substituabilité.

## Redéfinition et typage

On peut modifier le code d'une méthode héritée.

Et sa protection ? sa signature ?

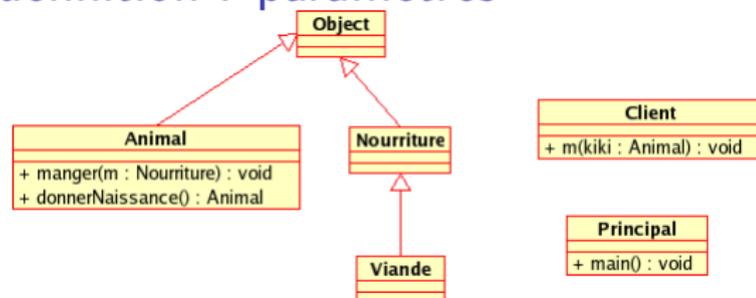
Si on veut respecter le typage, il faut respecter les contraintes de substituabilité.

il faut se souvenir envers qui est pris l'engagement de type.

## Redéfinition de protection

On peut modifier la protection d'une méthode héritée.  
Si on la renforce, les utilisateurs

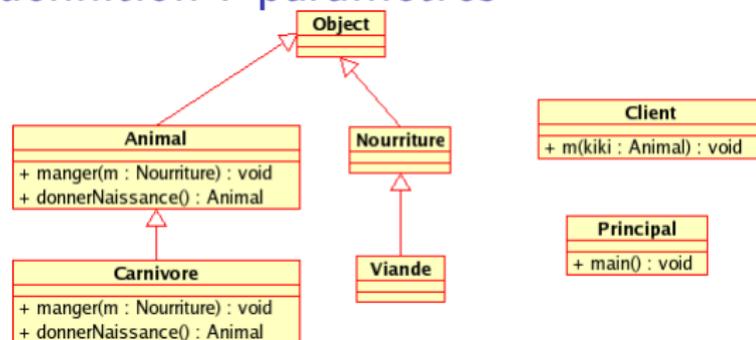
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }
}
```

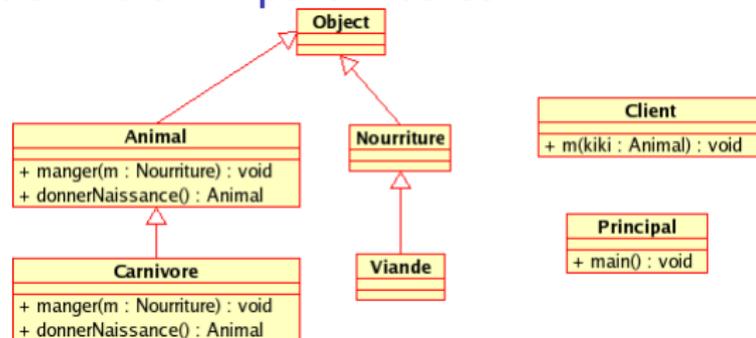
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
```

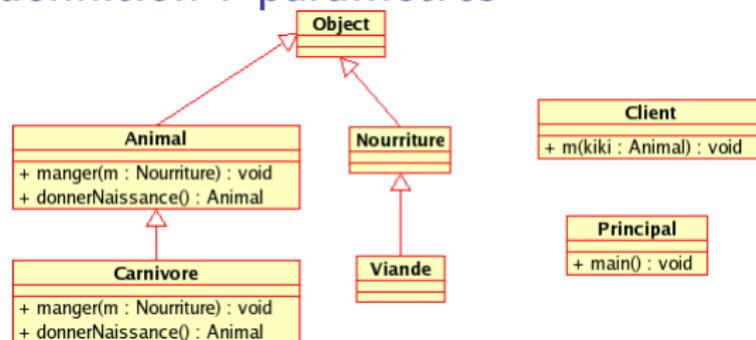
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
```

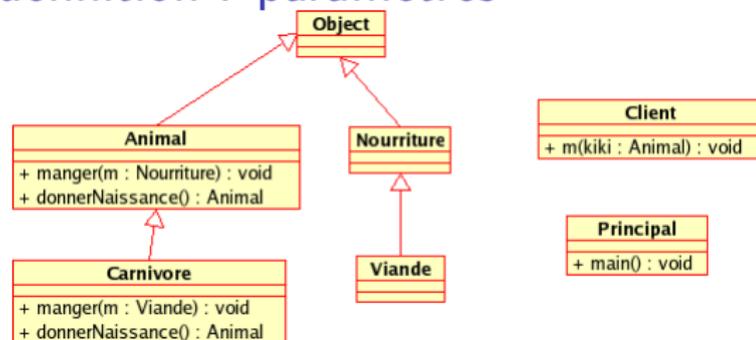
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
```

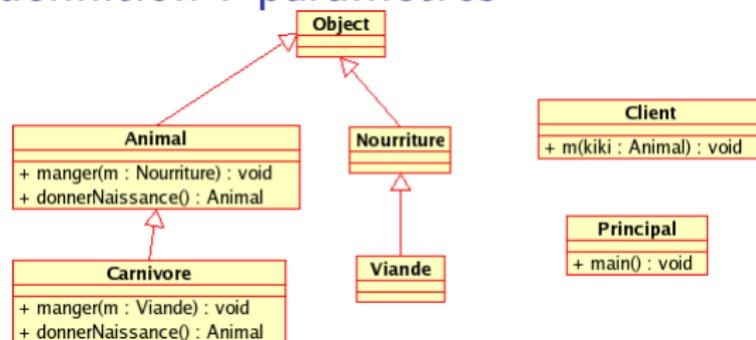
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
```

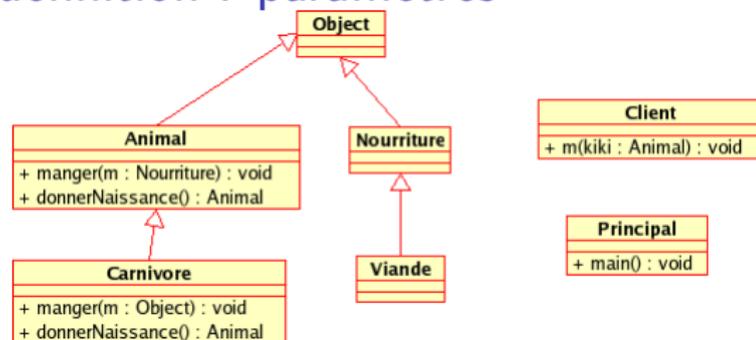
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n); \\Erreur !
```

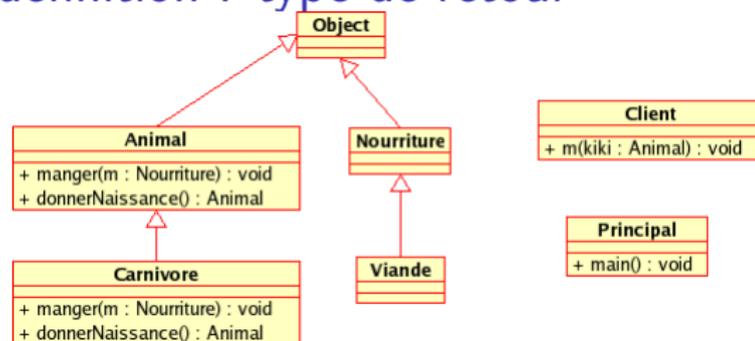
## Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n); \\Ok, mais peu utile
```

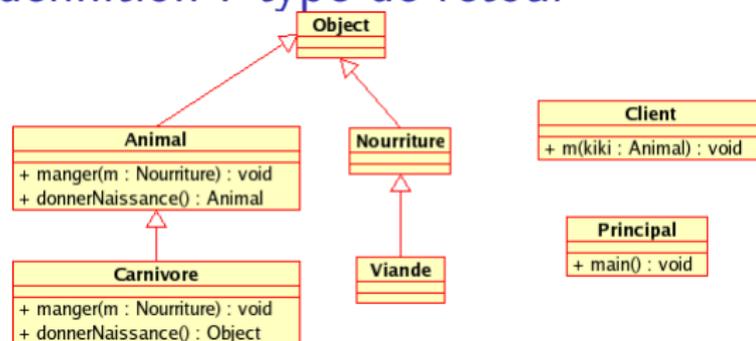
## Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();
```

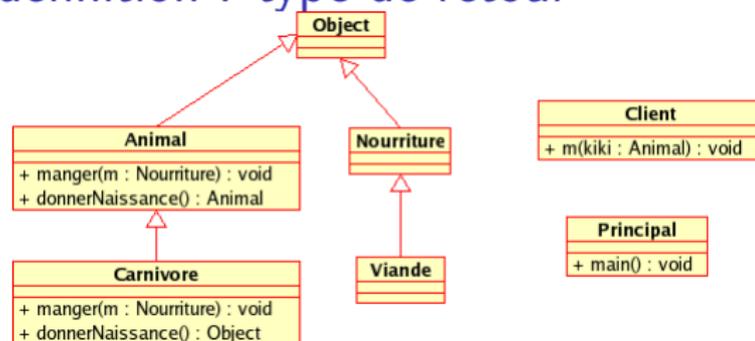
## Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();
```

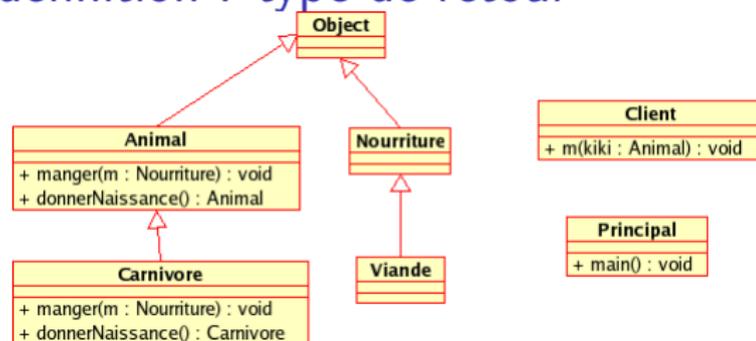
## Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();\\Erreur!
```

## Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance(); \\Ok
```

# Les règles

## Contraintes

Ce qu'on veut faire

Le typage

Java

C++

Eiffel

Smalltalk

## Paramètres

Covariance

Contravariance

Invariance

Invariance

Covariance

Libre<sup>†</sup>

## Retour

Covariance

Covariance

Covariance (depuis 1.5)

Covariance (depuis 3.0)

Covariance

Libre<sup>†</sup>

<sup>†</sup> plus précisément, l'ensemble des méthodes invoquées sur les paramètres et le résultat doivent être applicable, ce qui reste très libre. On autorise ainsi des collections hétérogènes.

# Résumé

- ▶ Typage statique  $\Rightarrow$  Contrôle
- ▶ Sous-typage  $\Rightarrow$  Extension avec de nouveaux objets
- ▶ Héritage  $\Rightarrow$  Réutilisation des attributs et méthodes
- ▶ Redéfinition et liaison dynamique  $\Rightarrow$  adaptation