

POO-IG

# Programmation Orientée Objet et Interfaces Graphiques

---

Cristina Sirangelo  
IRIF, Université Paris Cité  
[cristina@irif.fr](mailto:cristina@irif.fr)

Exemples et matériel empruntés :

- \* Core Java - C.Horstmann - Prentice Hall Ed.
- \* POO in Java - L.Nigro & C.Nigro - Pitagora Ed.

# Héritage

---

# Héritage et réutilisation de code

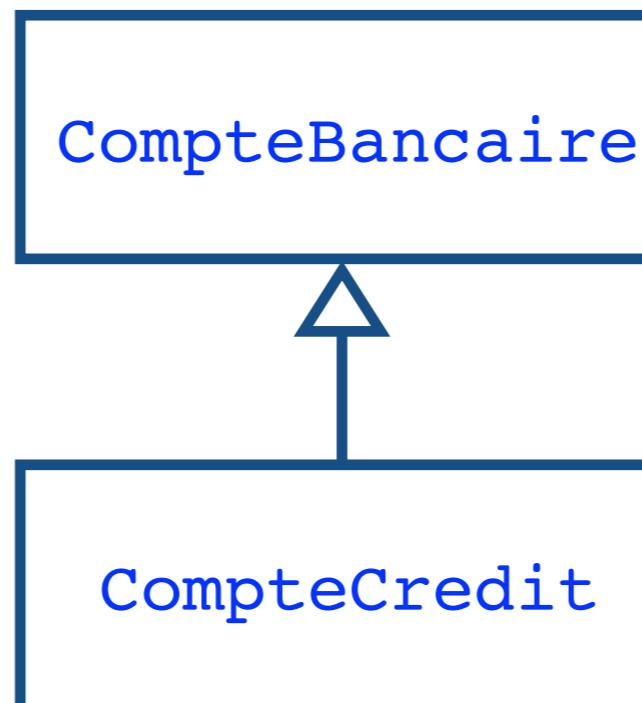
## Héritage : un autre mécanisme de réutilisation de code

- Certaines classes sont par nature des **spécialisation** d'autres classes :
  - ont les mêmes propriétés et méthodes + d'autres
- Exemple. Supposons :
  - d'avoir une classe CompteBancaire
  - de vouloir concevoir une nouvelle classe CompteCredit
- Un compte avec crédit **est un** compte bancaire
  - => la classe CompteCredit aura toutes les propriétés et méthodes de CompteBancaire plus d'autres spécifiques
- L'héritage permet de décrire la classe CompteCredit par différence :
  - **décrire uniquement les caractéristiques en plus ou différentes par rapport à l'autre classe**

# Héritage : syntaxe et notation

```
public class CompteBancaire { ... }
public class CompteCredit extends CompteBancaire { ...
    // champs et méthodes additionnelles ou différentes
}
```

- CompteCredit : **sous-classe** (ou **classe enfant**)
- CompteBancaire : **super-classe** (ou **classe parent**)
- Tous les membres de la classe parent sont automatiquement membres de la classe enfant ( **hérités**)
- On dit qu'il existe une relation **is-a** (est-un) entre CompteCredit et CompteBancaire



# Exemple de classe parent

```
package poo.banque;

public class CompteBancaire {
    private String numero;
    protected double solde = 0;
    public CompteBancaire(){numero ="";}
    public CompteBancaire( String numero ){
        //pre-condition: numero != null
        this.numero = numero;
    }
    public CompteBancaire( String numero, double solde ){
        //pre: numero != null && solde >= 0
        this.numero = numero; this.solde = solde;
    }
    public void verser( double montant ){
        if( montant<=0 ) throw new IllegalArgumentException
            ("montant non valide.");
        solde = solde + montant;
    }
//continue...
```

# Exemple de classe parent

```
// Suite : Class CompteBancaire
public boolean retirer( double montant ){
    if( montant <= 0 )
        throw new IllegalArgumentException("montant non
                                            valide.");
    if( montant > solde ) return false;
    solde -= montant;
    return true;
}
public double getSolde(){
    return solde;
}
public String getNumero(){
    return numero;
}
}//CompteBancaire
```

# Conception d'une sous-classe

```
package poo.banque;  
public class CompteCredit extends CompteBancaire {...}
```

- La classe CompteCredit hérite de CompteBancaire tous les champs et méthodes (pas les constructeurs)
- Donc par simple extends un compte avec credit a un solde un numéro et toutes les méthodes pour les manipuler
- En plus un compte avec credit doit permettre un découvert, jusqu'à une limite (crédit)

```
public class CompteCredit extends CompteBancaire {  
    private double credit = 1000;  
    //constructeurs et nouvelles méthodes  
    ...  
}
```

# Objets de sous-classe

Un objet de classe **CompteBancaire**

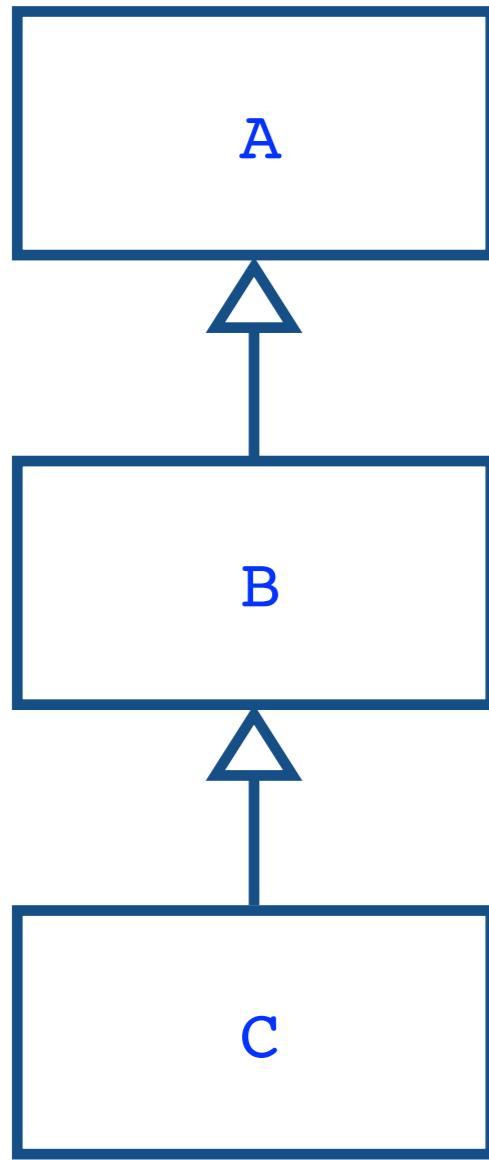
"3F56J790"	.numero
4000	.solde

Un objet de classe **CompteCredit**

"5T90H251"	.numero
30000	.solde
1000	.credit

# Hiérarchie de classes

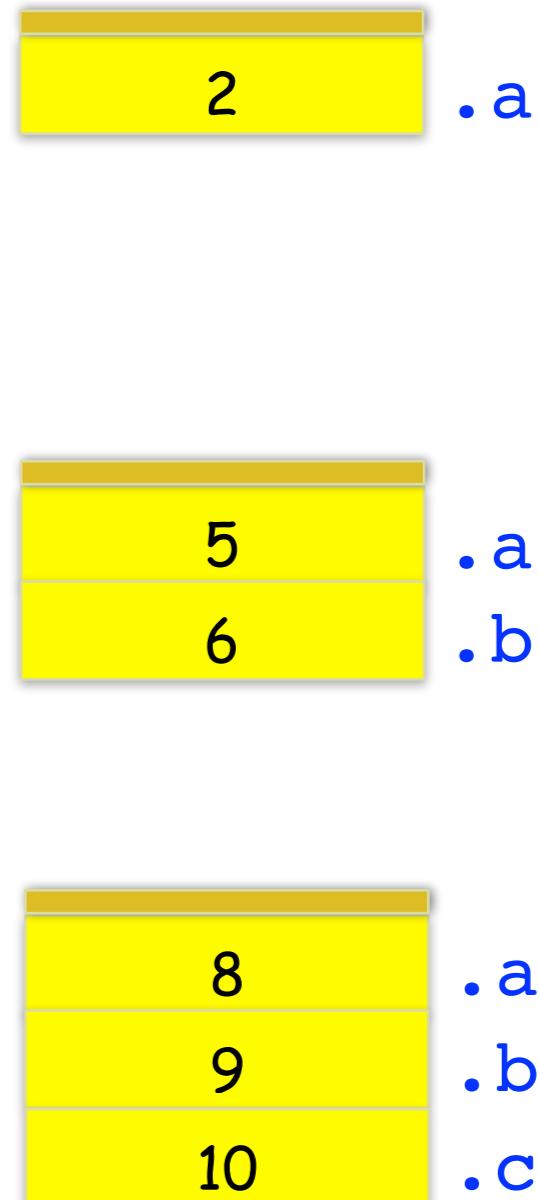
Diagramme



Définition de la classe

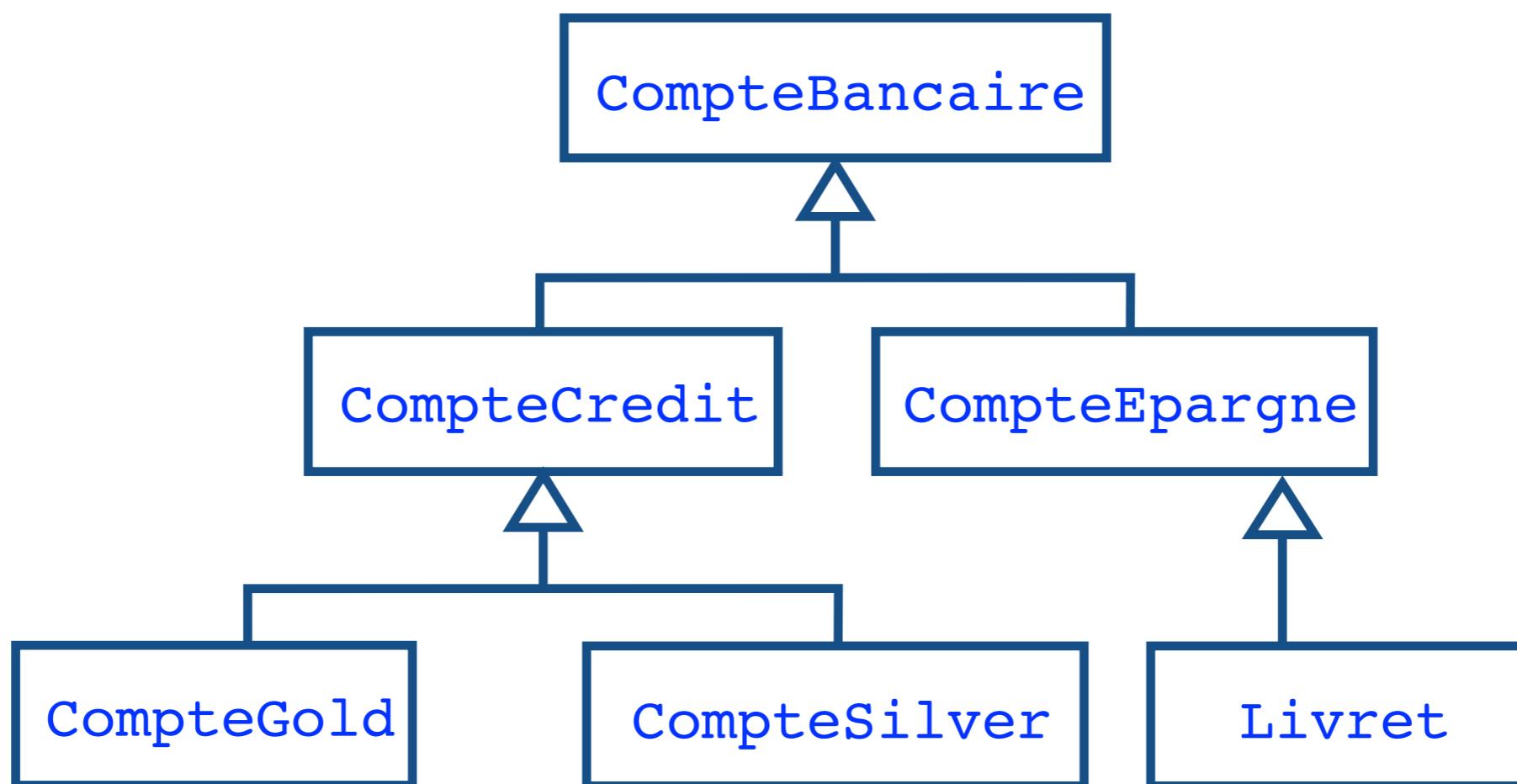
```
class A {  
    int a;  
    ...  
}  
  
class B extends A {  
    int b;  
    ...  
}  
  
class C extends B {  
    int c;  
    ...  
}
```

Exemple d'objet  
de la classe



# Hiérarchie de classes

- En Java une classe peut avoir plusieurs sous-classes
- Mais une classe peut avoir une seule classe parent
- => La hiérarchie est en général arborescente



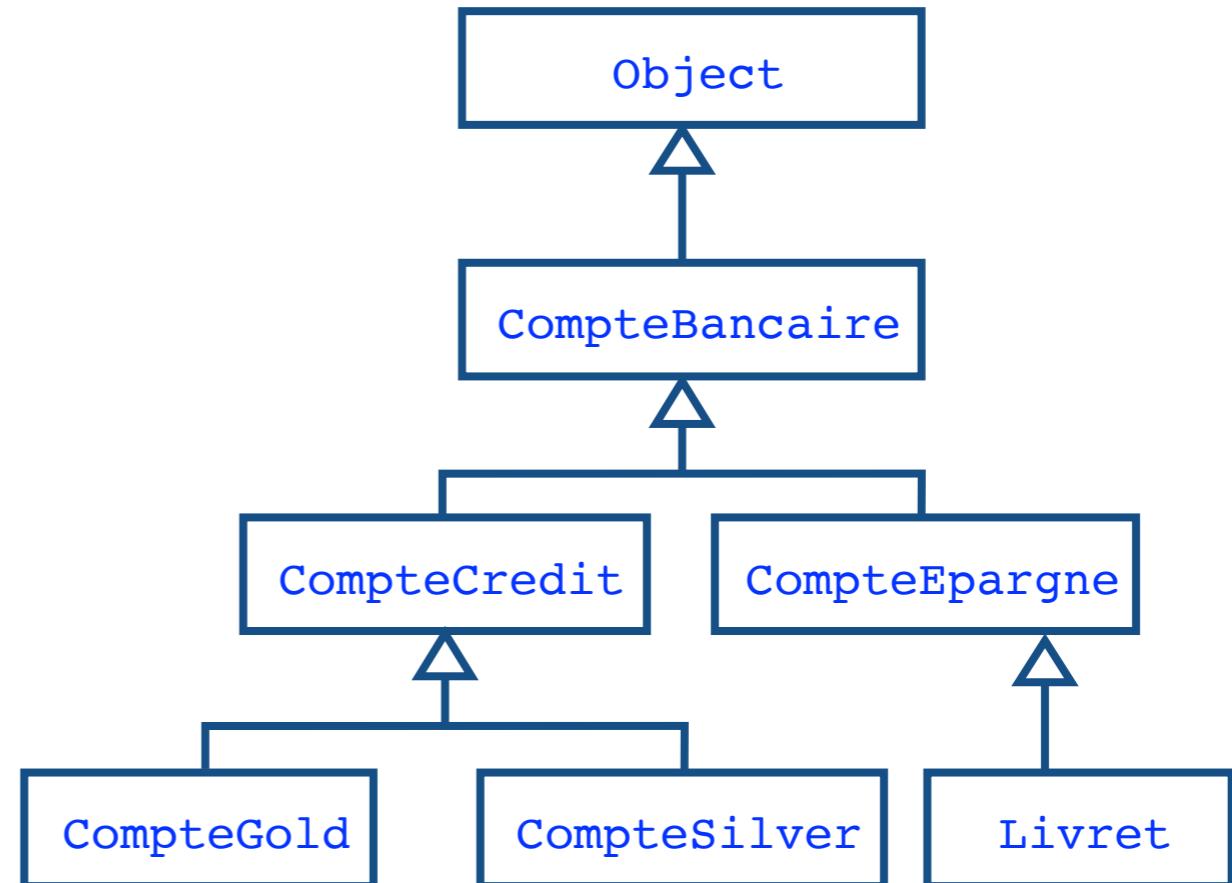
# Hiérarchie de classes

- La classe Object (dans `java.lang`) est la racine de cette hiérarchie
  - une classe qui n'étend aucune autre classe, étend implicitement Object

```
class CompteBancaire  
{...}
```

équivalent à

```
class CompteBancaire  
    extends Object  
{...}
```



- => Toute classe étend directement ou indirectement Object

# Méthodes additionnelles

- Les méthodes de la classe parent sont héritées par la classe enfant, qui peut en plus définir des méthodes additionnelles

```
public class CompteCredit extends CompteBancaire {  
    private double credit = 1000;  
    //constructeurs  
    ...  
    public double getCredit() { return credit; }  
    public void nouveauCredit( double credit ){  
        if( credit < 0 || solde + credit < 0)  
            throw new IllegalArgumentException  
                ("Nouveau credit non valide.");  
        this.credit = credit;  
    }  
    //autres méthodes  
    ...  
}
```

**Remarque :** les méthodes de la classe enfant peuvent manipuler à la fois les champs hérités (visibles) et les champs propres

# Méthodes additionnelles

- Les méthodes additionnelles ne sont évidemment pas accessibles par un objet de la classe parent

```
CompteBancaire cb = new CompteBancaire ("ASD634");  
CompteCredit cc = new CompteCredit ("SFD78634G");
```

```
cb.getCredit(); //ERREUR  
cc.getCredit(); //OK méthode propre
```

- En revanche les méthodes (visibles) de la classe parent sont également des méthodes de la classe enfant

```
cc.getSolde(); // OK méthode héritée
```

# Constructeurs de sous-classes

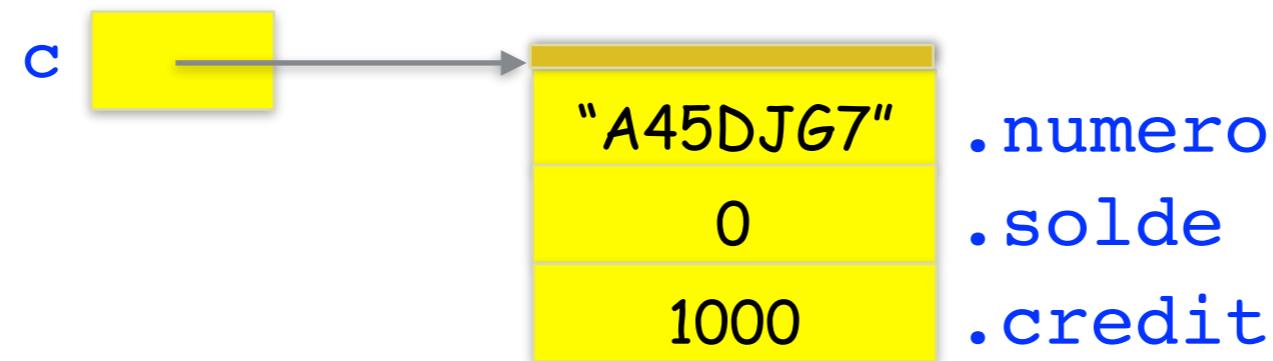
- Initialisation d'un objet de sous-classe
  - à la fois les champs hérités et les champs propres doivent être initialisés
- Pour initialiser les champs hérités il est nécessaire d'**invoyer le constructeur de la classe parent** : `super (...)`
- Si présent, `super(...)` doit être la première instruction du constructeur
- `super()` et `this()` ne peuvent pas être invoqués dans un même constructeur
- Si aucun des deux n'est invoqué, un appel `super()` (sans arguments) est implicite au début du constructeur
  - Dans ce cas : erreur si la classe parent n'a pas de constructeur sans arguments

# Constructeurs de sous-classes et super( )

```
public class CompteCredit extends CompteBancaire {  
    private double credit = 1000;  
    public CompteCredit  
        (String numero, double solde, double credit){  
        super( numero, solde );  
        if( credit < 0 || credit + solde < 0)  
            throw new IllegalArgumentException  
                ("credit ou solde non valide.");  
        this.credit = credit;  
    }  
    public CompteCredit () {} //super() implicite  
    public CompteCredit (String numero){  
        super( numero );  
    }  
    public CompteCredit (String numero, double solde){  
        super (numero, solde);  
    }  
    //methodes  
    ...
```

# Créer des objets de sous-classe

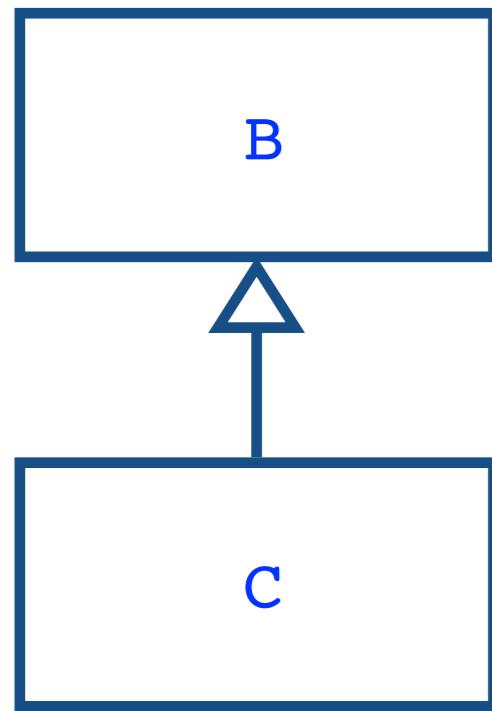
```
CompteCredit c = new CompteCredit ("A45DJG7");
```



# Redéfinition de champs

- Un champ avec le même nom d'un champ de la super-classe **cache** ce dernier

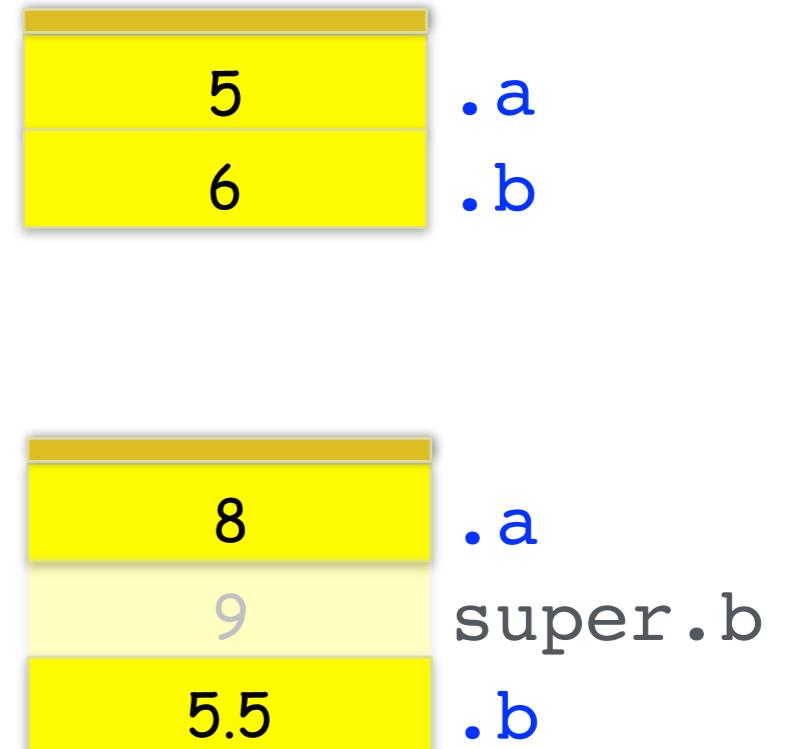
Diagramme



Définition de la classe

```
class B {  
    int a;  
    int b;  
    ...  
}  
  
class C extends B {  
    double b;  
    ...  
}
```

Exemple d'objet  
de la classe



# Redéfinition de champs et super.

```
class B {  
    int a;  
    int b;  
  
    ...  
}  
  
class C extends B {  
    double b;  
    void f() {  
        b = 5.5; // champ double de sous-classe  
        super.b = 9; // champs int de la super-classe  
    }  
  
    ...  
}  
  
B ob = new B();  
C oc = new C();  
ob.b = 9; // champs int de la super-classe  
oc.b = 5.5; // champ double de la sous-classe
```

# Overriding (Redéfinition de méthodes)

- Il est possible de redéfinir aussi des méthodes de la classe parent
- redéfinition : en anglais overriding
- pour qu'il y ait redéfinition, la méthode redéfinie dans la sous-classe doit avoir la même signature (nom de méthode et types des paramètres) que la méthode de la classe parent à redéfinir

```
public class CompteCredit extends CompteBancaire {  
    private double credit = 1000;  
    //constructeurs, getCredit et nouveauCredit  
    ...  
    public boolean retirer( double montant ){  
        //pre: montant > 0;  
        if( montant <= solde + credit ) {  
            solde -= montant;  
            return true;  
        }  
        return false;  
    }  
}
```

# Overriding et super.

- Une méthode redéfinie cache la méthode de la classe parent

```
CompteBancaire cb = new CompteBancaire ("A346",1000);
CompteCredit cc = new CompteCredit ("GA38",1000, 200);

cb.retirer (1200); // retirer de CompteBancaire, retrait refusé
cc.retirer (1200); //retirer de CompteCredit, retrait effectué
```

- Si une méthode (visible) de la classe parent a été redéfinie, elle est encore accessible dans la classe enfant : notation **super**.  
(e.g. **super.retirer (montant)** )
- Remarque : pas utile dans notre cas (**super.retirer** échoue si le solde devient négatif)

# Redéfinition : remarque

- Pour les variables : c'est le nom de la variable qui est pris en compte (pas le type).
  - dans une classe, à chaque nom de variable ne correspond qu'une seule déclaration.
  - même nom dans une sous-classe => redéfinition
- Pour les méthodes : c'est la signature (nom + type des paramètres) qui est prise en compte
  - dans une classe, à une signature correspond une seule définition
  - même signature dans une sous-classe => redéfinition
  - remarque : on peut avoir des méthodes de même nom et de signatures différentes (surcharge)

# Overriding et overloading

- Redéfinition : si le nom d'une méthode est le même, mais pas la signature, il n'y a pas overriding, mais **overloading** (surcharge)

```
public class CompteCredit extends CompteBancaire {  
    ...  
    public boolean retirer( double montant ) {...}  
  
    public boolean retirer( double montant, String libellé)  
    {...}  
}
```

- **retirer** est une méthode additionnelle
- ne cache pas la méthode **retirer** de la classe parent

```
CompteCredit cc = new CompteCredit ("GA38",1000, 200);  
  
cc.retirer (1000); // OK, mais méthode de la classe parent  
cc.retirer (1200, "loyer"); //OK
```

# Overriding : type de retour

- Le type de retour d'une fonction n'est pas partie de la signature  
=> peut changer dans une redéfinition,
- restriction : doit être un sous-type de l'original

```
public class CompteBancaire {  
    public CompteBancaire copie () {  
        return new CompteBancaire (numero, solde);  
    }  
}  
public class CompteCredit extends CompteBancaire {  
    public CompteCredit copie (){  
        return new CompteCredit (getNumero(), solde, credit);  
    } //overriding  
}
```

# Overriding : type de retour

- Le type de retour d'une fonction n'est pas partie de la signature  
=> peut changer dans une redefinition,
- restriction : doit être un sous-type de l'original

```
public class CompteBancaire {  
    public CompteBancaire copie () {  
        return new CompteBancaire (numero, solde);  
    }  
}  
public class CompteCredit extends CompteBancaire {  
    public Object copie () {...} // ERREUR à la compilation  
}
```

# Overriding : modificateur d'accès

- Le modificateur d'accès peut changer dans une redéfinition
- restriction : uniquement pour élargir l'accès

```
public class A {  
    protected double f() {...}  
}  
public class B extends A {  
    public double f() {...} //overriding  
}  
  
private < (package) < protected < public
```

# Overriding : modificateur d'accès

- Le modificateur d'accès peut changer dans une redefinition
- restriction : uniquement pour élargir l'accès

```
public class A {  
    protected double f() {...}  
}  
public class B extends A {  
    private double f() {...} // ERREUR à la compilation  
}
```

private < package < protected < public

# Overriding : annotations

- Optionnellement, une annotation peut indiquer au compilateur qu'une méthode est une redéfinition

```
public class CompteCredit extends CompteBancaire {  
    ...  
    @Override  
    public boolean retirer( double montant ) {  
        ...  
    }  
}
```

- Le compilateur pourra détecter des éventuels erreur (e.g. changement de la signature)

# Interdire l'overriding

- Le modificateur **final** interdit la redéfinition pour une méthode
- Remarque : en revanche une variable avec modificateur **final** (i.e. une constante) peut être occultée

```
public class A {  
    public final int i = 3;  
    public final void f() {...}  
}
```

```
public class B extends A {  
    private final int i = 2; //occulte super.i  
    public void f() {...} // erreur à la compilation  
}
```

# Polymorphisme

- Un objet d'une sous-classe est également un objet de sa super-classe
  - il en possède tous les champs et méthodes
- exemple : un CompteCredit est également un CompteBancaire

```
public class CompteBancaire {...}  
public class CompteCredit extends CompteBancaire{...}
```

- Donc un objet d'une sous-classe peut être affecté à une variable d'une super-classe

```
CompteCredit cc = new CompteCredit("AG675", 0, 200);  
CompteBancaire cb = new CompteBancaire("234BD", 0);  
cb = cc; //et pas viceversa
```

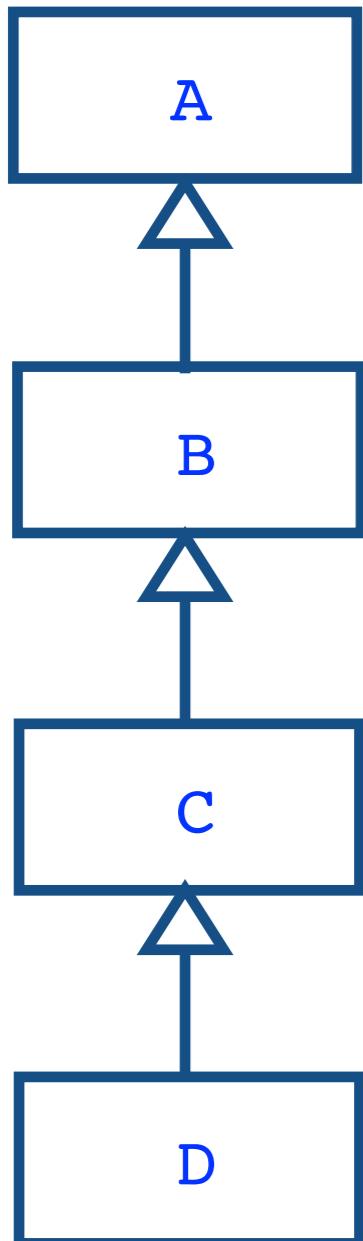
- Phénomène appelé polymorphisme
  - une même variable peut référencer des objets de plusieurs types différents pendant sa vie

# Type déclaré et type effectif

- Consequence du polymorphisme
- Un objet a un type déclaré et un type effectif
  - **type déclaré** : type de la déclaration , eg,  
`CompteBancaire cb;`
    - **CompteBancaire est le type déclaré de cb**
    - associé au nom de la variable en phase de compilation,  
ne change jamais
  - **type effectif** : classe avec laquelle l'objet référencé a été construit  
`CompteCredit cc = new CompteCredit("AG675", 0, 200);  
cb = cc;`
    - **CompteCredit est le type effectif de cb**
    - peut changer dynamiquement (i.e. pendant l'exécution)
    - peut être testé dynamiquement (cf. slides suivants)

# getClass

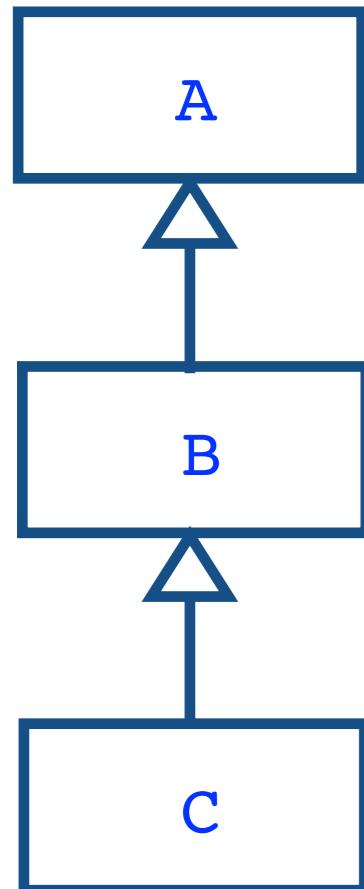
- getClass retourne **le type effectif** d'un objet



```
B c = new C();  
// c a type effectif C  
c.getClass() == C.class; //true  
c.getClass() == B.class; //false  
c.getClass() == A.class; //false  
c.getClass() == D.class; //false
```

# Polymorphisme : règle générale

- Un objet de type effectif *C* est considéré également de tous les types de niveau plus haut que *C* dans la hiérarchie d'héritage



```
C c = new C();  
// c a type déclaré et effectif C  
// c est également de type B et A
```

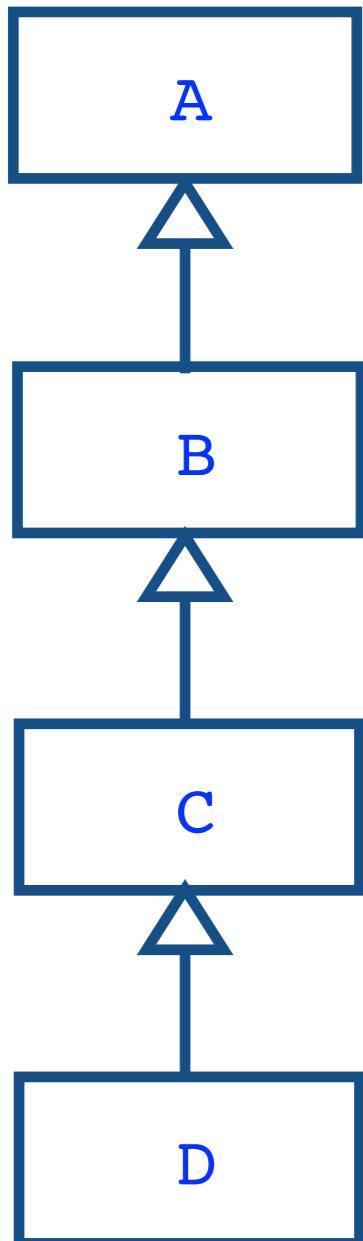
- => Un objet de type effectif *C* peut être référencé par une variable de type déclaré *D* pour tout type *D* de niveau supérieur ou égal à *C* dans la hiérarchie d'héritage

```
B b = new C(); //OK  
//b a type déclaré B et type effectif C  
A a = new C(); //OK  
//a a type déclaré A et type effectif C  
C c = new A(); //ERREUR
```

# instanceof

**obj instanceof A**

- retourne true si obj est de type A (c-à-d A est un ancêtre du type effectif de obj)



```
B c = new C();  
// c a type effectif C  
c instanceof C; //true  
c instanceof B; //true  
c instanceof A; //true  
c instanceof D; //false
```

# Polymorphisme et type déclaré

- Java est un **langage typé**
  - le compilateur doit vérifier la légalité des appels des méthodes et des accès aux champs par rapport au type déclaré:
    - `a.f()` est légal si pour le type déclaré de la variable `a` il existe une méthode `f()` qui peut s'appliquer à un objet de ce type
    - `a.m` est légal si pour le type déclaré de la variable `a` il existe une variable `m` qui peut s'appliquer à un objet de ce type
  - Remarque : **le compilateur ne connaît que le type déclaré des variables**, la vérification ci-dessus ne peut que concerner le type déclaré !

# Polymorphisme et type déclaré

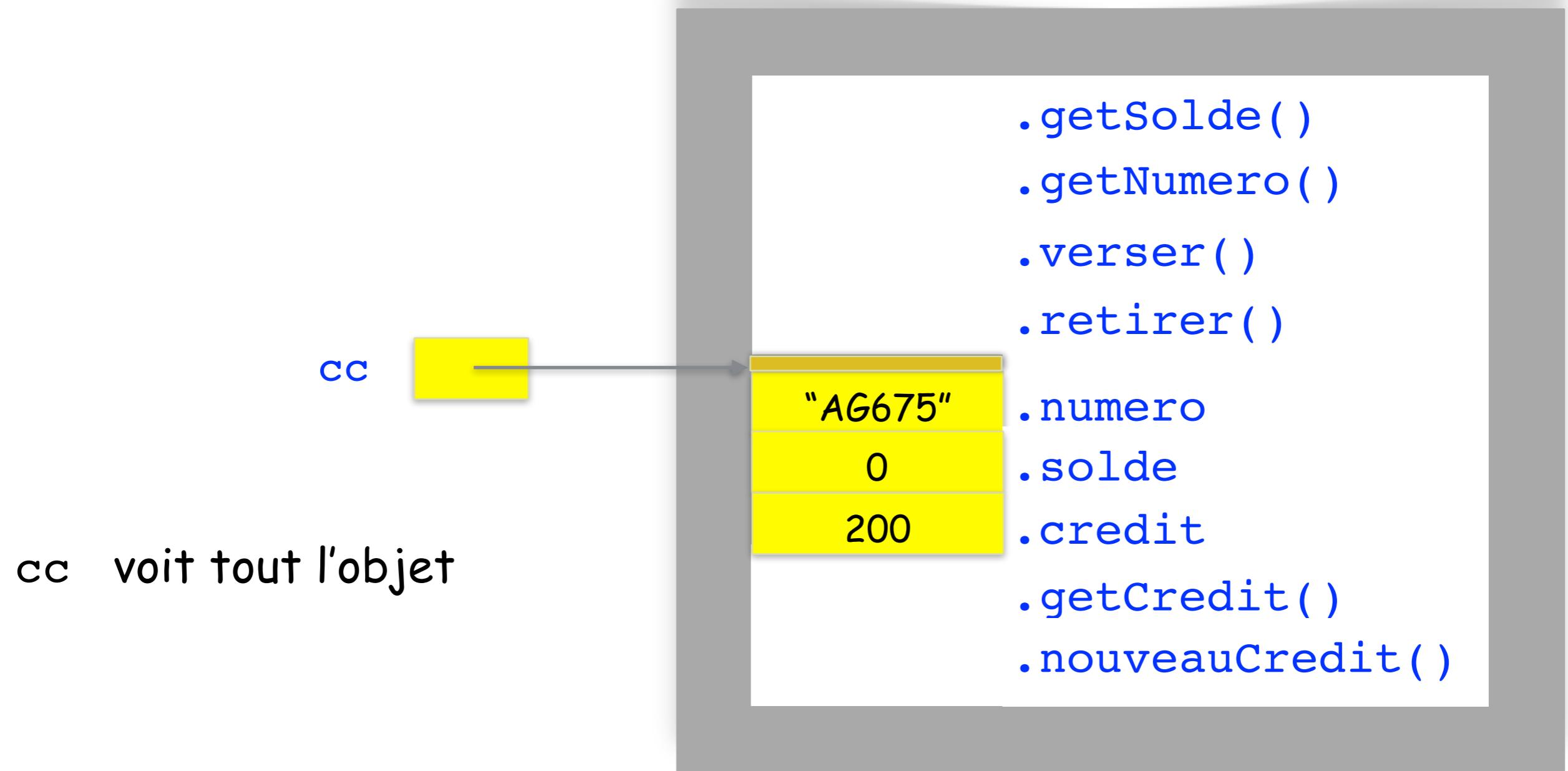
- => le type déclaré dit ce qu'on peut faire avec l'objet
- Seulement les champs et méthodes (visibles) de la classe C sont accessibles sur un objet de type déclaré C
  - même si le type effectif est en réalité plus spécifique

```
CompteCredit cc = new CompteCredit("AG675", 0, 200);
CompteBancaire cb = cc;
// cb a type déclaré CompteBancaire,
// mais type effectif CompteCredit
// cb peut être uniquement manipulé comme un
//CompteBancaire
cb.verser(1000); //OK
cb.getSolde(); //OK
cb.nouveauCredit(500); //ERREUR
//en revanche cc a type déclaré CompteCredit
cc.nouveauCredit(500); //OK
```

# Polymorphisme et type déclaré

- Le type déclaré peut être vu comme une sorte de masque sur l'objet :

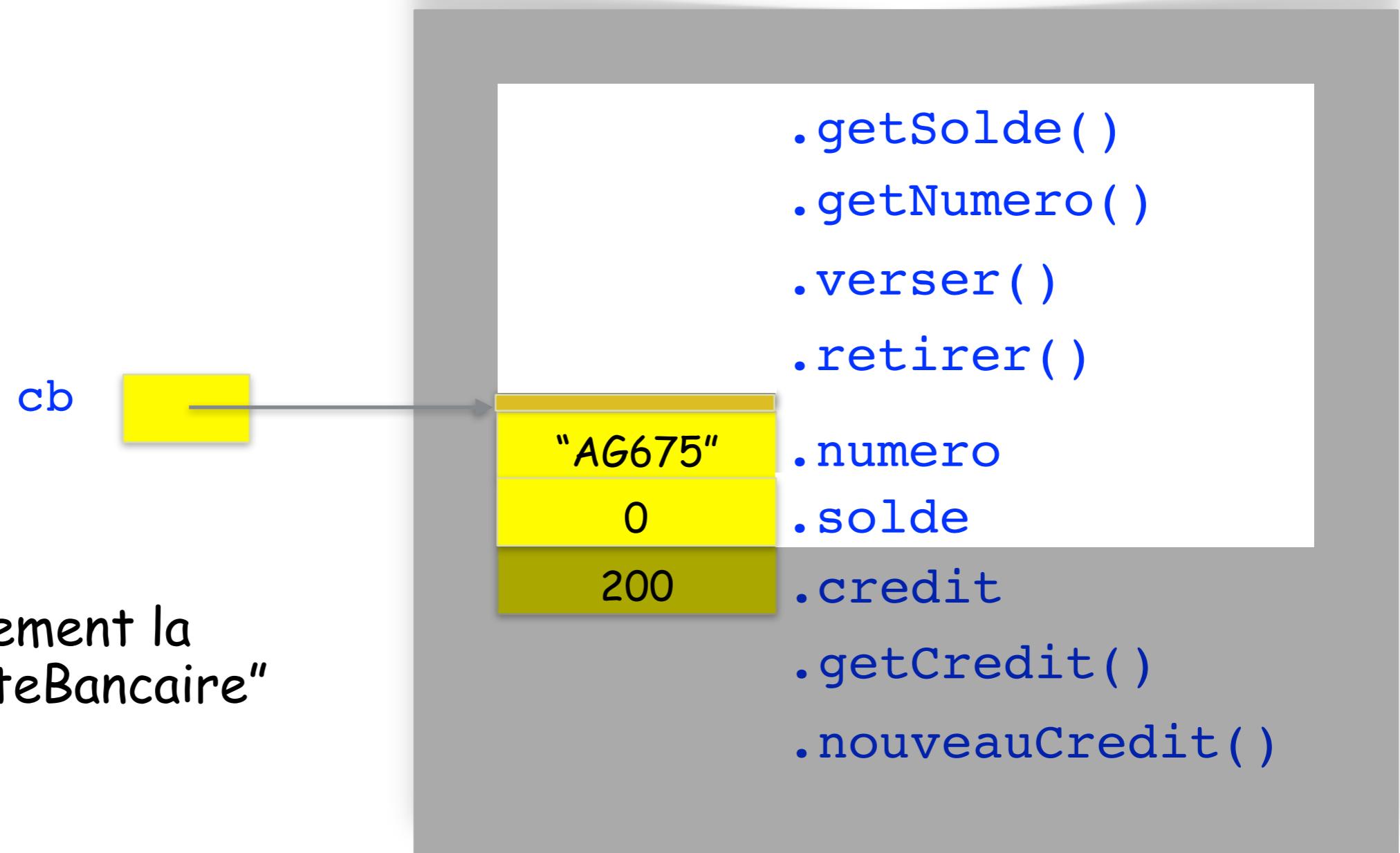
```
CompteCredit cc = new CompteCredit("AG675", 0, 200);
```



# Polymorphisme et type déclaré

- Le type déclaré peut être vu comme une sorte de masque sur l'objet :

```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);
```

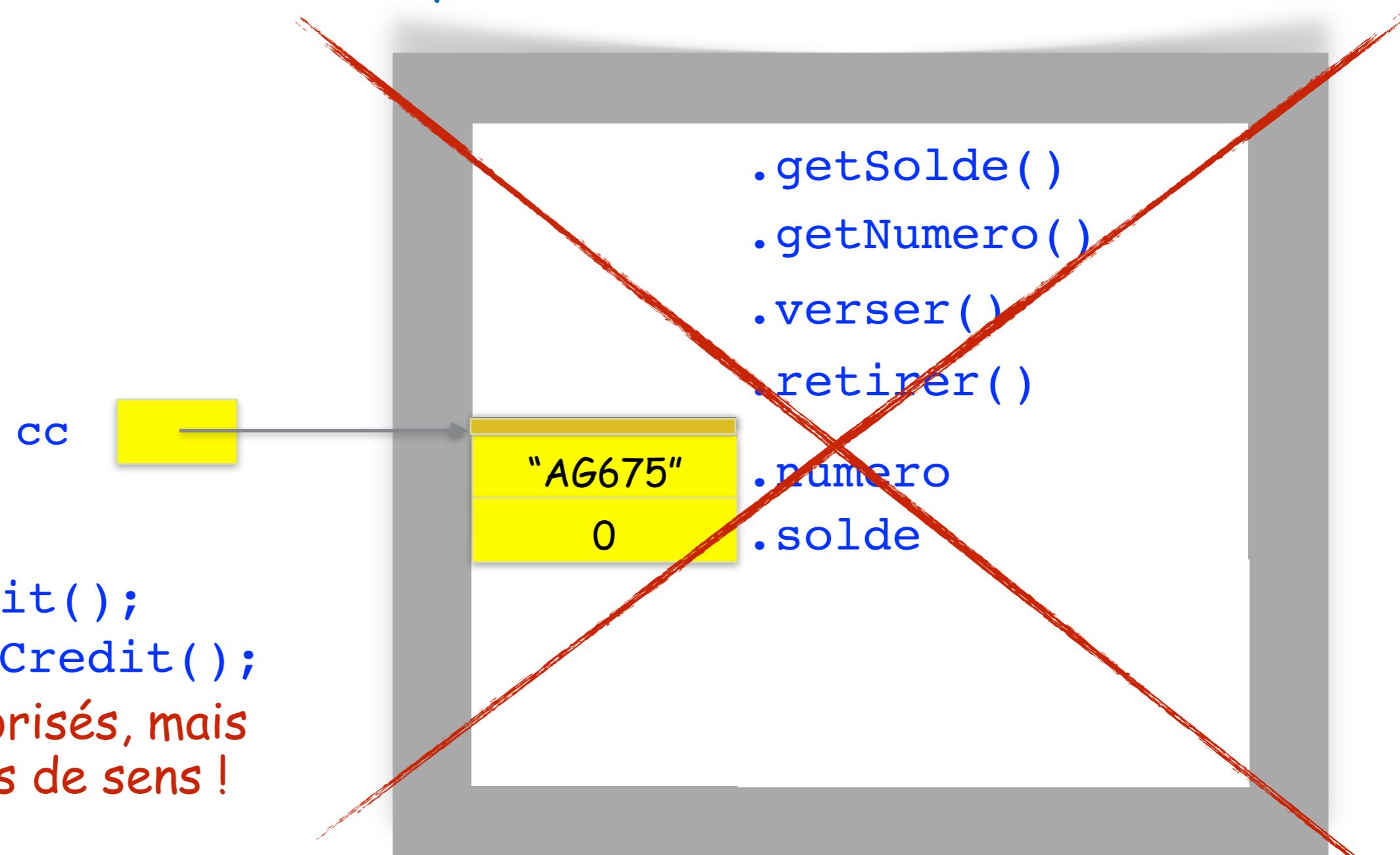


`cb` voit uniquement la  
"partie CompteBancaire"  
de l'objet

# Polymorphisme et type déclaré

- Ce qui explique pourquoi il n'est pas autorisé d'avoir :

~~CompteCredit cc = new CompteBancaire("AG675"); //ERREUR~~

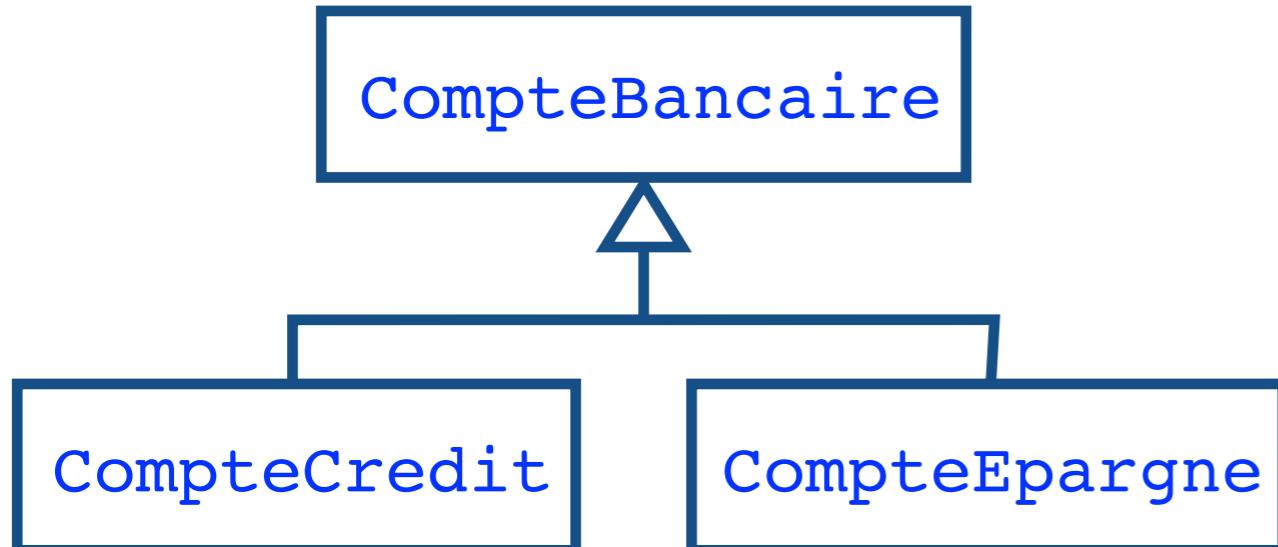


cc.getCredit();  
cc.nouveauCredit();

seraient autorisés, mais  
n'auraient pas de sens !

# Polymorphisme : intérêt

- Traiter des objets uniformément, avec les méthodes qu'ils ont en commun, même s'ils sont de types différents

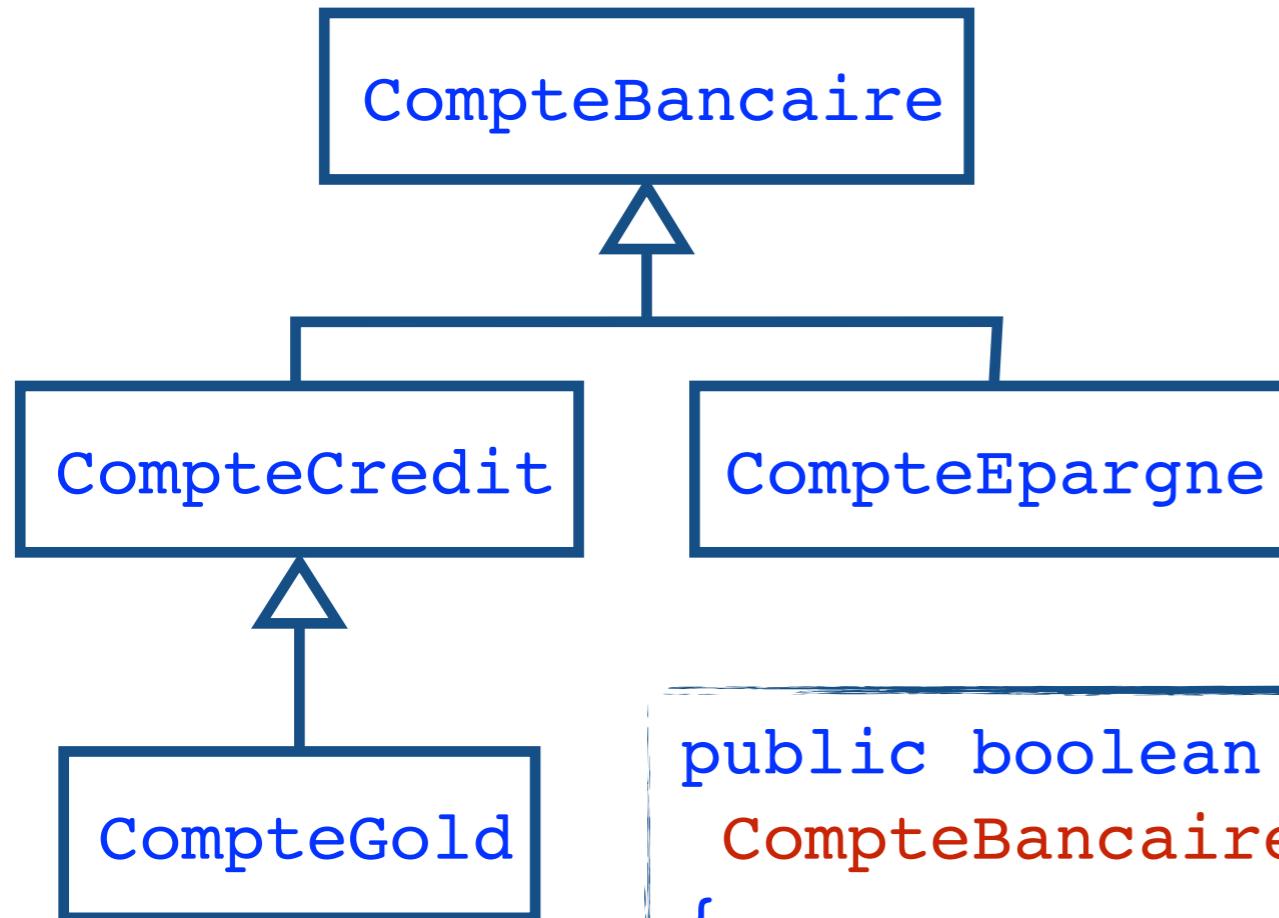


Exemple 1

```
CompteBancaire [ ] comptes =  
        new CompteBancaire[3];  
comptes[0] = new CompteCredit("C234");  
comptes[1] = new CompteEpargne("E986");  
comptes[2] = new CompteGold ("G836");  
double solde = 0;  
for (CompteBancaire c : comptes) {  
    solde += c.getSolde();  
}
```

# Polymorphisme : intérêt

- Traiter des objets uniformément, avec les méthodes qu'ils ont en commun, même s'ils sont de types différents

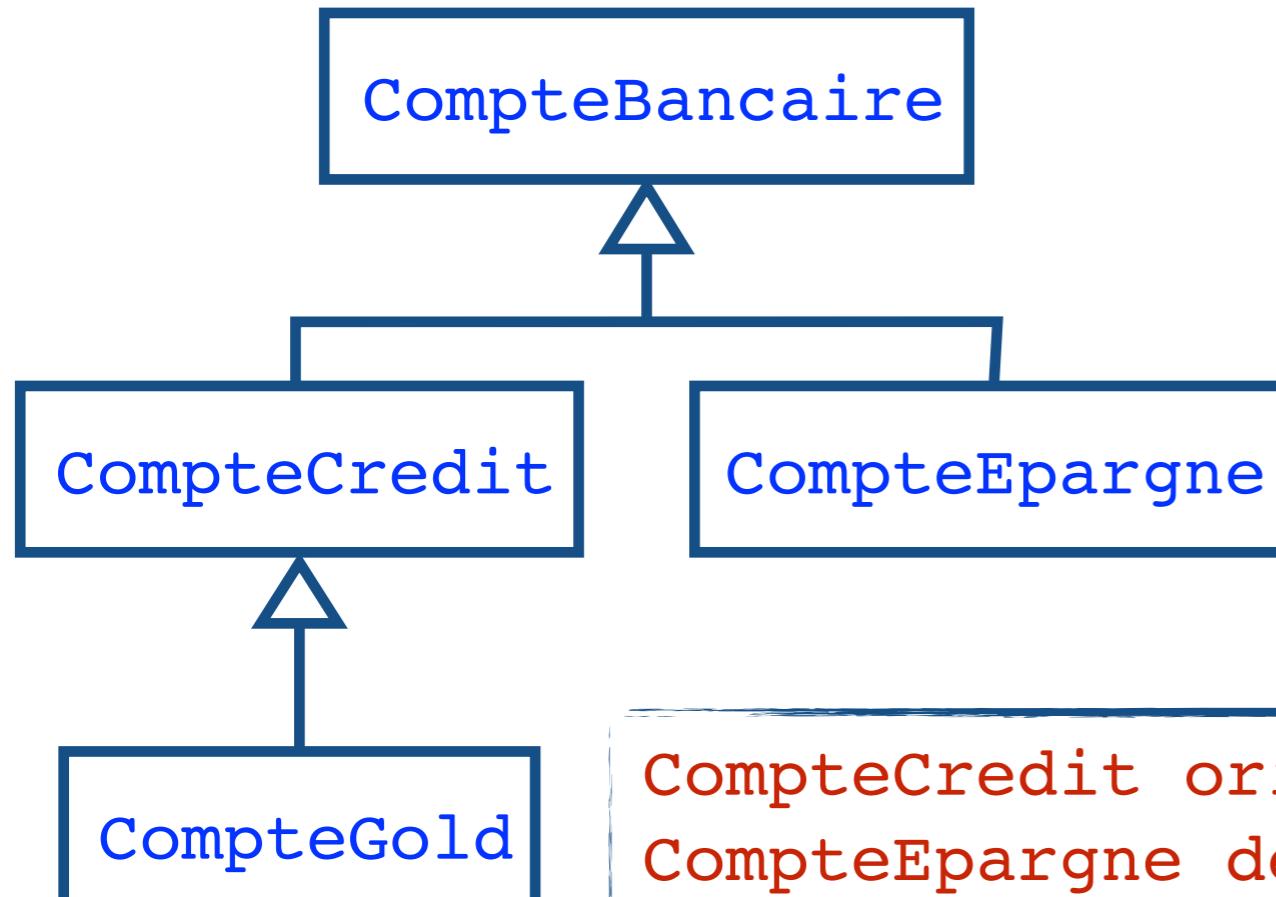


Exemple 2

```
public boolean virement (double montant,  
CompteBancaire origine, CompteBancaire dest)  
{  
    boolean b = origine.retirer(montant);  
    if (b) dest.verser(montant);  
    return b;  
}
```

# Polymorphisme : intérêt

- Traiter des objets uniformément, avec les méthodes qu'ils ont en commun, même s'ils sont de types différents



Exemple 2

```
CompteCredit orig = new CompteCredit("E986");
CompteEpargne dest = new CompteEpargne ("G836");
virement (orig, dest, 100);
```

# Dynamic binding

- Le polymorphisme est très puissant en présence d'overriding
- Rappel : retirer est redéfinie par CompteCredit

```
public class CompteBancaire {  
    ...  
    public boolean retirer( double montant ) {...}  
}  
public class CompteCredit extends CompteBancaire {  
    ...  
    @Override  
    public boolean retirer( double montant ) {...}  
}
```

# Dynamic binding

- Le polymorphisme est très puissant en présence d'overriding
- Rappel : retirer est redéfinie par CompteCredit

```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);
cb.retirer(100);
```



invoque la fonction retirer redéfinie dans CompteCredit  
même si cb est de type déclaré CompteBancaire !

- **Dynamic binding** : C'est le type effectif d'un objet qui détermine la définition de la méthode à utiliser
  - le type effectif est connu seulement à l'exécution
  - => le lien (binding) entre le nom d'une fonction et sa définition est fait seulement à l'exécution (dynamic)

# Dynamic binding : remarque

Remarque :

- le **type déclaré** détermine **quelles méthodes** on peut invoquer
  - fait statiquement (en phase de compilation)

```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);
cb.retirer(...); // ok
cb.getCredit(); //erreur à la compilation
```

- le **type effectif** détermine **quelle définition** est à utiliser pour ces méthodes
  - fait dynamiquement (en phase d'exécution)

```
cb.retirer(...); // retirer de CompteCredit
```

# Dynamic binding : utilité

- Pour maintenir cohérent l'état d'un objet, un objet doit être toujours manipulé avec sa propre version des méthodes
- Exemple

```
public boolean virement (double montant,
                        CompteBancaire origine, CompteBancaire dest){
    boolean b = origine.retirer(montant);
    if (b) dest.verser(montant);
    return b;
}

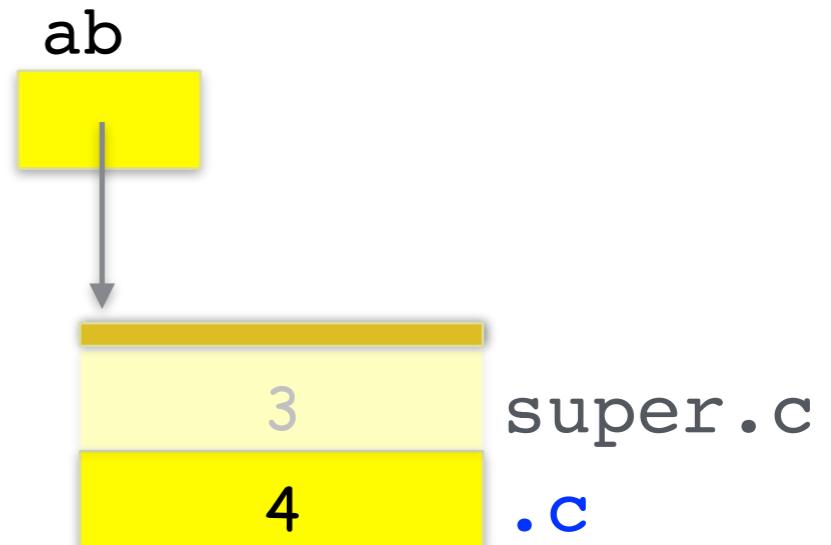
CompteCredit orig = new CompteCredit("E986", 1000, 200);
CompteEpargne dest = new CompteEpargne ("G836");
virement (orig, dest,1100); // virement effectué
```

- Si la définition de retirer utilisée était celle de CompteBancaire, le virement ci dessus (valide) ne serait pas effectué

# Dynamic binding vs static binding

- Le dynamic binding ne s'applique pas au variables !
  - le lien entre le nom d'une variable et sa définition a lieu à la compilation (static binding) => ne dépend pas du type effectif
  - Redéfinir un champ n'a donc pas le même effet que redéfinir une fonction

```
class A {  
    int c = 3;  
}  
class B extends A {  
    int c = 4;  
}  
...  
B b = new B();  
A ab = b;  
System.out.println(b.c); // affiche 4  
System.out.println(ab.c); // affiche 3, static binding  
System.out.println(((B)ab).c); // affiche 4
```



# Dynamic binding vs static binding

- Autre conséquence du static binding : une méthode de classe A peut uniquement accéder aux champs de la classe A (ou de ces ancêtres)

```
class A {  
    int c = 3;  
    public int getC () {  
        return c; // c de A: déterminé à la compilation  
    }  
}  
class B extends A {  
    int c = 4;  
}  
...  
B ob = new B();  
System.out.println(ob.getc()); // affiche 3  
System.out.println(ob.c); // affiche 4
```

# Dynamic binding vs static binding

- Une combinaison des deux : exemple

```
class A {  
    int c = 3;  
    public int getC () {  
        return c; // c de A: déterminé à la compilation  
    }  
}  
class B extends A {  
    int c = 4;  
    public int getC () {  
        return c; // c de B: déterminé à la compilation  
    }  
}  
...  
B ob = new B(); A ab = ob;  
System.out.println(ob.getC()); // affiche 4  
System.out.println(ab.getC()); // affiche 4
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

a.f();  
b.f();  
ab.f();  
((B)ab).f();  
((A)b).f();

a.g();  
b.g();  
ab.g();  
((B)ab).g();  
((A)b).g();

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f();  
ab.f();  
((B)ab).f();  
((A)b).f();
```

```
a.g();  
b.g();  
ab.g();  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f();  
ab.f();  
((B)ab).f();  
((A)b).f();
```

```
a.g(); //g de A  
b.g();  
ab.g();  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f();  
((B)ab).f();  
((A)b).f();
```

```
a.g(); //g de A  
b.g();  
ab.g();  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f();  
((B)ab).f();  
((A)b).f();
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g();  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f();  
((A)b).f();
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g();  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f();  
((A)b).f();
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g(); //g de B  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f(); //static f B  
((A)b).f();
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g(); //g de B  
((B)ab).g();  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f(); //static f B  
((A)b).f();
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g(); //g de B  
((B)ab).g(); //g de B  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f(); //static f B  
((A)b).f(); //static f A
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g(); //g de B  
((B)ab).g(); //g de B  
((A)b).g();
```

# Dynamic vs static binding : méthodes static

- Une méthode de classe (static) peut être redéfinie ou surchargée comme toute autre méthode
- Mais **static binding** ! (même comportement que les variables)

```
class A{  
    static void f(){System.out.println("static f A");}  
    void g(){System.out.println("g de A"); }  
}  
  
class B extends A{  
    static void f(){System.out.println("static f B");}  
    void g(){System.out.println("g de B"); }  
}  
//...  
A a=new A(); B b=new B(); A ab = b;
```

```
a.f(); //static f A  
b.f(); //static f B  
ab.f(); //static f A  
((B)ab).f(); //static f B  
((A)b).f(); //static f A
```

```
a.g(); //g de A  
b.g(); //g de B  
ab.g(); //g de B  
((B)ab).g(); //g de B  
((A)b).g(); //g de B
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f();  
b.f();  
b.g();  
b.h();  
ab.f();  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f();  
b.g();  
b.h();  
ab.f();  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g();  
b.h();  
ab.f();  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g(); //-> static f B  
b.h();  
ab.f();  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g(); //-> static f B  
b.h(); //-> static f A  
ab.f();  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g(); //-> static f B  
b.h(); //-> static f A  
ab.f(); //-> static f A  
ab.h();  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g(); //-> static f B  
b.h(); //-> static f A  
ab.f(); //-> static f A  
ab.h(); //-> static f A  
((B)ab).g();
```

# Dynamic vs static binding : méthodes static

## Exemple

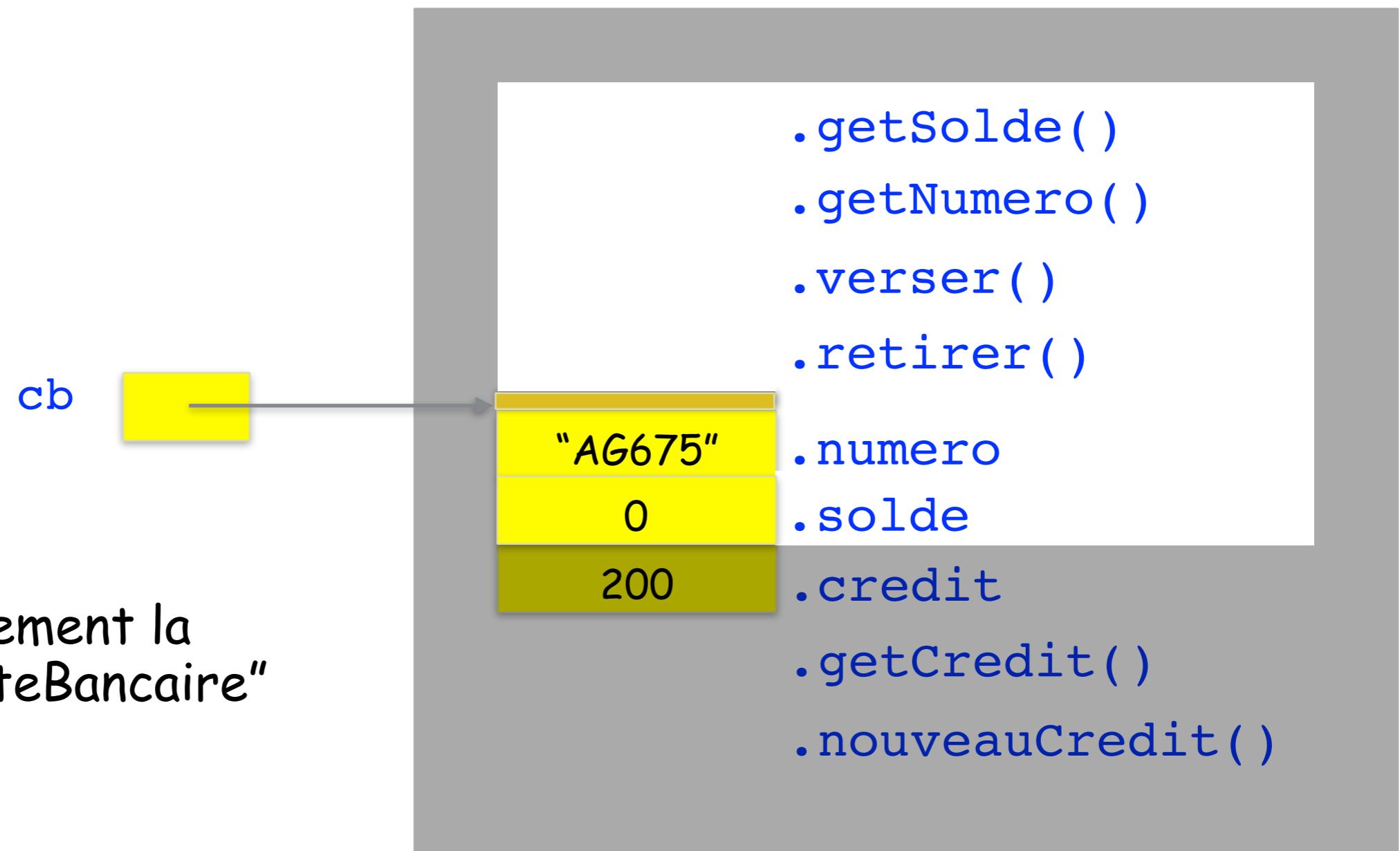
```
public class A {  
    public static void f() {  
        System.out.println  
            ("static f A");  
    }  
    public void h() {f();}  
}  
  
public class B extends A {  
    public static void f() {  
        System.out.println  
            ("static f B");  
    }  
    public void g()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

```
A a = new A();  
B b = new B();  
A ab = b;  
  
a.f(); //-> static f A  
b.f(); //-> static f B  
b.g(); //-> static f B  
b.h(); //-> static f A  
ab.f(); //-> static f A  
ab.h(); //-> static f A  
((B)ab).g(); //-> static f B
```

# Rappel

- Le type déclaré peut être vu comme une sorte de masque sur l'objet :

```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);
```

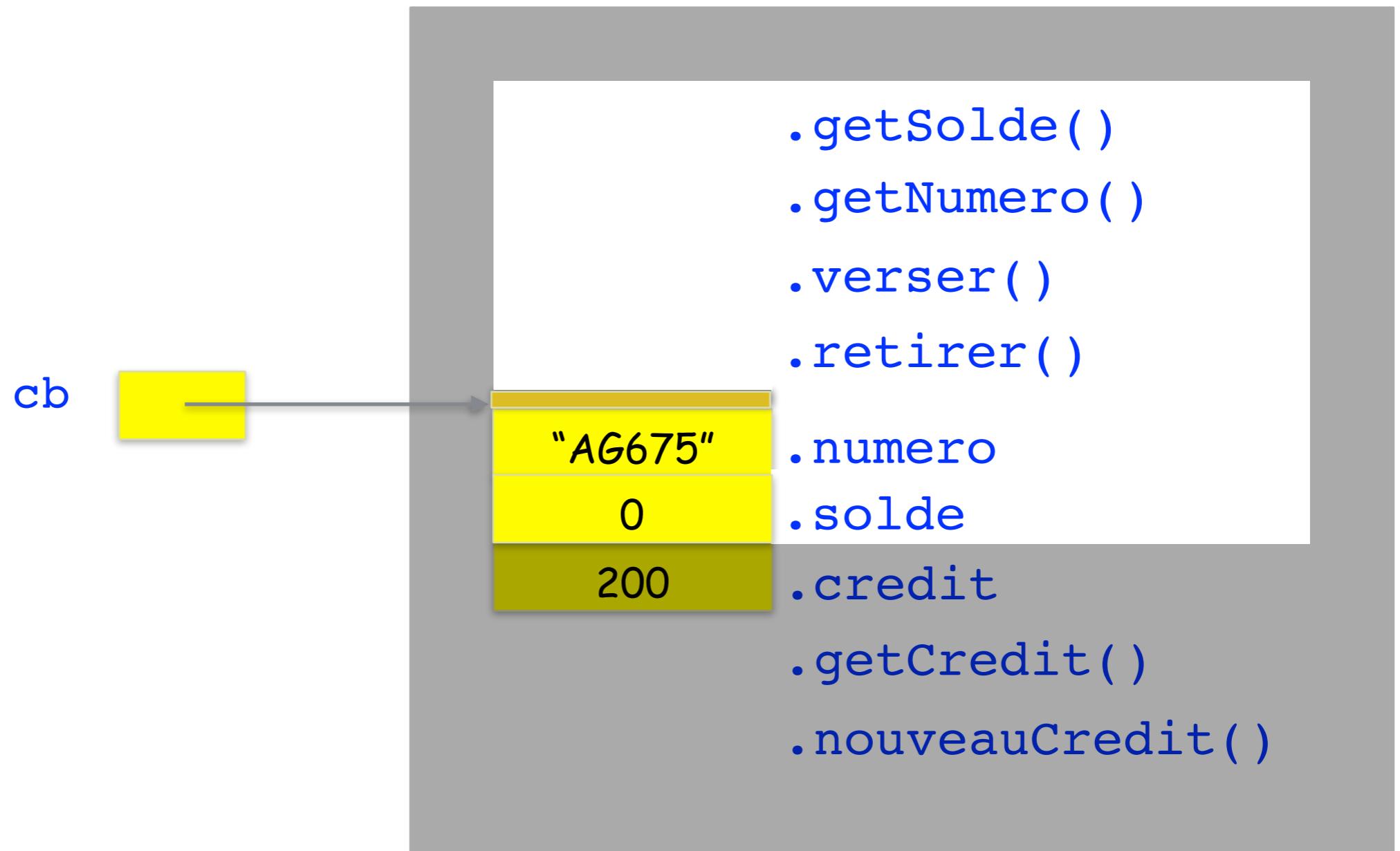


`cb` voit uniquement la  
"partie CompteBancaire"  
de l'objet

# Transtypage (casting)

- Pour utiliser un objet entièrement il est possible de forcer le **changement de son type déclaré**

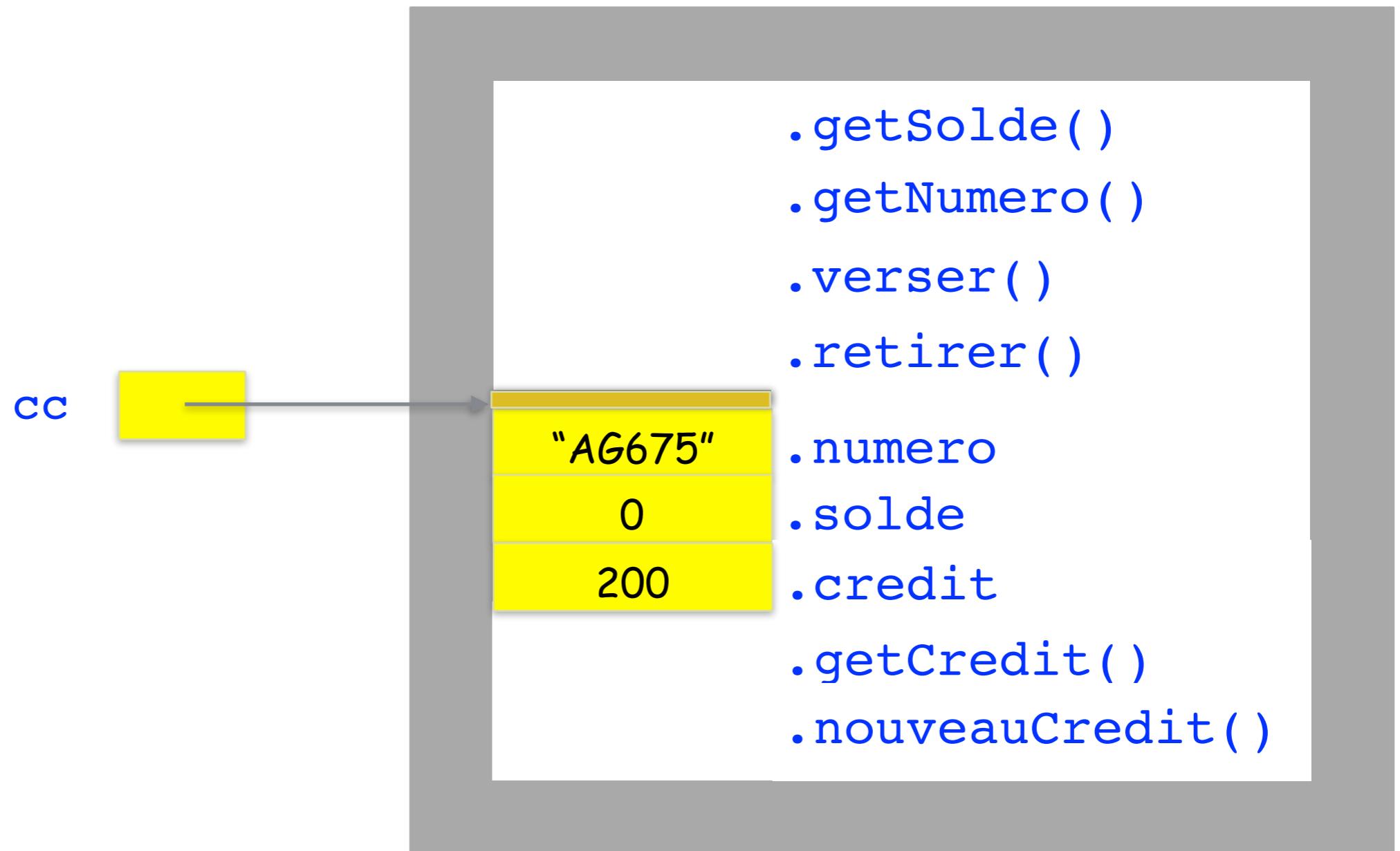
```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);  
CompteCredit cc = (CompteCredit) cb;
```



# Transtypage (casting)

- Pour utiliser un objet entièrement il est possible de forcer le **changement de son type déclaré**

```
CompteBancaire cb = new CompteCredit("AG675", 0, 200);  
CompteCredit cc = (CompteCredit) cb;
```



# Transtypage (casting) et instanceof

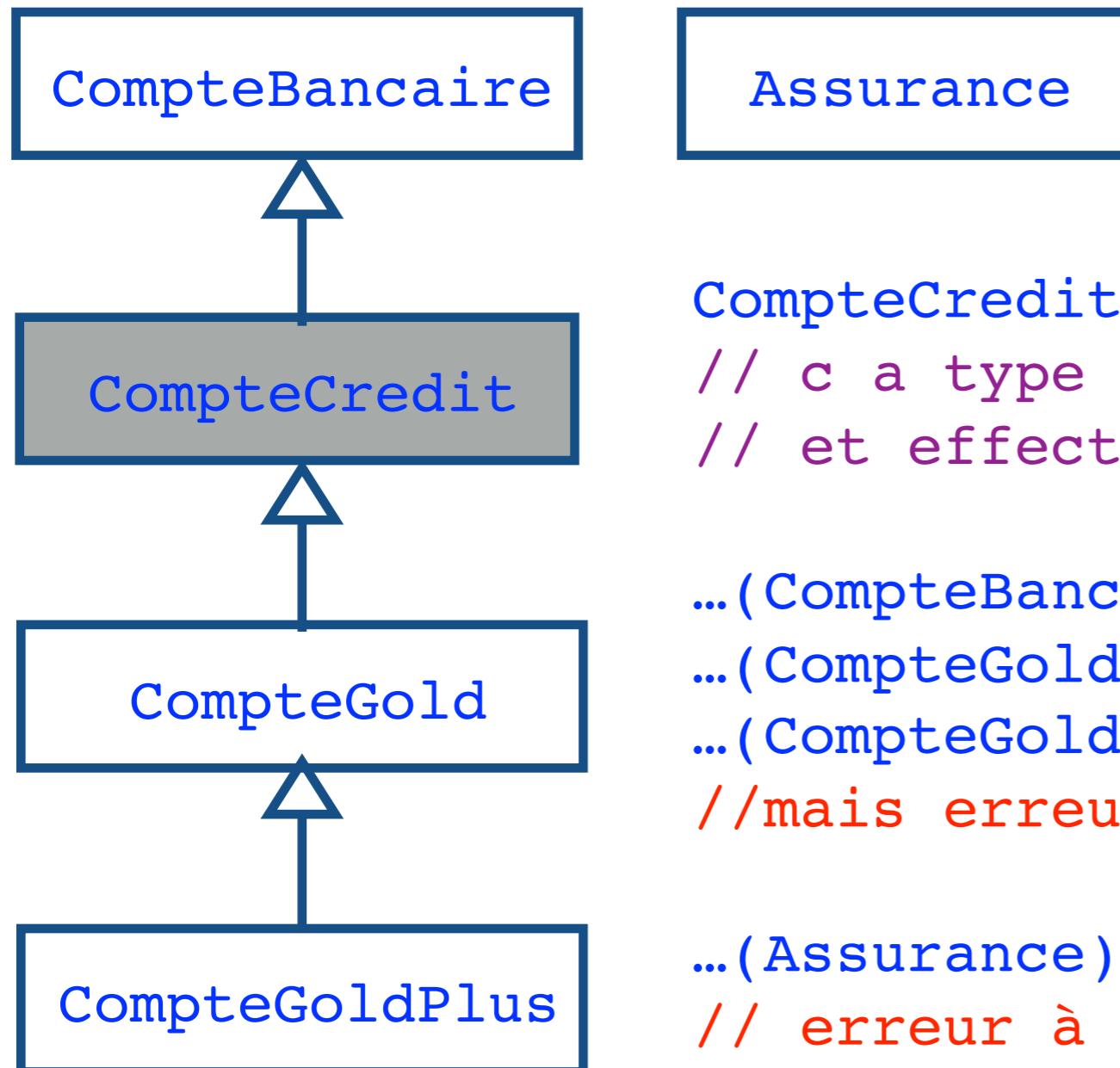
## (CompteCredit) cb

- Renvoie cb converti en type déclaré CompteCredit
- À effectuer uniquement si on est sûr que cb fait référence à un objet de type CompteCredit
- pour le vérifier dynamiquement : instanceof

```
CompteBancaire cb;  
CompteCredit cc;  
...  
if (cb instanceof CompteCredit) {  
    cc = (CompteCredit) cb;  
}
```

# Casting : règles

- Sur les types référence, le casting est autorisé uniquement si les types déclarés appartiennent à une même hiérarchie d'héritage



```
CompteCredit cc = new CompteGold(...);  
// c a type déclaré CompteCredit  
// et effectif CompteGold
```

```
...(CompteBancaire) cc; //OK upcasting  
...(CompteGold) cc; //OK downcasting sûr  
...(CompteGoldPlus) cc; //autorisé en compil.  
//mais erreur à l'execution
```

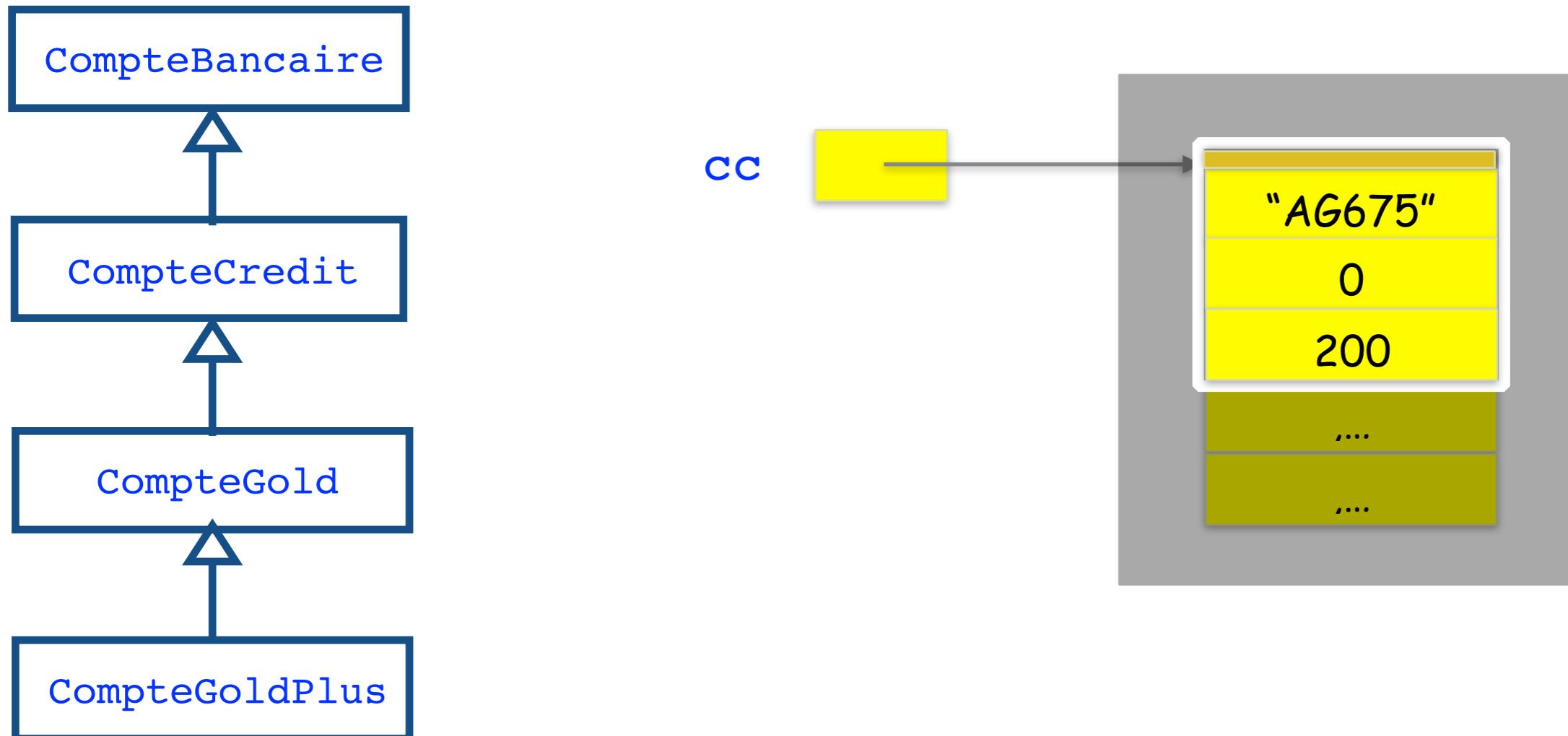
```
...(Assurance) cc;  
// erreur à la compilation
```

# Upcasting : toujours sûr

- Conversion vers un type de plus haut niveau dans la hiérarchie
- Toujours sûr : réduit le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteBancaire cb = (CompteBancaire) cc; //OK upcasting
```

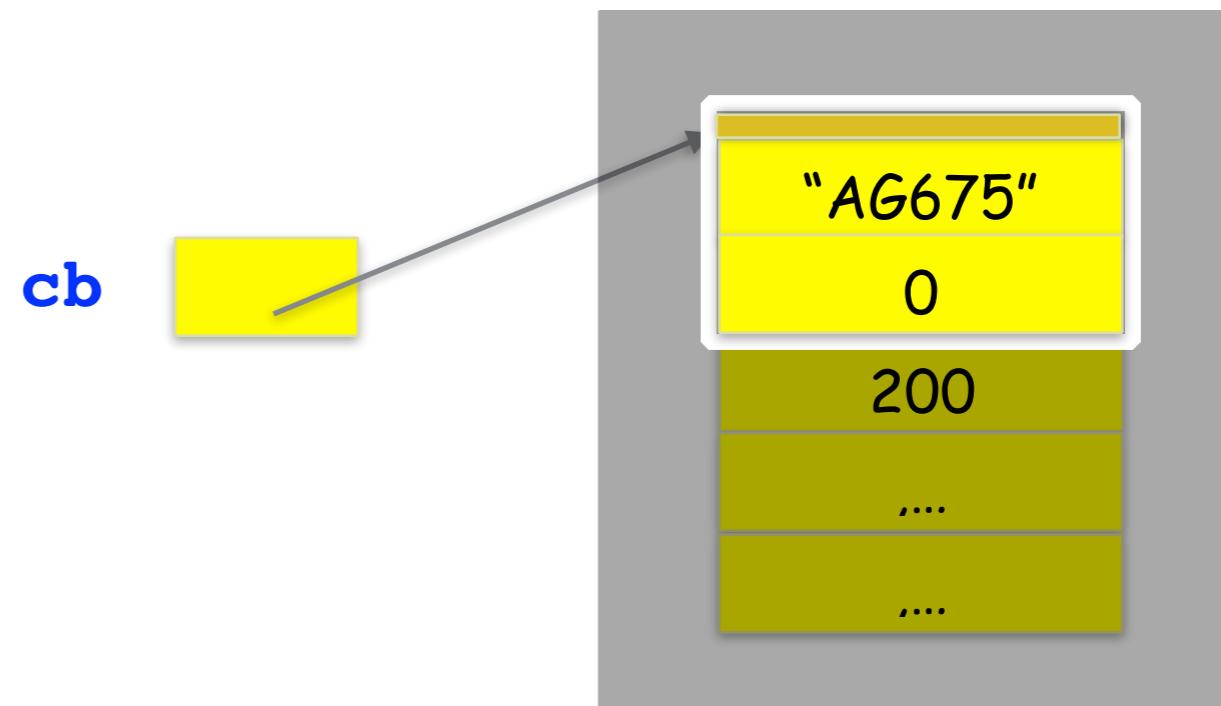
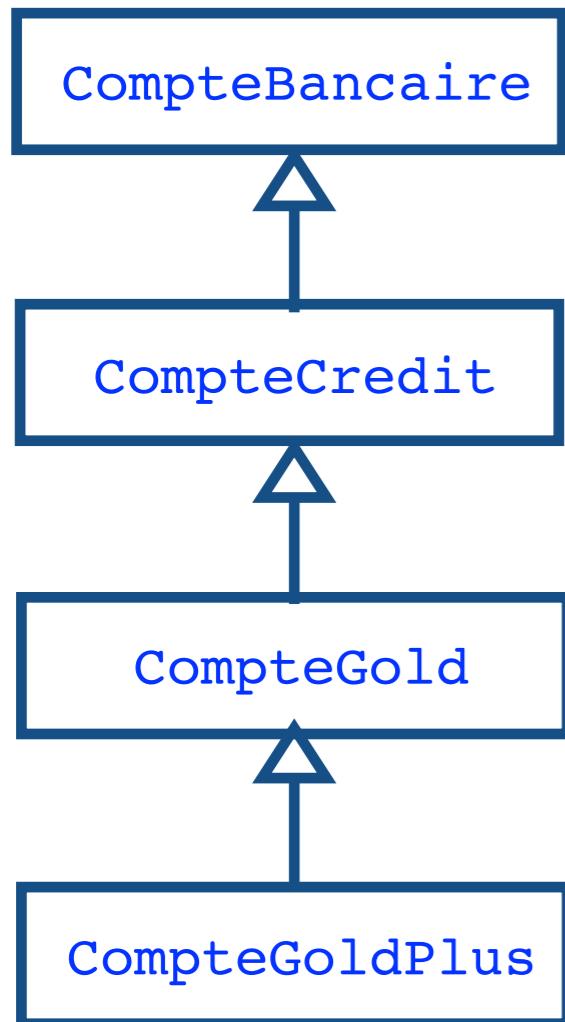


# Upcasting : toujours sûr

- Conversion vers un type de plus haut niveau dans la hiérarchie
- Toujours sûr : réduit le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteBancaire cb = (CompteBancaire) cc; //OK upcasting
```



# Upcasting : toujours sûr

- Conversion vers un type de plus haut niveau dans la hiérarchie
- Toujours sûr : réduit le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteBancaire cb = (CompteBancaire) cc; //OK upcasting
```

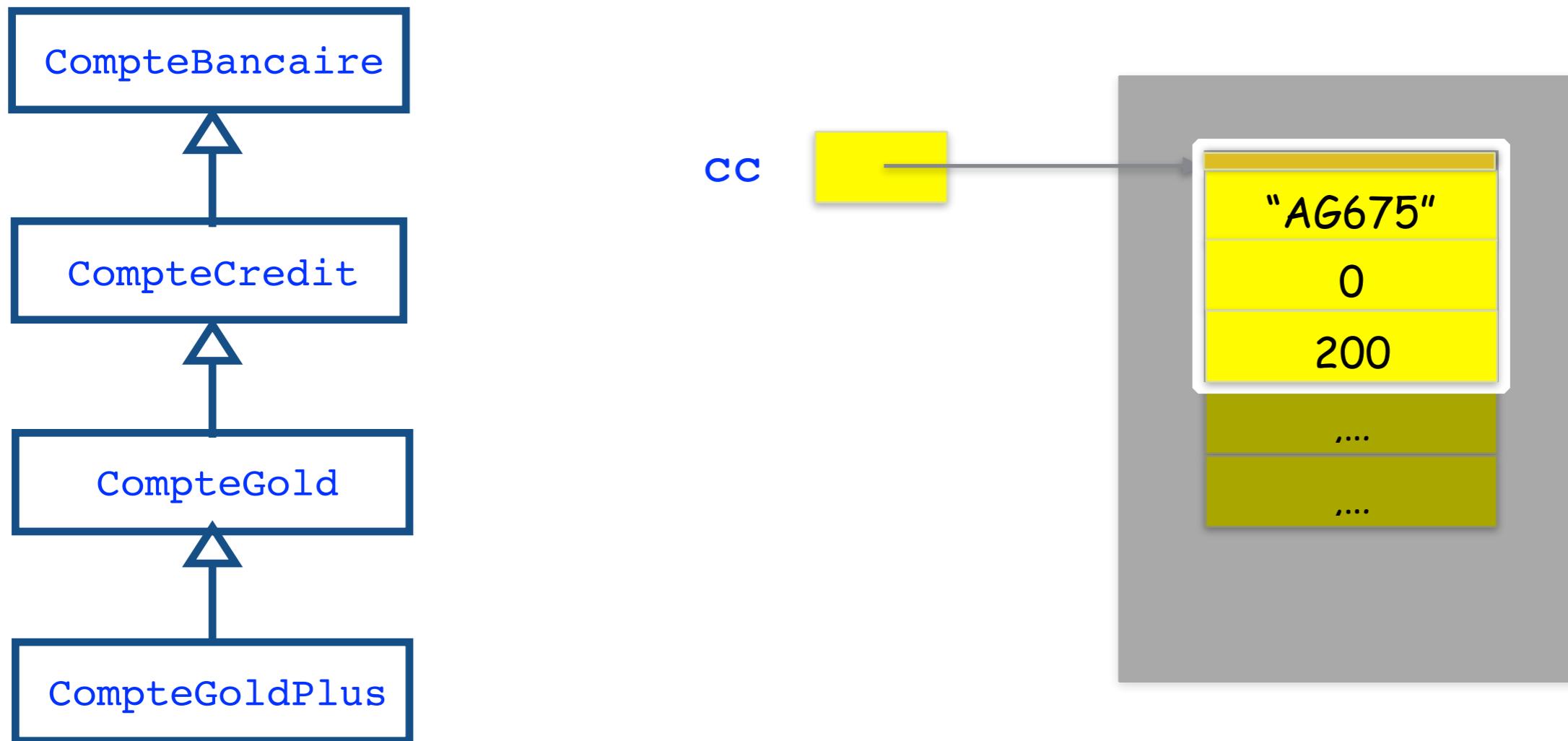
- Équivaut à une affectation (upcasting implicite)

```
CompteBancaire cb = cc;
```

# Downcasting

- Conversion vers un type de plus bas niveau dans la hiérarchie
- Élargie le "masque"

```
CompteCredit cc = new CompteGold(...);
```

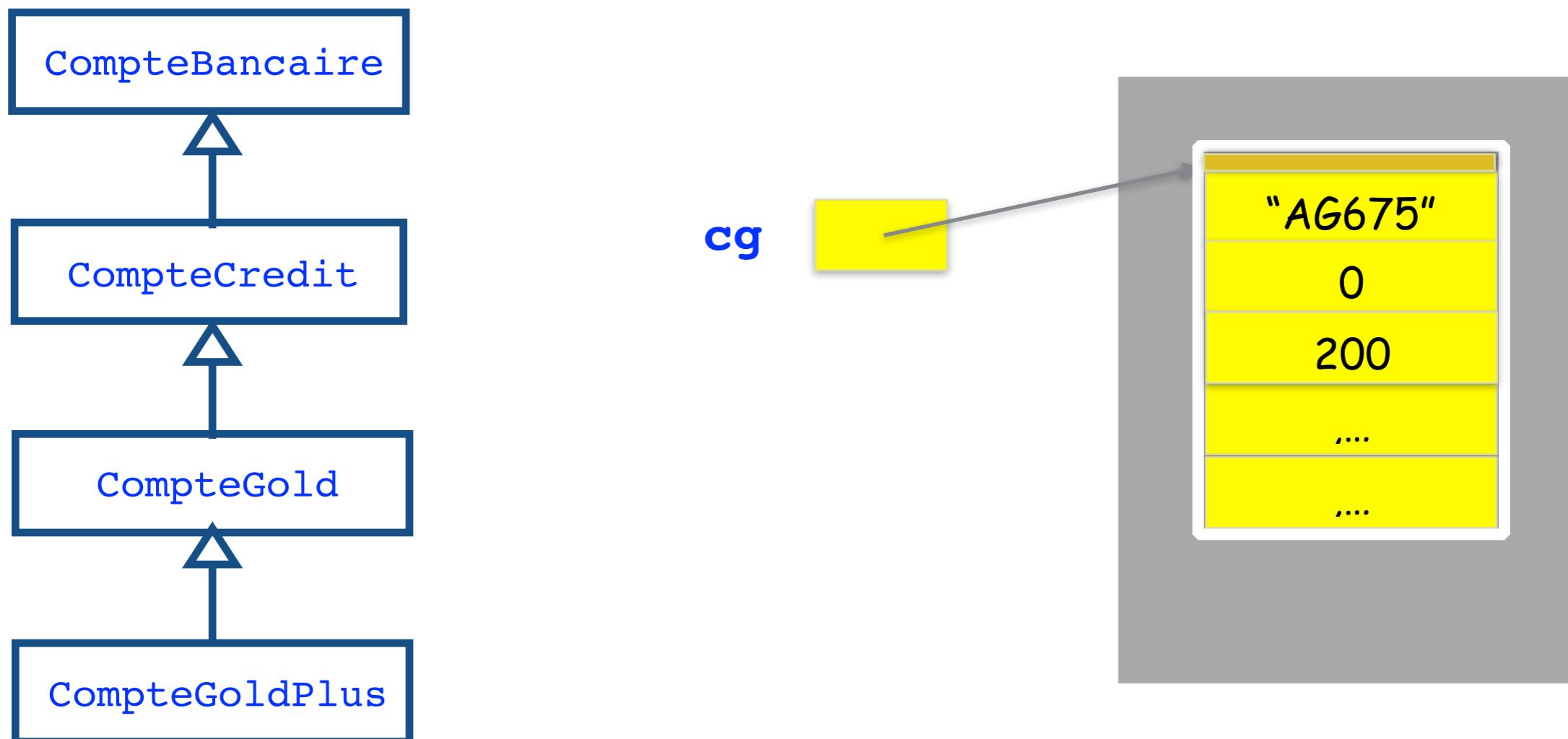


# Downcasting

- Conversion vers un type de plus bas niveau dans la hiérarchie
- Élargie le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteGold cg = (CompteGold) cc; //OK downcasting sûr
```



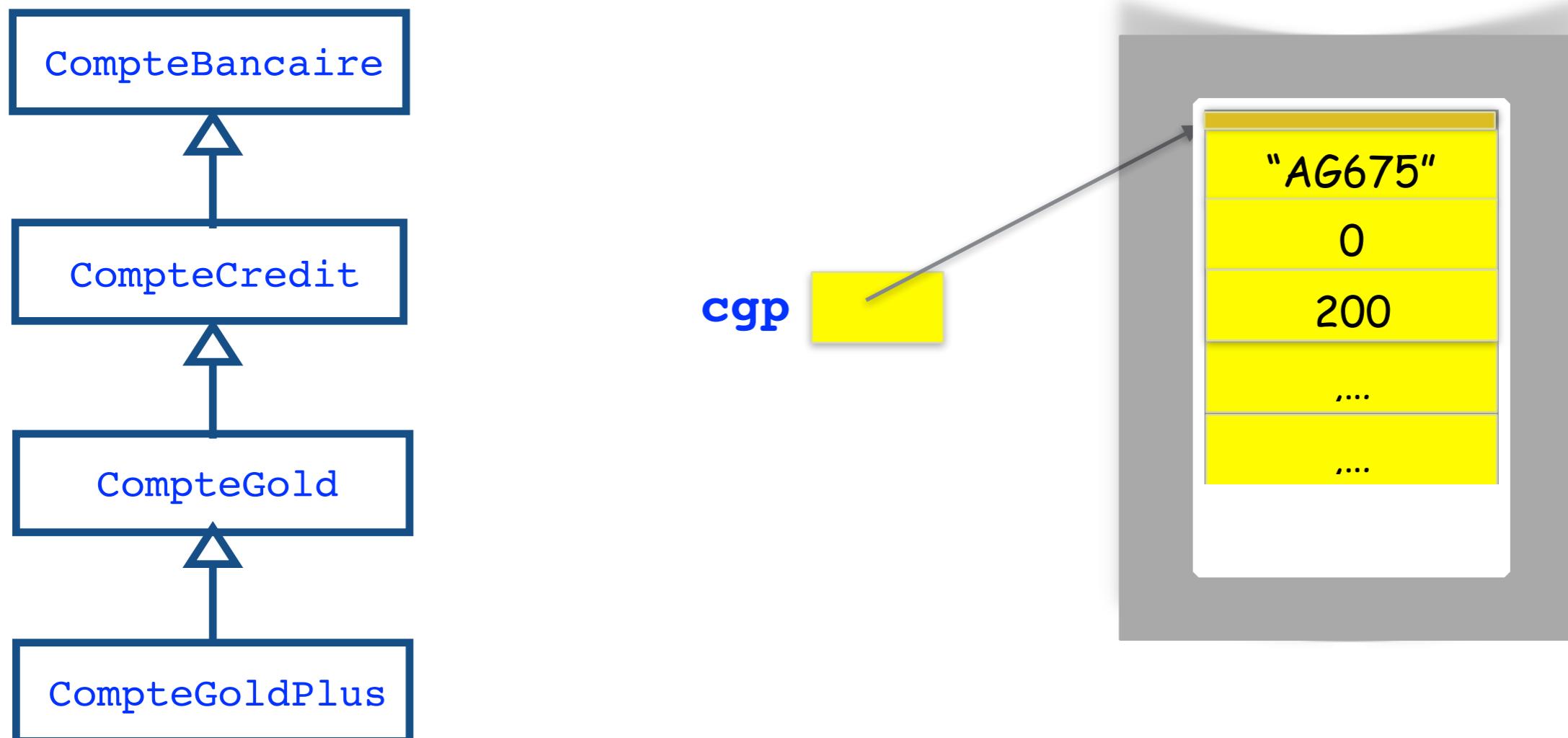
# Downcasting

- Conversion vers un type de plus bas niveau dans la hiérarchie
- Élargie le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteGold cg = (CompteGold) cc; //OK downcasting sûr
```

```
CompteGoldPlus cgplus = (CompteGoldPlus) cc; //ERREUR runtime
```



# Downcasting

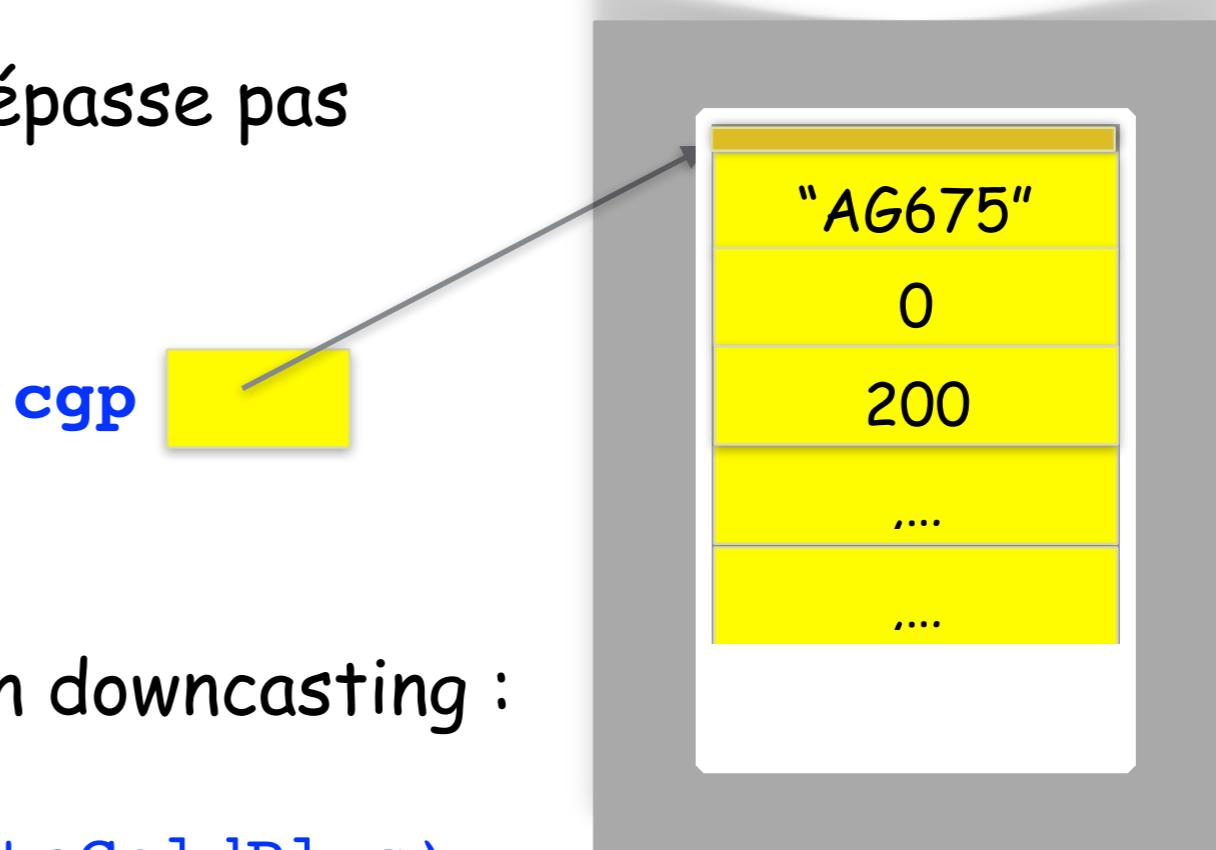
- Conversion vers un type de plus bas niveau dans la hiérarchie
- Élargie le "masque"

```
CompteCredit cc = new CompteGold(...);
```

```
CompteGold cg = (CompteGold) cc; //OK downcasting sûr
```

```
CompteGoldPlus cg = (CompteGoldPlus) cc; //ERREUR runtime
```

- Sûr uniquement s'il ne dépasse pas le type effectif.



- Toujours tester avant un downcasting :

```
if (cc instanceof CompteGoldPlus)
{ CompteGoldPlus cg = (CompteGoldPlus) cc;...}
```

# Dowcasting : "pattern matching" pour instanceof

Raccourci pour instanceof introduit en Java 15 et 16:

- Au lieu d'écrire :

```
if (cc instanceof CompteGoldPlus)
{ CompteGoldPlus cgp = (CompteGoldPlus) cc;
  //utilisation de cgp de type CompteGoldPlus ...
}
```

- On peut écrire de façon plus synthétique :

```
if (cc instanceof CompteGoldPlus cgp)
{ //utilisation de cgp de type CompteGoldPlus...
}
```

- évite le casting explicite

# Exemple d'utilisation d'une hiérarchie de classes

---

code/poo/banque/Banque.java

# Contrôle d'accès à la super-classe

- Chaque membre de la super-classe est hérité par la sous-classe
- Toutefois, comme toute autre classe, la sous-classe :
  - n'a pas accès aux membres private de la super-classe
  - a accès aux membres avec visibilité package uniquement si elle se trouve dans le même package que la super-classe
  - a accès aux membres protected et public de la super-classe
- Le contrôle d'accès est vérifié à la compilation

# Contrôle d'accès à la super-classe

```
public class CompteBancaire {  
    private String numero;  
    ...  
}  
public class CompteCredit extends CompteBancaire{  
    ...  
    //ne peut pas manipuler numero directement, pour ce faire  
    //doit utiliser les méthodes de CompteBancaire  
    public CompteCredit( String numero ){  
        this.numero = numero; //ERREUR  
        super( numero );  
    }  
    ...  
}
```

# Contrôle d'accès à la super-classe : protected

- Les membres protected sont visibles uniquement par les sous classes et par les classes du même package
- On peut donc déclarer protected les champs ou méthodes qu'on ne veut pas rendre globalement visibles, mais on veut que les sous-classes puissent manipuler ou redéfinir. Exemples typiques :
  - méthodes qui ont une implémentation significative uniquement dans les sous-classes, où seront redéfinies public (exemple : `clone()`)
  - champs que la sous-classe doit manipuler plus finement qu'avec les méthodes de la super-classe exemple : protected `solde` est nécessaire pour la redéfinition de retirer
- Utiliser avec précaution les champs protected ! risque de violer l'encapsulation
  - une classe externe introduite dans le même package a accès aux champs protected !

# Ordre d'initialisation en l'absence d'héritage (rappel)

- Quand un constructeur est invoqué, l'objet est déjà créé et ses champs initialisés aux valeurs par défaut
  - champs numériques : **0**
  - champs référence : **null**
  - champs boolean : **false**
- Invocation d'un constructeur de classe **C** \* :
  1. Si le constructeur invoque **this(...)** : Invocation (réursive) de l'autre constructeur de **C**.  
Sinon: exécution des initialiseurs et des blocs d'initialisation de **C**, dans l'ordre de définition

```
class A { ...
    int i =1;
    ...
    { i++; }
}
```

2. Exécution du corps du constructeur (**this()** n'en fait pas partie)

\* en l'absence d'héritage

# Ordre d'initialisation dans le cas général

- Quand un constructeur est invoqué, l'objet est déjà crée et ses champs initialisés aux valeurs par défaut (`0`, `null`, `false`)
- Invocation d'un constructeur de classe *C* :
  1. Si le constructeur invoque `this(...)` : Invocation (réursive) de l'autre constructeur de *C*.  
Sinon :
    - A. Invocation (réursive) du constructeur de la classe parent de *C* (appel `super(...)` explicite ou implicite )
    - B. exécution des initialisateurs et des blocs d'initialisation de *C*, dans l'ordre de définition
  2. Exécution du corps du constructeur (`this(...)` et `super(...)` n'en font pas partie)

# Exemple d'initialisation

```
class Mere {  
    int i = 1;  
    Mere () {  
        System.out.println("constructeur Mere() " + this);  
    }  
    public String toString() { return "i = " + i;}  
}  
class Fille extends Mere{  
    int j = 2;  
    {System.out.println("bloc initialisation Fille " + this);}  
    Fille() {  
        super();  
        System.out.println("constructeur Fille() " + this);  
    }  
    Fille(int k) {  
        this();  
        System.out.printf("constructeur Fille(%s) %s\n",k, this);  
    }  
    public String toString() {return "i = " + i +", j = " + j}  
}
```

# Exemple d'initialisation

```
public static void main (String[ ] args) {  
    Fille f = new Fille (3);  
}  
}
```

- Donne lieu aux opérations suivantes dans l'ordre :

i = 0 j = 0

Mere()  
i = 1

affichage: constructeur Mere() i = 1, j = 0

j=2

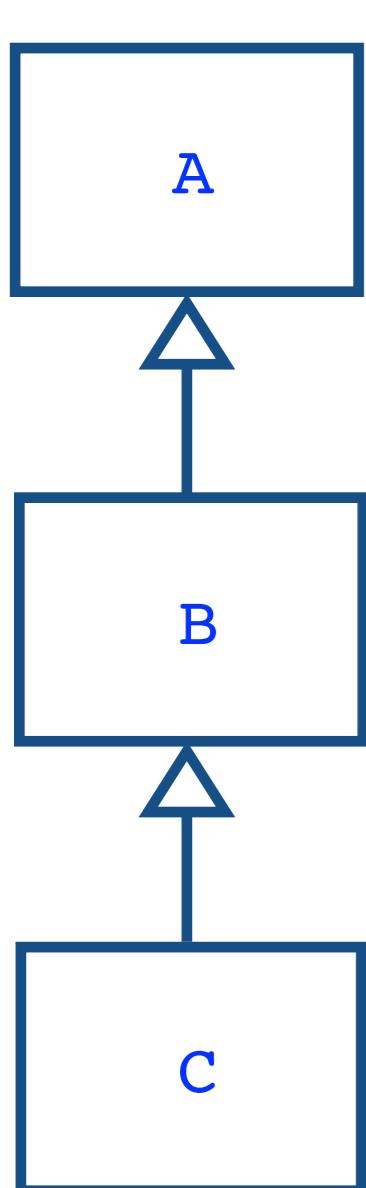
affichage: bloc initialisation Fille i = 1, j = 2

affichage: constructeur Fille() i = 1, j = 2

affichage: constructeur Fille(3) i = 1, j = 2

Fille(3)

# Ordre d'initialisation : exemple



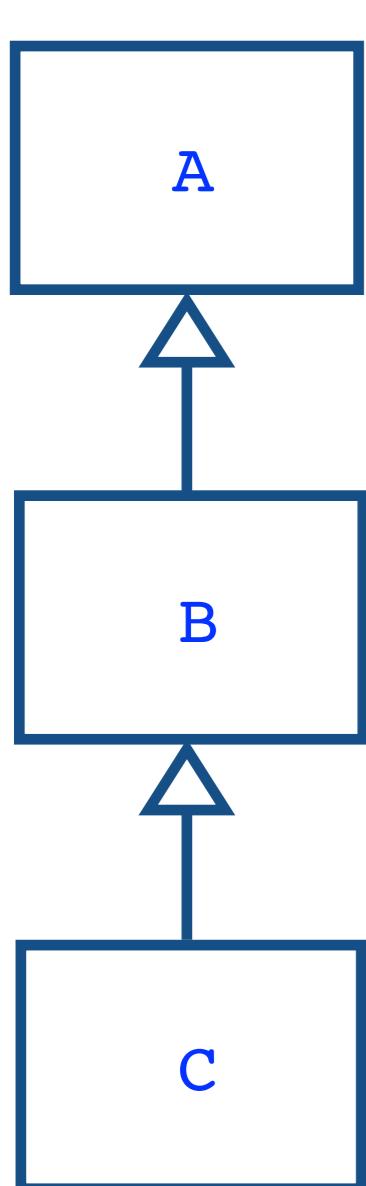
```
class A {  
    int a = 1;  
    { bloc1 }  
}  
  
class B extends A {  
    int b = 2;  
    { bloc2 }  
    B(int i)  
        {b = i;}  
}  
  
class C extends B {  
    int c = 3;  
    { bloc3 }  
    C(int i, int j) {  
        super(i); c = j;}  
}
```

C obj = new C(5, 7)  
génère les initialisations  
suivantes sur les champs de obj

a = 0  
b = 0  
c = 0  
a = 1  
{ bloc1 }  
b = 2  
{ bloc2 }  
b = 5  
c = 3  
{ bloc3 }  
c = 7

C(5, 7) [ B(5) [ A() ] ]

# Ordre d'initialisation : résumé



```
class A {  
    int a = 1;  
    { bloc1 }  
}  
  
class B extends A {  
    int b = 2;  
    { bloc2 }  
    B(int i)  
        {b = i;}  
}  
  
class C extends B {  
    int c = 3;  
    { bloc3 }  
    C(int i, int j) {  
        super(i); c = j;}  
}
```

Invocation du constructeur de C en résumé :

- initialisation par défaut
  - 0, null, false
- puis, pour chaque super-classe de C dans la hiérarchie, du plus haut niveau jusqu'à C:
  - initialisateurs + blocs + corps constructeur

# Classes scellées

- En java 17 "sealed classes" :

```
public sealed class Shape  
    permits Circle, Rectangle, Square {...}
```

- Cela veut dire que (même si la classe est public) les seules classes qui peuvent étendre Shape sont les classes Circle, Rectangle et Square.

# La classe Object

- Toute classe hérite de la classe Object
  - racine de la hiérarchie d'héritage
  - en l'absence d'extension, **extends Object est implicite**

`class CompteBancaire {...}`

équivalent à

`class CompteBancaire extends Object {...}`

- => tout objet est de type Object
  - seules les types primitifs (types numériques, caractères, et boolean) ne sont pas des objets

# La classe Object : méthodes

- => toute classe peut redéfinir les méthodes de Object :
  - `public boolean equals(Object obj)`
  - `public String toString()`
  - `protected Object clone() throws CloneNotSupportedException`
  - `public final Class getClass()`
  - `public int hashCode()`
  - et d'autres (`wait`, `notify`, `notifyall`)
- Ces méthodes ont une implémentation de base dans la classe Object
  - souvent pas suffisante pour les sous-classes : overriding

## equals()

- Implémentation dans la classe Object : égalité des références

```
public class A{  
    private int i;  
    private int j;  
    public A(int i,int j){this.i = i;  this.j = j;}  
}  
  
...  
A a = new A(1,2);  
A b = new A(1,2);  
A c = a;  
System.out.println(a == b); //false  
System.out.println(a.equals(b)); //false  
System.out.println(a == c); //true  
System.out.println(a.equals(c)); //true
```

## equals()

- Implémentation dans la classe Object : égalité des références
- En général on veut en revanche tester l'"égalité profonde"

```
public class A{  
    private int i;  
    private int j;  
    public A(int i,int j){this.i = i;  this.j = j;}  
    @Override  
    public boolean equals(Object o){  
        if (o == this) return true;  
        if (!(o instanceof A)) return false;  
        A a = (A)o;  
        return i == a.i && j == a.j;  
    }  
}
```

redéfinition de equals()

toujours  
présent

attention : paramètre de type Object, et non pas A, sinon overloading

# equals()

```
...
A a = new A(1,2);
A b = new A(1,2);
A c = a;
System.out.println(a == b); //false
System.out.println(a.equals(b)); //true
System.out.println(a == c); //true
System.out.println(a.equals(c)); //true
```

redéfinition de equals()

# Redéfinition de equals( ) : sous-classe

- Une sous-classe peut adopter le test d'égalité de sa super-classe
- Ou bien le redéfinir
- Dans le premier cas equals compare uniquement les parties super-classe des objets :

```
Classe CompteBancaire { ...
    //deux comptes sont identiques s'ils ont le même numéro
    public boolean equals( Object o ){
        if( o == this ) return true;
        if( !(o instanceof CompteBancaire) ) return false;
        CompteBancaire c = (CompteBancaire)o;
        return numero.equals( c.numero );
    }
}

Classe CompteCredit extends CompteBancaire { ...
    //le critère sur le numero s'applique aussi bien;
    //equals() n'est pas redéfini
}
```

# Redéfinition de equals( ) : sous-classe

- Si en revanche equals( ) doit être redéfini dans une sous-classe :
  - tester l'égalité des parties super-classe avec super.equals( )
  - ensuite tester l'égalité des champs propres de la sous-classe

Premier essai :

```
public class A{  
    private int i; private int j;...  
    public boolean equals(Object o) {...}  
}  
public class B extends A {  
    private int k;...  
    public boolean equals(Object o)  
    { if (! super.equals(o)) return false;  
        if( !(o instanceof B) ) return false;  
        B b = (B)o;  
        return b.k == k; }  
}
```

# Redéfinition de equals( ) : sous-classe

Une complication :

```
A a = new A(1,2);  
B b = new B(1,2,3);
```

- `b.equals(a)` renvoie false (a n'est pas une instance de B!)
- `equals()` doit être symétrique (requis par la spécification de Java)
- `a.equals(b)` doit renvoyer également false
- Mais ce n'est pas le cas avec :

```
public class A { ...  
public boolean equals(Object o){  
    if (o == this) return true;  
    if (!(o instanceof A)) return false;  
    A a = (A)o;  
    return i == a.i && j == a.j;  
}
```

```
}
```

b est une instance de A !

# Redéfinition de equals( ) : sous-classe

- Quand equals( ) est redéfini dans une sous-classe B de A, la classe A devrait tester égalité des types effectifs (au lieu de instanceof)

```
public class A{  
    private int i; private int j;...  
    public boolean equals(Object o){  
        if (o == this) return true;  
        if (o == null) return false;  
        if (getClass() != o.getClass()) return false;  
        A a = (A)o; return i == a.i && j == a.j;  
    }  
}  
public class B extends A {  
    private int k;...  
    public boolean equals(Object o){  
        if (! super.equals(o)) return false;  
        //égalité des types effectifs déjà testée  
        B b = (B)o; return b.k == k;  
    }  
}
```

## hashCode()

- Renvoie un code entier dérivé de l'objet (pour utilisation dans des tables hash)
- Implémentation par défaut dans la classe Object :
  - renvoie un entier dérivé de l'adresse interne de l'objet
- Si hashCode() est redéfinie, sa sémantique doit respecter :
  - si deux objets sont égaux selon equals(), leur hashCode() est le même
- Remarque : il n'est pas demandé que deux objets distincts aient different hashCode()
  - mais les performances des tables hash sont meilleures si on se rapproche de cette hypothèse

## hashCode()

```
A a=new A(1,2);
A b=new A(1,2);
A c=a;
System.out.println(a.hashCode()); //-> 26022015
System.out.println(b.hashCode()); //-> 3541984
System.out.println(c.hashCode()); //-> 26022015
```

- Remarque : a et b ont codes hash différents parce que les adresses des deux objets sont différent (hashCode() par défaut).
- Mais si a.equals(b) renvoie vrai, a et b devraient avoir le même hashCode!
- => Si une classe A redéfinit equals(), elle devrait également redéfinir hashCode()
  - si on compte l'utiliser dans des tables hash

# Redéfinition de hashCode( )

- Combiner les codes hash des champs qui déterminent l'égalité

```
public class A{  
    private int i; private int j;  
    @Override  
    public boolean equals(Object o){  
        if (o == this) return true;  
        if (o == null) return false;  
        if (getClass() != o.getClass()) return false;  
        A a = (A)o;  
        return i == a.i && j == a.j;}  
    @Override  
    public int hashCode() {  
        return Objects.hash(i, j);  
    }  
}
```

- La fonction statique hash de la classe `java.util.Objects` effectue une "bonne" combinaison des codes hash des ses arguments

# Redéfinition de hashCode( ) : sous-classe

```
public class B extends A {  
    private int k;  
    @Override  
    public boolean equals(Object o){  
        if (! super.equals(o)) return false;  
        B b = (B)o;  
        return b.k == k;  
    }  
    public int hashCode() {  
        return Objects.hash(super.hashCode(), k);  
    }  
}
```

# Redéfinition de hashCode()

```
public class CompteBancaire { ...
    public boolean equals( Object o ){
        if( !(o instanceof CompteBancaire) ) return false;
        if( o == this ) return true;
        CompteBancaire c = (CompteBancaire)o;
        return numero.equals( c.numero );
    }//equals
    public int hashCode() {
        return numero.hashCode();
        //hachage d'une unique valeur
    }
}
```

- Pour hacher un seul champ a :
  - si a est un objet a.hashCode()
  - si a est un tableau : Arrays.hashCode(a)
  - si a est de type primitif int : Integer.hashCode(a)
    - similaire pour les autre types primitifs

## toString()

- Fonction de conversion d'un objet en chaîne de caractères
- Implémentation par default dans la classe Object :

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

- Il est utile de la **redéfinir systématiquement** dans une classe
  - renvoyer une chaîne représentative de l'état de l'objet

```
public class CompteBancaire {  
    ...  
    public String toString(){  
        return getClass().getName() + String.format(  
            ": Compte=%s Solde=E %1.2f", numero, solde );  
    }  
}
```

# toString() pour une sous-classe

- Compléter super.toString()

```
public class CompteCredit {  
    ...  
    public String toString(){  
        return String.format( super.toString() +  
            " Credit=E %.2f Decouvert=E %.2f", credit,  
            solde >=0 ? 0 : -solde );  
    }  
}  
...  
CompteCredit cc =  
    new CompteCredit ("34F5T67", 1000, 200);  
cc.verser (500); cc.retirer (1650);  
System.out.println (cc.toString());  
// -> poo.banque.CompteCredit: Compte=34F5T67 Solde=E  
-150.00 Credit=E 200.00 Decouvert=E 150.00
```

# toString() et casting implicite

- `toString()` est implicitement invoquée sur un objet quand une conversion en `String` est nécessaire

`System.out.println (cc);`

équivalent à

`System.out.println (cc.toString());`

`String s = cc + ".";`

équivalent à

`String s = cc.toString() + ".";`

## clone()

- Censé renvoyer une copie de l'objet
- Implémentation **par défaut** dans la classe Objet :
  - **copie superficielle** : la valeur de chaque champ est copiée dans le clone
- Toutefois **clone()** est **protected** dans la classe Object
  - => elle est présente dans chaque classe mais pas accessible en dehors de la classe elle-même
  - Raison :
    - l'implémentation par défaut de **clone()** est considérée pas suffisante pour une classe arbitraire
    - **protected** a le but de rendre **obligatoire la redéfinition (public) par les sous-classes**

# clone()

- Pour encore plus de protection, Java impose de “déclarer” explicitement qu’une classe redéfinit clone()
  - Sinon une erreur d’exécution se produit quand elle est invoquée

```
classe Point implements Cloneable {  
    private double x,y;  
  
    ...  
  
    public Point clone() throws CloneNotSupportedException  
    {...}  
}
```

- Cette erreur est une exception de type CloneNotSupportedException, qui arrête le programme si elle n'est pas gérée
- Cela explique pourquoi la signature de clone() doit indiquer qu'elle peut soulever cette exception à runtime

# Clonage par défaut (superficiel)

- Si on est content avec le clonage par défaut, on ne fait rien d'autre que l'invoquer :

```
classe Point implements Cloneable {  
    private double x,y;  
  
    ...  
  
    public Point clone() throws CloneNotSupportedException {  
        return (Point) super.clone();  
    }  
}
```

- Remarques
  - `clone()` de la classe `Object` est accessible à l'intérieur de la classe `Point`
  - `clone()` de la classe `Object` fait la copie superficielle de tous les champs (y compris `x` et `y`)
  - `super.clone()` retourne un `Object` => casting

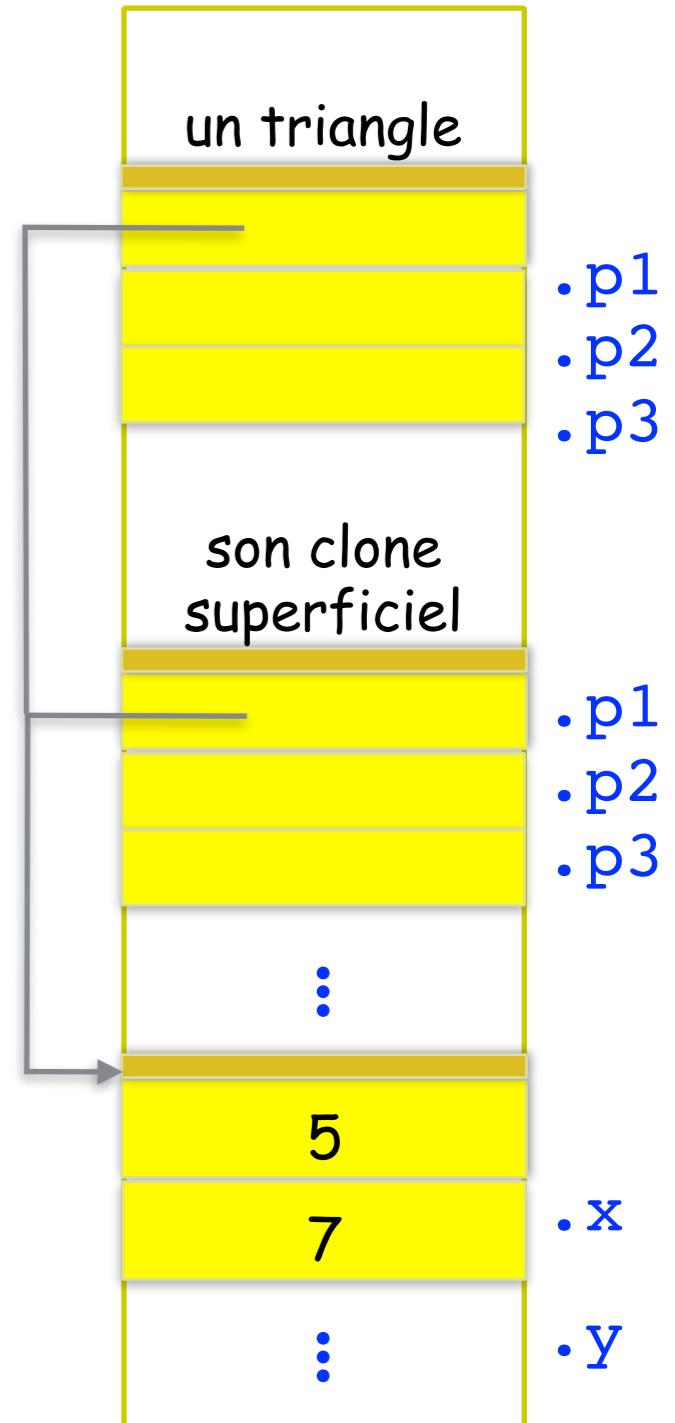
# Clonage par défaut (superficiel)

- Le plus souvent le clonage superficiel n'est pas suffisant

```
classe Triangle implements Cloneable {  
    private Point p1, p2, p3  
    ...  
    public Triangle clone()  
        throws CloneNotSupportedException {  
            return (Triangle) super.clone();  
    }  
}
```

- le clonage superficiel ferait en sorte qu'un triangle et son clone partagent leurs points

- La modification d'un triangle aurait comme conséquence la modification de son clone
  - => pas ce qu'on veut!



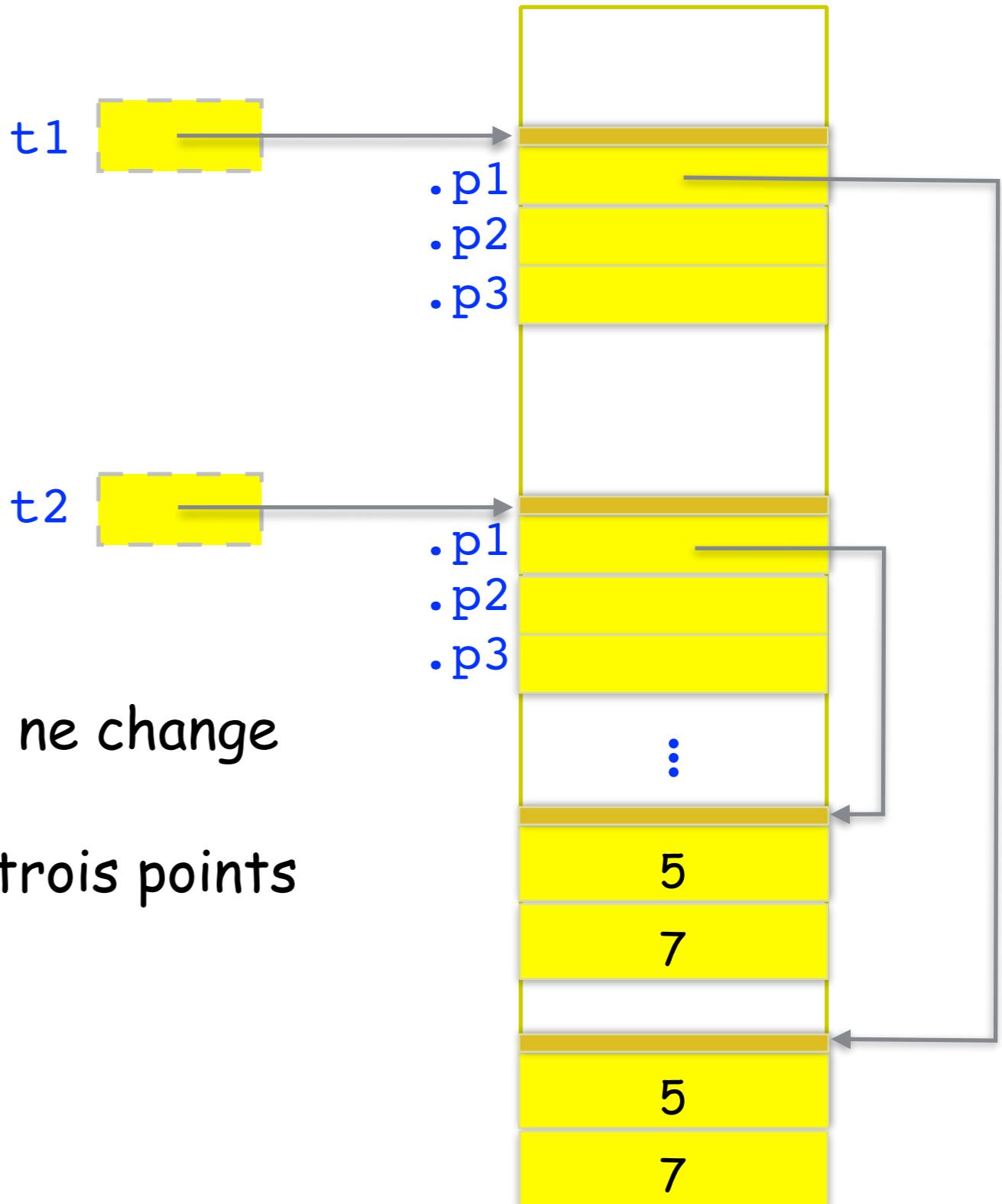
# Clonage profond

- Dans ces cas :
  - `clone()` doit être invoqué sur les champs de type non-primitif

```
classe Triangle implements Cloneable {  
    private Point p1, p2, p3;  
    ...  
    public Triangle clone() throws CloneNotSupportedException {  
        //pour cloner les champs de type primitif :  
        Triangle copie = (Triangle) super.clone();  
        // pour cloner les autres champs :  
        copie.p1 = p1.clone();  
        copie.p2 = p2.clone();  
        copie.p3 = p3.clone();  
        return copie;  
    }  
}
```

# Clonage profond

```
t1 = new Triangle(...);  
t2 = t1.clone();
```



- Si on modifie `t1`, son clone `t2` ne change pas => OK
- chacun a sa propre copie des trois points

# Clonage : remarques

- En implémentant Cloneable, clone( ) est possible pour la classe et les classes descendantes

```
class Point implements Cloneable {  
    ...  
}
```

```
class DataPoint extends Point{  
    double data;  
    ...  
    public DataPoint clone() throws CloneNotSupportedException  
    {return (DataPoint) super.clone();}  
}
```

- Ci-dessus super.clone() invoque la méthode clone de Object donc fait la copie superficielle de tous les champs, y compris les champs propres de la sous-classe (data)

# Clonage profond ou superficiel?

Il faut évaluer le cas par cas

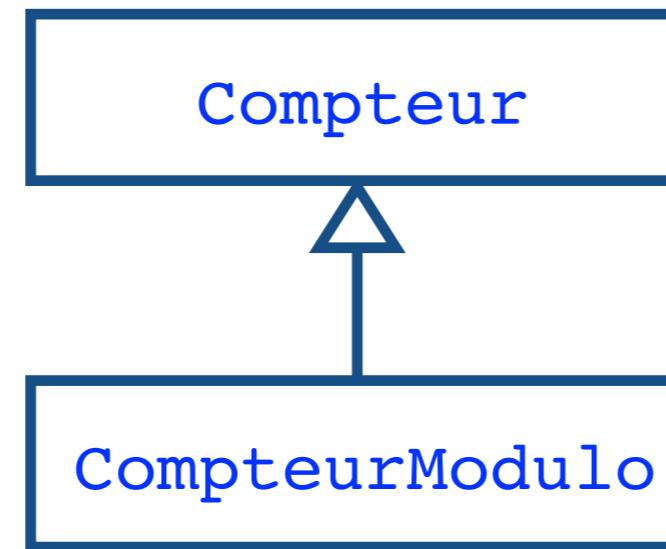
- Dans certains cas, l'effet de partager des références est voulu
- Le clonage superficiel des champs immuables est aussi en général OK
  - Exemple une Personne et son clone partagent leur nom (type String immutable)

# Clonage profond ou superficiel? Exemples

---

- `code/poo/clonage/StringList.java`
- `code/poo/clonage/IntegerStack.java`

# Un autre exemple de hiérarchie de classes



code/poo/compteurs/

# Relations entre les classes

- Il existe plusieurs relations possible entre les classes, les plus communes :
  - "est un" => Heritage
    - Un compte credit **est un** compte bancaire
    - Un compte gold **est un** compte credit
    - Un compteur modulo **est un** compteur
  - "a un", "a des" => Aggregation
    - une banque **a des** comptes bancaires
    - un triangle **a trois** points
  - "utilise" => Dependance
    - un achat **utilise** un compte bancaire
      - les méthodes de Achat manipulent des comptes bancaires
- **Attention** à ne pas utiliser de l'heritage dans tous les cas!

# Exercices

---

# Exercice 1

```
class Mere{  
    void f(int i){  
        System.out.println("f("+i+") de Mere");  
    }  
    void f(String st){  
        System.out.println("f('"+st+"'') de Mere");  
    }  
}
```

- Indiquer dans la classe Fille, quelles méthodes sont
  - additionnelles
  - redéfinitions (overriding)
  - surcharge (overloading)

# Exercice 1 (suite)

```
class Mere{  
    void f(int i){...}  
    void f(String st){...}  
}
```

```
class Fille extends Mere{  
    void g(){  
        System.out.println("g() de Fille");  
        f();  
        f(3);  
        f("bonjour");  
    }  
    void f(int i){  
        System.out.println("f("+i+") de Fille");  
    }  
  
    void f(){  
        System.out.println("f() de Fille");  
    }  
}
```

# Exercice 1 (suite)

```
class Mere{  
    void f(int i){...}  
    void f(String st){...}  
}
```

```
class Fille extends Mere{  
    void g(){ //additionnelle  
        System.out.println("g() de Fille");  
        f();  
        f(3);  
        f("bonjour");  
    }  
    void f(int i){  
        System.out.println("f("+i+") de Fille");  
    }  
  
    void f(){  
        System.out.println("f() de Fille");  
    }  
}
```

# Exercice 1 (suite)

```
class Mere{  
    void f(int i){...}  
    void f(String st){...}  
}
```

```
class Fille extends Mere{  
    void g(){ //additionnelle  
        System.out.println("g() de Fille");  
        f();  
        f(3);  
        f("bonjour");  
    }  
    void f(int i){//overriding  
        System.out.println("f("+i+") de Fille");  
    }  
  
    void f(){  
        System.out.println("f() de Fille");  
    }  
}
```

# Exercice 1 (suite)

```
class Mere{  
    void f(int i){...}  
    void f(String st){...}  
}
```

```
class Fille extends Mere{  
    void g(){ //additionnelle  
        System.out.println("g() de Fille");  
        f();  
        f(3);  
        f("bonjour");  
    }  
    void f(int i){//overriding  
        System.out.println("f("+i+") de Fille");  
    }  
  
    void f(){//overloading  
        System.out.println("f() de Fille");  
    }  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);  
    f.f(4);  
    m=f;  
    m.f(5);  
    ((Fille)m).g();  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);  
    m=f;  
    m.f(5);  
    ((Fille)m).g();  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);  
    ((Fille)m).g();  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);  
    ((Fille)m).g();  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();                         ▷ g() de Fille  
    f.g();  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();                         ▷ g() de Fille  
    f.g();                                  ▷ f() de Fille  
}
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere  
                                         ▷ g() de Fille
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille
```

# Exercice 1 (suite)

Prévoir la sortie du programme suivant :

```
public static void main(String[ ] args) {  
  
    Mere m = new Mere();  
    Fille f = new Fille();  
    m.f(3);                                ▷ f(3) de Mere  
    f.f(4);                                ▷ f(4) de Fille  
    m=f;  
    m.f(5);                                ▷ f(5) de Fille  
    ((Fille)m).g();  
    f.g();  
}  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere  
                                         ▷ g() de Fille  
                                         ▷ f() de Fille  
                                         ▷ f(3) de Fille  
                                         ▷ f('bonjour') de Mere
```

## Exercice 2

```
class A{
    public int i=4;
    public void f(){
        System.out.println("f() de A, i="+i);
    }
    public void g(){
        System.out.println("g() de A, i="+i);
    }
}
class B extends A{
    public int i=3;
    public void f(){
        System.out.println("f() de B, i="+i);
        g();
    }
}
```

## Exercice 2 (suite)

---

Prévoir la sortie du programme suivant :

```
A a=new B();  
a.f();
```

```
System.out.println("a.i="+a.i);  
System.out.println("((B) a).i="+((B)a).i);
```

## Exercice 2 (suite)

Prévoir la sortie du programme suivant :

```
A a=new B();
a.f();                                ▷ f() de B, i=3

System.out.println("a.i="+a.i);
System.out.println("((B) a).i="+( (B)a).i);
```

## Exercice 2 (suite)

Prévoir la sortie du programme suivant :

```
A a=new B();
a.f();
System.out.println("a.i="+a.i);
System.out.println("((B) a).i="+((B)a).i);
```

- ▶ f() de B, i=3
- ▶ g() de A, i=4

## Exercice 2 (suite)

Prévoir la sortie du programme suivant :

```
A a=new B();  
a.f();  
  
System.out.println("a.i="+a.i);  
System.out.println("((B) a).i="+((B)a).i);
```

- ▶ f() de B, i=3
- ▶ g() de A, i=4
- ▶ a.i=4

## Exercice 2 (suite)

Prévoir la sortie du programme suivant :

```
A a=new B();  
a.f();  
  
System.out.println("a.i="+a.i);  
System.out.println("((B) a).i="+((B)a).i);
```

- ▶ f() de B, i=3
- ▶ g() de A, i=4
- ▶ a.i=4
- ▶ ((B) a).i=3

# Exercice 3

---

## Exercice 3

```
class Base{
    protected String n = "Base";
    protected String nom(){return "Base";}
}
class Derive extends Base{
    protected String n = "Derive";
    protected String nom(){ return "Derive"; }
    protected void print(){
        Base mref = (Base) this;
        System.out.println(this.nom());
        System.out.println(this.n);
        System.out.println(mref.nom());
        System.out.println(mref.n);
        System.out.println(super.nom());
        System.out.println(super.n);
    }
}
```

## Exercice 3 (suite)

---

Prévoir la sortie du programme suivant

```
public static void main(String[ ] args) {  
    Derive d = new Derive();  
    d.print();  
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom());
    System.out.println(this.n);
    System.out.println(mref.nom());
    System.out.println(mref.n);
    System.out.println(super.nom());
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n);
    System.out.println(mref.nom());
    System.out.println(mref.n);
    System.out.println(super.nom());
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom());
    System.out.println(mref.n);
    System.out.println(super.nom());
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom()); //Derive
    System.out.println(mref.n);
    System.out.println(super.nom());
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom()); //Derive
    System.out.println(mref.n); //Base
    System.out.println(super.nom());
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom()); //Derive
    System.out.println(mref.n); //Base
    System.out.println(super.nom()); //Base
    System.out.println(super.n);
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom()); //Derive
    System.out.println(mref.n); //Base
    System.out.println(super.nom()); //Base
    System.out.println(super.n); //Base
}
```

## Exercice 3 : résultat

```
protected void print(){
    Base mref = (Base) this;
    System.out.println(this.nom()); //Derive
    System.out.println(this.n); //Derive
    System.out.println(mref.nom()); //Derive
    System.out.println(mref.n); //Base
    System.out.println(super.nom()); //Base
    System.out.println(super.n); //Base
}
```