# Programmation Orientée Objet : Transtypage (cast): upcasting, downcasting

E. Hyon, V. Bouquet[1]

[1]valentin.bouquet@parisnanterre.fr

Licence MIASHS Miage - 2023/2024

# Rappel : Cast (« forçage de type »)

- Le *cast* est le fait de forcer le compilateur à considérer une variable comme étant d'un type qui n'est pas le **type déclaré** ou le **type réel** de la variable.
- En Java, les variables sont soit de type primitif soit des références vers des instances d'une classe.

# Cast : pour les types primitifs

On peut,

1. **Cast explicite** : forcer le type d'une variable de type primitif en un autre type primitif, exemple :

```
// Une constante numerique a virgule est
// consideree comme etant de type double
float z=6.8f;
System.out.println((int)z);
```

2. **Cast implicite** : affecter à une variable d'un type primitif la valeur d'une variable d'un autre type primitif, exemple :

```
int x = 5;
float y = x; // y = 5.0
```

# Cast : pour les types primitifs

## Exemple : appeler une fonction de calcul

```
int calcul(int x) {
  /* faire calcul */
}
```

On souhaite appeler la fonction de calcul mais la variable **val** dont nous disposons est de type float.
Problème : le code suivant ne compile pas :

```
calcul(val);
/* Erreur: la methode calcul n'est pas
applicable pour un argument de type float */
```

Solution : changer le type de la variable dynamiquement

```
calcul((int) val);
```

# Cast : pour les types primitifs

## Exemple : encodage UTF-8

Encodage des caractères UTF-8 entre 0 et 255 (1 octet = 8 bits) :

```
int[] UTF8 = new int[256];
for (int i=0; i < UTF8.length; i++) UTF8[i] = i;
```

On souhaite afficher tous les caractères UTF-8.

Problème : le code suivant affiche les entiers

```
for (int i=0; i < UTF8.length; i++)
  System.out.println(UTF8[i]);
```

Solution : changer le type dynamiquement

```
for (int i=0; i < UTF8.length; i++)
  System.out.println((char) UTF8[i]);
```

# Cast : pour les références

On peut,

1. convertir une référence du type d'une classe donnée en une référence du type d'une autre classe ;

```
Chien c = new Chien(); // Chien herite d'Animal
Animal a = (Animal) c;
```

2. affecter à une référence d'une classe la valeur d'une référence d'une autre classe ;

```
Animal a = new Chien();
```

Notions de **upcasting** et **downcasting**

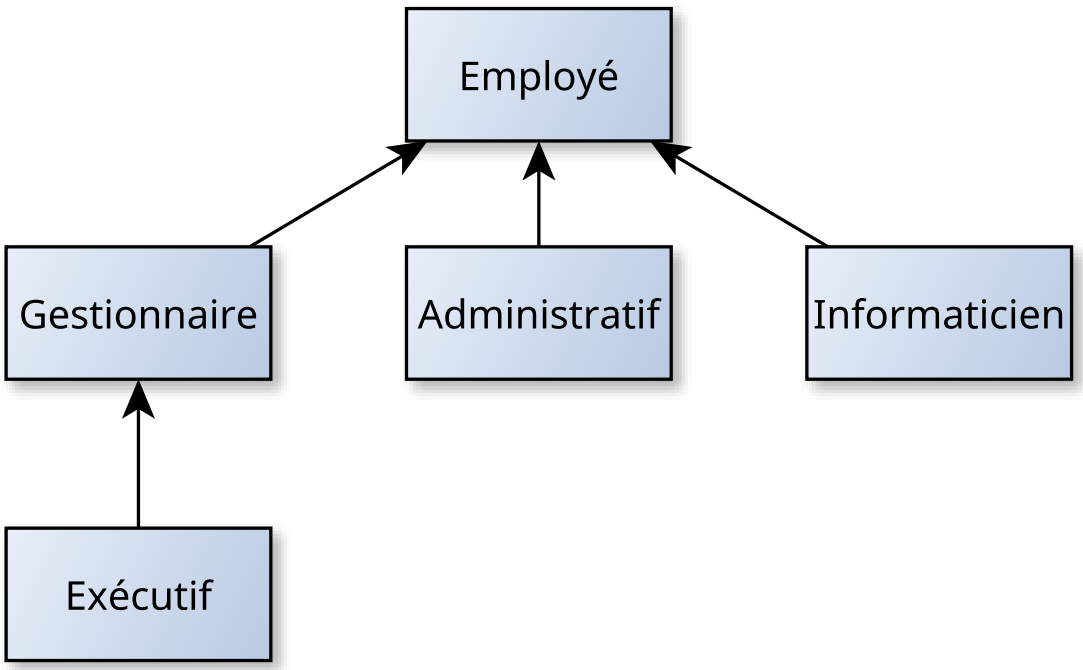
**Attention**

Il doit exister une relation d'héritage entre les deux classes.

# Cast : pour les références

## Exemple : application de gestion des employés

Voici la hiérarchie des sous-classes de la classe Employe :



# Cast : pour les références

## Exemple : casts implicites

Initialisation d'un tableau d'employés contenant des gestionnaires, informaticiens, administratifs, et des exécutifs :

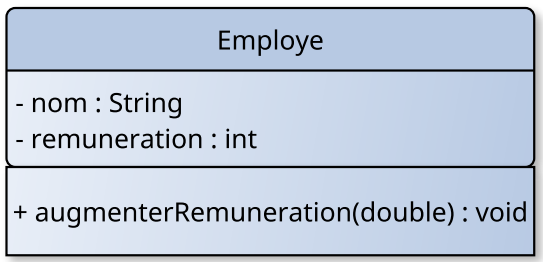
```
Employe[] listeEmployes = new Employe[5];

listeEmployes[0] = new Gestionnaire("Jean-Yves");
listeEmployes[1] = new Informaticien("Pierrette");
listeEmployes[2] = new Administratif("Yvette");
listeEmployes[3] = new Informaticien("Maurice");
listeEmployes[4] = new Executif("Emmanuel");
```

# Cast : pour les références

## Exemple : vérifier le type avec instanceof

Objectif 1 : augmenter la rémunération des informaticiens (et seulement des informaticiens)

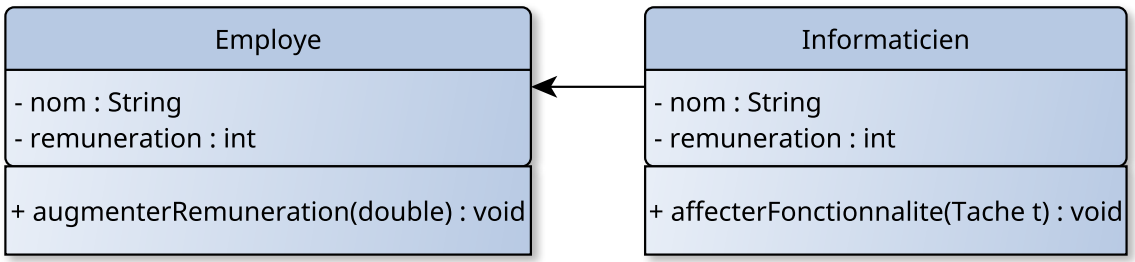



```
for (int i = 0; i < listeEmploye.length; i++) {
   Employe e = listeEmploye[i];
   if (e instanceof Informaticien) {
        e.augmenterRemuneration(3.5);
   }
}
```

# Cast : pour les références

## Exemple : accéder à une méthode d'une sous-classe (1)

Objectif 2 : affecter une nouvelle fonctionnalité pour améliorer le site web à l'informaticien Maurice



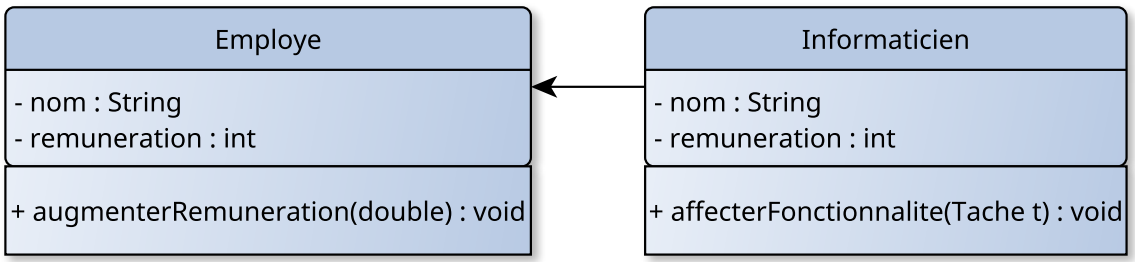

Problème : le code suivant ne compile pas

```
for (int i = 0; i < listeEmploye.length; i++) {
  Employe e = listeEmploye[i];
  if (e instanceof Informaticien) {
    if (e.getNom() == "Maurice")
      e.affecterFonctionnalite(tache);
      // methode non definie dans Employe !
  }
}
```

# Cast : pour les références

## Exemple : accéder à une méthode d'une sous-classe (2)

Objectif 2 : affecter une nouvelle fonctionnalité pour améliorer le site web à l'informaticien Maurice




Solution : changer le type à l'exécution

```
for (int i = 0; i < listeEmploye.length; i++) {
  Employe e = listeEmploye[i];
  if (e instanceof Informaticien &&
      e.getNom() == "Maurice") {
    Informaticien maurice = (Informaticien) e;
    maurice.affecterFonctionnalite(tache);
  }
}
```

# Upcasting : classe fille → classe mère

Pour une référence déclarée du type d'une classe $A$ :
affecter une référence vers un objet d'une autre classe $B$
est possible uniquement si $B$ est l'une des sous-classes de $A$.

```
public class A { ... }
public class B extends A { ... }

A ref = new B() // CORRECT
```

On dit que *ref* est une référence « surclassée » (elle est du type `A` et contient l'adresse d'une instance d'une sous-classe de `A`).

Moyen mnémotechnique *upcast* = surclassement = la référence est plus haute que l'objet réel.

# Upcasting : Méthodes invoquées

**Règle**

On ne peut invoquer que des méthodes (visibles) définies dans la classe de la référence.

```
public class A { public void f() {...}  }
public class B extends A { public void g() }

A ref = new B()
ref.f() // Compile et s'execute correctement
ref.g() // Erreur a la compilation :
```

Erreur compilateur :

```
la methode g n'est pas definie pour le type A
```

# Upcasting : attributs et méthodes visibles

**Règle**

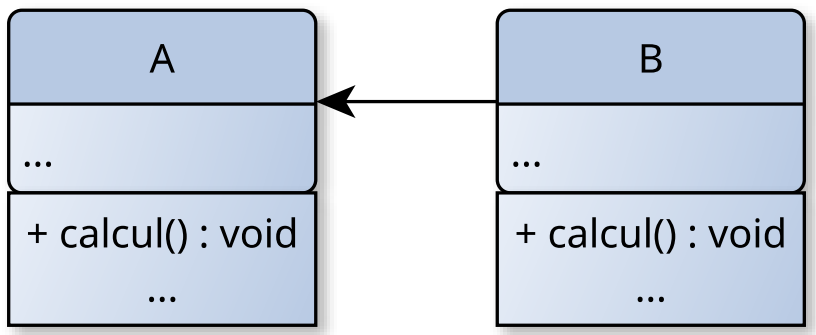
En effet, quand une variable *ref* est une référence de classe *A* alors ne sont disponibles à l'utilisation :

- les attributs (visibles) de *A*.
- les méthodes (visibles) de *A*.

Cette règle est vérifiée à la compilation.

# Upcasting : redéfinition de méthode (1)

Si la sous-classe `B` redéfinit une méthode héritée, quelle est la méthode invoquée ?




```
public class A { public void calcul() {...}  }
public class B extends A { public void calcul() }

A ref = new B();
ref.calcul();
```

# Upcasting : redéfinition de méthode (2)

**Règle**

Le choix de la méthode à invoquer se fait au moment de l'exécution et non lors de la compilation (liaison dynamique).

C'est **toujours** la méthode du type de l'objet instancié (l'objet en mémoire) qui est exécutée si celle-ci a été redéfinie.

Le code suivant exécute la méthode *calcul* redéfinit dans la classe B :

```
A ref = new B();
ref.calcul();
```

# Upcasting : redéfinition de méthode (3)

```
class A {
    public void f () {
     System.out.println("A: f");
    }
}
class B extends A {
   public void f () { System.out.println("B: f");}
   public void g()  {  System.out.println("B: g");}
}

A ref = new B();
ref.f(); // affiche B: f
ref.g(); // erreur a la compilation
```

# Upcasting : redéfinition de méthode (4)

```
class A {
    public void f () {  System.out.println("A: f");}
    public void g()  {  System.out.println("A: g");}
}
class B extends A {
  public void f () { System.out.println("B: f");}
}

A ref1 = new B();
ref1.f(); // affiche B: f
ref1.g(); // affiche A: g

A ref2 = new A();
ref2.f(); // affiche A: f
ref2.g(); // affiche A: g
```

# Upcasting : redéfinition de méthode (5)

```
class A {
    public void f () { System.out.println("A: f"); }
}
class B extends A {
  public void f () { System.out.println("B: f");}
  public void g()  {  System.out.println("B: g");}
}

A ref1 = new B();
ref1.f(); // affiche B: f
ref1.g(); // erreur a la compilation du programme

B ref2 = (B) ref1;
ref2.g(); // affiche B: g
```

# Downcasting : classe mère → classe fille

On peut convertir une référence « surclassée » pour « libérer » certaines fonctionnalités cachées par le surclassement.

```
A ref = new B(); // surclassement, upcasting
B ref2 = (B) ref; // declassement, downcasting
```

Pour que la « conversion » fonctionne, il faut qu'à l'exécution le type réel de la référence à convertir soit une des sous-classe de $B$ !

# Downcasting : utilisation de instanceof

Pour savoir si une référence réfère à une **instance** d'une des sous-classes d'une classe donnée, on utilise le mot-clé **instanceof**.

```
if (ref instanceof Classe) { /* ... */ }
```

- si `ref` réfère à une instance de type *Classe*, alors l'expression renvoie **true**
- si `ref` réfère à une instance qui est une sous-classe de *Classe*, alors l'expression renvoie **true**
- autrement, l'instruction renvoie **false**

Exemple

```
A ref = new A();
if (ref instanceof A)
{
        System.out.println("ref est de type A");
}
```

# Downcasting : exemples de instanceof (1)

## Exemple

La classe `C` étend la classe `B` qui étend la classe `A`.

```
A a; A a2; C c; B b;
a=new A(); b=new B(); c=new C(); a2=new B()

a instanceof A ; // TRUE
a instanceof B ; // FALSE

b instanceof A ; // TRUE
b instanceof B ; // TRUE
b instanceof C ; // FALSE

c instanceof A ; // TRUE
c instanceof B ; // TRUE
c instanceof C ; // TRUE

a2 instanceof B ; // TRUE
```

# Downcasting : exemples de instanceof (2)

## L'exemple précédent avec une légère modification

### Exemple

La classe `C` étend la classe `B` qui étend la classe `A`.

Références déclarées de type $A$. Instances réelles types différents

```
A a1; A a2; A a3;
a1=new A(); a2=new B(); a3=new C();

a1 instanceof A ; // TRUE
a1 instanceof B ; // FALSE

a2 instanceof A ; // TRUE
a2 instanceof B ; // TRUE
a2 instanceof C ; // FALSE

a3 instanceof A ; // TRUE
a3 instanceof B ; // TRUE
a3 instanceof C ; // TRUE
```

# Downcasting : mise en garde !

**Attention**

Le downcasting ne permet pas de convertir une instance d'une classe donnée en une instance d'une sous-classe !

```
class A { ... }
class B extends A { ... }

A a = new A();
B b = (B) a; // Erreur a l'execution
```

```
Exception in thread "main"
java.lang.ClassCastException:
A cannot be cast to B
```

# Downcasting : mise en garde (suite)

## Attention

Le déclassement (downcasting) ne permet pas de convertir une instance d'une classe donnée en une instance d'une sous-classe.

car une classe mère n'est pas une classe fille.

Imaginons une classe Animal et une classe Chien qui hérite de Animal. Le déclassement (downcasting) ne fonctionne pas.

## Attention

Un chien est un animal mais tous les animaux ne sont pas des chiens.

# Downcasting : récupération des fonctionnalités

```
class A {
   public void f () { System.out.println("A: f"); }
}
class B extends A {
  public void f () { System.out.println("B: f");}
  public void g()  {  System.out.println("B: g");}
}

A ref = new B();
ref.f(); // affiche B: f
B b = (B) ref;
b.g(); // affiche B: g
```

# Autoboxing

## Définition

L'autoboxing désigne le mécanisme qui, dans certaines situations, transforme automatiquement une variable de type primitif en une instance de la classe enveloppe associée.

## Exemple

```
Integer i = 6; // Integer i = new Integer(6);
Character c = 'c'; // Character c = new Character('c');
```

# Unboxing

## Définition

L'unboxing désigne le mécanisme qui, dans certaines situations, transforme une instance d'une des classes enveloppes des types primitifs en une variable du type primitif associé. C'est le mécanisme inverse de l'autoboxing.

## Exemple

```
Integer i = new Integer(-42);
int i = valeurAbsolue(i);
/* ... */
public static int valeurAbsolue(int i) { /* ... */}
```