Héritage

Virginia Aponte

Département Informatique
CNAM-Paris

9 septembre 2023

Qu'est-ce que l'héritage ?

- Définir classe/interface B par extension d'1 classe/interface A .
- `B extends A { ...composantes « propres » à B ...}`
- conséquences :
  - 1 instance de B possède :
    membres non privés de A + propres à B
  - A et B sont 2 classes ou 2 interfaces ;
    - `class B extends A { ...}` (A est une classe) ;
    - `interface B extends A { ...}` (A est une interface) ;
  - typage : +1 étage dans hiérarchie de types (A au-dessus de B) ;
  - typage : *B sous-type de A*.

Exemple : extension de `Compte` (pre-existante)

NOtez que `retrait` échoue si le solde est insuffisant.

```
class Compte{
   double solde;
   public Compte(double init){ solde = init;}
   public double getSolde() { return solde }
   public void  depot(double m) { solde = solde + m }
   public void  retrait(double m) {
      if (solde < m) throw new ProvisionInsuffisante();
      solde = solde - m ;
   }
}
```

Définir les comptes avec découvert

Les comptes avec découvert ont un découvert maximal autorisé.
Nous définirons `CompteDecouvert` par extension de `Compte`.

Les instances de `CompteDecouvert` devront :

- Être des Comptes :
  - on doit pouvoir obtenir le solde d'un compte avec découvert, réaliser un dépot dessus, etc.
- Auront des méthodes nouvelles :
  - p.e., pour fixer le découvert maximal autorisé.
- Auront des méthodes au comportément adapté :
  - on doit re-écrire dans B le code de `retrait` qui ne doit plus échouer si le solde est insuffisant (dans la limite du découvert autorisé).

Définir `CompteDecouvert` par extension de `Compte`

- + 1 variable d'instance nouvelle : `decMax`
- + 1 méthode nouvelle pour fixer sa valeur :
  `void setDecMax(double m)`
- + 1 constructeur propre :
  `CompteDecouvert(double init,double dmax)`
- + nouvelle version (redéfinie) de `retrait` : autorise un solde négatif dans la limite du découvert autorisé.

CompteDecouvert par extension de Compte

```
class CompteDecouvert extends Compte{
   double decMax;
   public void setDecMax(double m){decMax = m;}

   public CompteDecouvert(double m, double dm){
      super(m); decMax = dm;
   }

   /** Version rédefinie de retrait **/
   public void retrait(double m) {
      if (solde+decMax >= m)
                  solde = solde - m;
      else throw new provisionInsuffisante();
   }
}
```

Extension : B étend A

Syntaxe d'une extension :

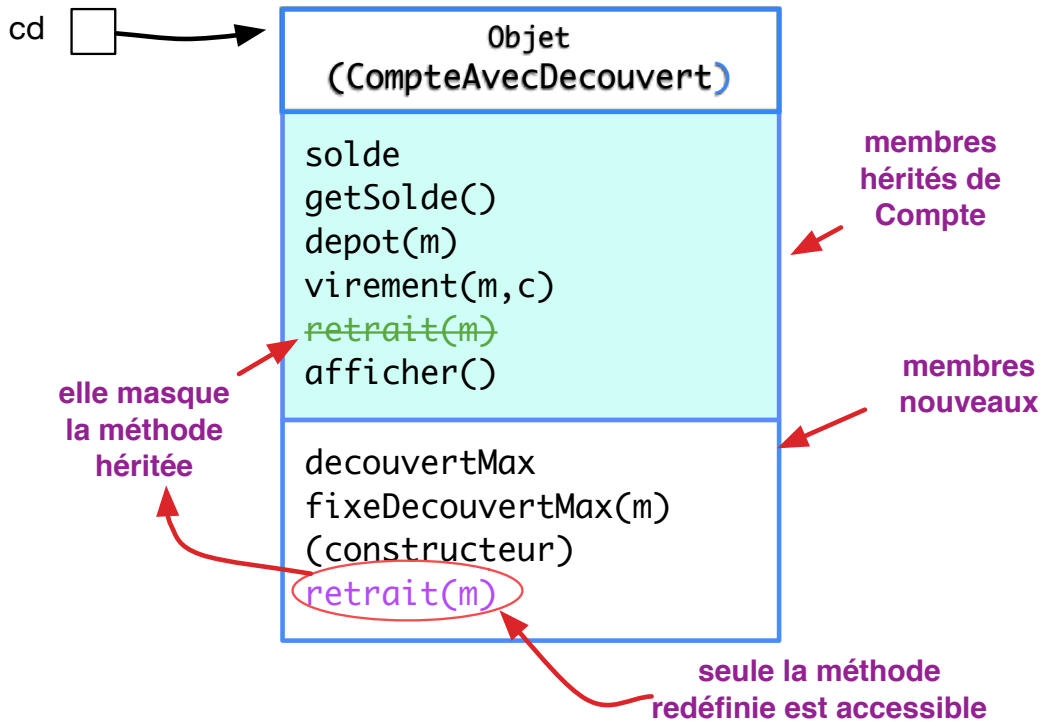
```
class B extends A { ... }
```

Dans la nouvelle classe B :

- on *hérite* des composantes non privés de A, **sauf** ses constructeurs. Ce qui est hérité est visible dans B.
- on peut déclarer des nouveaux attributs/méthodes.
- on peut *surcharger* des méthodes de A : si B les définit avec d'autres signatures.
- on peut *re-définir* (*override*) des méthodes de A. Il s'agit de re-écrire dans B le code d'une méthode de A en gardant la même signature.

Membres d'une instance de CompteDecouvert

```
class CompteDecouvert extends Compte { ... }
CompteDecouvert cd = new CompteDecouvert(...);
```



Attention : en réalité, l'objet contient «toutes» les variables d'instance + pointeurs vers méthodes classes et super-classes..

Sous-classes et super-classes

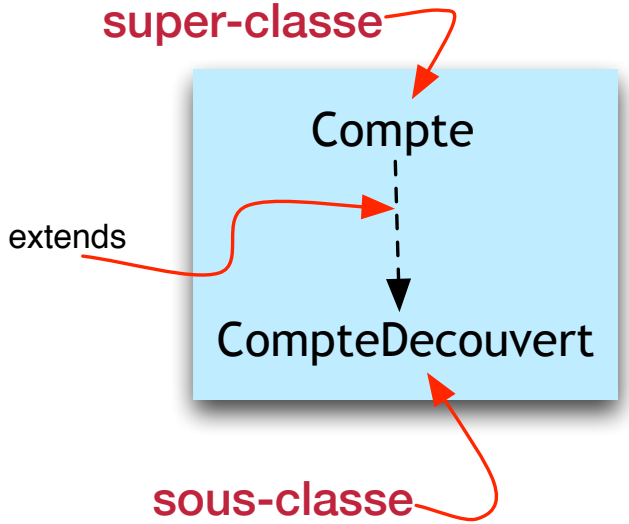
```
class CompteDecouvert extends Compte { ... }
```

- `CompteDecouvert` est une *sous-classe* de `Compte`.
- `Compte` est *la super-classe* de `CompteDecouvert`.
- Une classe peut être définie par extensions succésives. `CompteRemunereDecouvert` étend `CompteAveDecouvert` qui étend `Compte`.
- Elle aura une super-classe, une super-super-classe, etc...

Sous-classes et super-classes

```
class CompteDecouvert extends Compte {
  double decouvetMax;
  void fixeDecMax(double dm){…}
  void retrait(double m) throws..{…}
}
```

redéfinition (overriding)



L'instance de la sous-classe «à accéss/sait faire»

```
CompteDecouvert cd = new CompteDecouvert(...);
```

Que peut-on faire avec l'objet `cd` ?

- `cd` a accès à ses variables/méthodes propres :

```
cd.setDecMax(500);
```

- `cd` a accès aux variables/méthodes héritées non privées, en gardant le même comportement que dans la super-classe (si non rédéfinies) ;

```
cd.depot(50);
cd.getSolde();
```

- `cd` peut obtenir un comportément adapté à ce que modélise l'objet si appel à une méthode rédéfinie dans la sous-classe :

```
cd = new CompteDecouvert(800,1500); // 1500 déc.autorisé
cd.retrait(2000);   // quel comportement?
```

Pas d'échec ⇒ exécute méthode `retrait` redéfinie par sous-classe.

Appel à `retrait` avec comportement redéfini

Qu'affiche ce programme ?

```
CompteDecouvert d1 = new CompteDecouvert(6000,0);
try {  d1.retrait(7000.00);
       System.out.println("retrait OK" );
} catch (ProvisionInsuffisante e)  {
       System.out.println("Probleme 1er retrait");
}
d1.setDecMax(2000.00);
try { d1.retrait(7000.00);
      System.out.println("retrait OK"+
            "nouveau solde = " + d1.getSolde() );
} catch (ProvisionInsuffisante e) {
         System.out.println("Probleme 2eme retrait");};
```

redéfinition ≈ *overriding* (en anglais)

Autre exemple : `CompteRemunere`

```
class CompteRemunere extends Compte{
   private double taux ; // taux de remunération
   private double interets;

   public void setTaux(double m){this.taux = m; }

   public CompteRemunere (double m, double t){
      super(m) ; setTaux(t); interets = 0;
   }
   public void crediterInterets(){
      interets = getSolde()*taux/100;
      depot(interets);
   }
}
```

Notez que la méthode `crediterInterets()` invoque `depot()` heritée de `Compte`.

Héritage et constructeurs : super

```
class CompteDecouvert extends Compte{
   // constructeur
   public CompteDecouvert(double m, double dm){
      super(m);          // appel constructeur super-classe
      this.decMax = dm;  // initialisation var propres
}
  // Dans le main
   CompteDecouvert cd = new CompteDecouvert(50, 1500);
```

new CompteDecouvert(...) $\Rightarrow$ initialisation parties propre + héritée :

1. en 1er : invoquer le constructeur de la super-classe :
   - via (mot-clé **super**) pour initialiser partie héritée $\Rightarrow$ **super**(m);
   - en transmettant valeurs attendues (ici, m).
2. ensuite, initialiser partie prope $\Rightarrow$ **this**.decMax = dm;

Constructeurs implicites

```
class B extends A{
   public B(int i){
      //pas d'appel à super(), appel implicite inséré !
      this.x=i;
   }
}
```

Si super **n'est pas invoqué** dans le constructeur de B :

- le compilateur ajoute un appel implicite au constructeur sans arguments : **super**()
- attention : le constructeur sans argument doit être défini dans la super-classe A !

Ce code est correcte si A possède un contructeur sans arguments

Constructeurs implicites : exemple

```
class A {
  public A() { System.out.println("A");} }

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

Affichages de **new** B();

A
B

Où y t-il des appels implicites à `super()` ?

Héritage et visibilité

```
class Compteur {
   private int x;
   public void set(int i) { this.x = i;}
   public int get(){ return this.x;}
   public void incr() { this.x = this.x + 1;}
}
class CompteurPas extends Compteur{
   private pas;
   public CompteurPas(int p){this.pas = p;}
   public void incr() { // redefinition
     this.x= this.x + this.pas; // erreur: x non visible!!
   }
}
```

- Membres privés ⇒ non visibles dans les sous-classes ;
- les rendre visibles dans sous-classes (et pas ailleurs) ⇒ **protected**

Solution avec protected

```
class Compteur {
   protected int x;
   public void set(int i) { this.x = i;}
   public int get(){ return this.x;}
   public void incr() { this.x = this.x + 1;}
}
class CompteurPas extends Compteur{
   private pas;
   public CompteurPas(int p){this.pas = p;}
   public void incr() { // redefinition
     this.x= this.x + this.pas; //ok.
   }
}
```

- x est maintenant visible dans les sous-classes de Compteur.
- Y a-t-il une solution en gardant x privé ?

Solution en gardant x privé

```
class Compteur {
   private int x;
   public void set(int i) { this.x = i;}
   public int get(){ return this.x;}
   public void incr() { this.x = this.x + 1;}
}
class CompteurPas extends Compteur{
   private pas;
   public CompteurPas(int p){this.pas = p;}
   public void incr() { // redefinition
     this.set(this.get()+ this.pas);
   }
}
```

- x est visible pour les méthodes héritées `set` et `get` ;
- qui sont suffisantes pour implanter le comportement re-défini.

La variable super : accès à la super-classe

super.m() : accès au membre m (*non privé*) de la super-classe

```
class CompteurPas extends Compteur{
  private int pas;
  public void incr() { // redefinition
    for (int i=1; i<= this.pas; i++){
        super.incr(); // methode de la super-classe
    }
  }
```

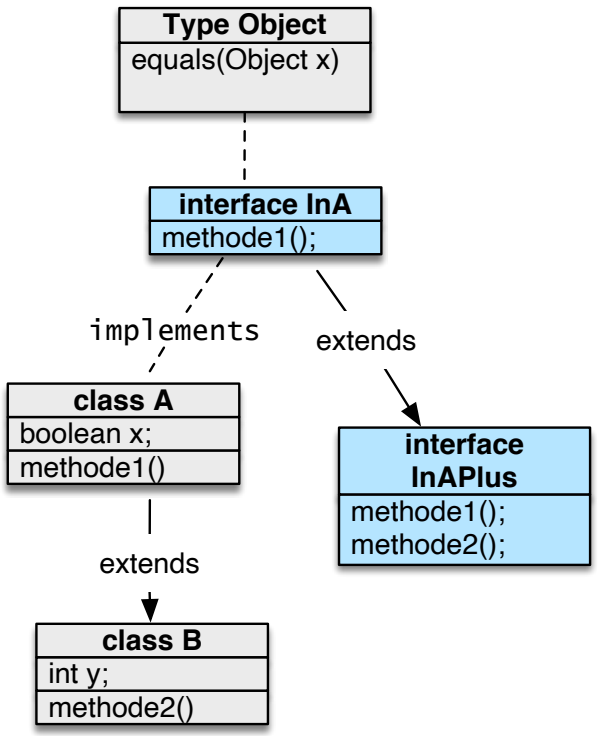
- Utile pour accèder aux membres redéfinis dans la sous-classe.
- Ici, incr() est redéfinie $\Rightarrow$ **super**.incr() est la méthode de la super-classe.

Attention : ce n'est pas la meilleure implantation...

Héritage et typage

Hiérarchie de types objets en Java

Les types objets en Java sont organisés en une hierarchie avec `Object` à sa racine.
Nous étudions ici le sens et l'utilisation de cette hiérarchie. Concept clé : polymorphisme objet. Plus un type sera bas, plus il contiendra des méthodes parmi celles de ses antécedents dans la hiérarchie.



Trois sorte de types d'objets en Java

- les interfaces (profils de méthodes)
- les classes (attributs + profils de méthodes + code)
- les classes abstraites (mélange des 2)

$\Rightarrow$ spécifients les méthodes sont accessibles sur leurs instances.

Héritage en Java

- Définition d'un type B par extension d'un type A pré-existant.
- + 1 étage dans la hiérachié : B est mis juste au-dessous de A.
- B est un *sous-type* de A (B est plus bas que A)
- A est un *super-type* de B (A est plus haut que B).

Deux mécanismes d'extension :

1. via `extends` :
   - entre classes : `class C2 extends C1`
   - entre interfaces : `interface I2 extends I1`
2. via `implements` : une classe ajoute du code et d'autres méthodes p/r à l'interface implantée
   - `class C implements I`

B Hérite de A

```
public class A {
  public int f1 (int x)throws MyException { ... }
}
public class B extends A { ... }
```

- signification comportemental
  - toute méthode publique appelable sur A existe aussi sur B
  - appelable *au moins* dans les même cas que A
  - constructeurs et méthodes privées $\neq$

a. voir la généricité pour un assouplissement de cette règle

B Hérite de A

```
public class A {
  public int f1 (int x)throws MyException { ... }
}
public class B extends A { ... }
```

- signification comportemental
  - toute méthode publique appelable sur A existe aussi sur B
  - appelable *au moins* dans les même cas que A
  - constructeurs et méthodes privées $\neq$
- garanti par construction par le compilateur Java :

a. voir la généricité pour un assouplissement de cette règle

B Hérite de A

```
public class A {
  public int f1 (int x)throws MyException { ... }
}
public class B extends A { ... }
```

- signification comportemental
  - toute méthode publique appelable sur A existe aussi sur B
  - appelable *au moins* dans les même cas que A
  - constructeurs et méthodes privées $\neq$
- garanti par construction par le compilateur Java :
  - B hérite automatiquement de toutes les données non privées de A
  - méthodes héritées par défaut $\Rightarrow$ même code

a. voir la généricité pour un assouplissement de cette règle

B Hérite de A

```
public class A {
  public int f1 (int x)throws MyException { ... }
}
public class B extends A { ... }
```

- signification comportemental
  - toute méthode publique appelable sur A existe aussi sur B
  - appelable *au moins* dans les même cas que A
  - constructeurs et méthodes privées $\neq$
- garanti par construction par le compilateur Java :
  - B hérite automatiquement de toutes les données non privées de A
  - méthodes héritées par défaut $\Rightarrow$ même code
  - si redéfinition compilateur exige signature [a] *identique*
  - possiblement des **throws** en moins

a. voir la généricité pour un assouplissement de cette règle

B Hérite de A

```
public class A {
  public int f1 (int x)throws MyException { ... }
}
public class B extends A { ... }
```

- signification comportemental
  - toute méthode publique appelable sur A existe aussi sur B
  - appelable *au moins* dans les même cas que A
  - constructeurs et méthodes privées $\neq$
- garanti par construction par le compilateur Java :
  - B hérite automatiquement de toutes les données non privées de A
  - méthodes héritées par défaut $\Rightarrow$ même code
  - si redéfinition compilateur exige signature [a] *identique*
  - possiblement des **throws** en moins
- B possiblement **étendue** : méthodes et données supplémentaires

a. voir la généricité pour un assouplissement de cette règle

Le typage : une analyse du compilateur

Typage =

- Analyse du compilateur sur la nature des données et des appels
- But : garantie $\Rightarrow$ le code compilé ne produit pas d'erreur à l'exécution due à une incompatibilité de types
- Exemple d'erreur si on exécutait sans faire d'analyse :
  - Appliquer `getSolde()` sur l'objet "Abc" $\Rightarrow$ lève `NoSuchMethodException`.

Le typage : monomorphe ou polymorphe ?

Typage Monomorphe :

- chaque expression du programme : un seul type possible
- garantie facile à établir

Typage Polymorphe :

- chaque expression du programme : plusieurs types possibles
- même garantie
- code plus générique

Héritage et Polymorphisme OO

En induisant une hiérarchie par extension de types, l'héritage permet le polymorphisme.

- Le typage OO est polymorphe, i.e., tout objet a plusieurs types
- « b a le type B » se dit « b *est-un* B »
  - si b est une instance de B, alors b *est-un* B
  - également, b *est-un* T pour tout T super-type de B (plus haut que B)
  - on parle de « compatibilité ascendante » de b avec tous ses super-types
- Ex : si B hérite de A $\Rightarrow$ alors b aura au moins deux types :
  - b *est-un* B
  - b *est-un* A, etc

Exemple avec `CompteRemunere`

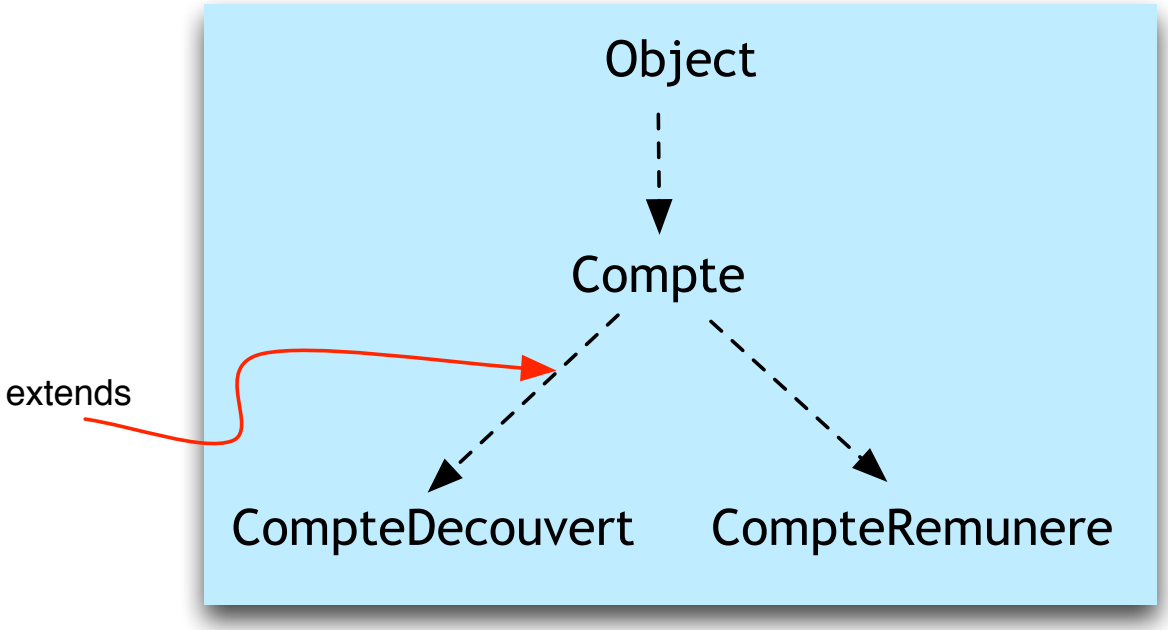
```
class CompteRemunere extends Compte{
   private double taux ; // taux de remunération
   private double interets;

   public void setTaux(double m){this.taux = m; }

   public CompteRemunere (double m, double t){
      super(m) ; setTaux(t); interets = 0;
   }
   public void crediterInterets(){
      interets = getSolde()*taux/100;
      depot(interets);
   }
}
```

Exemple : la sous-hiérarchie de `Compte`

```
class CompteDecouvert extends Compte {...}
class CompteRemunere extends Compte {...}
CompteRemunere cr = new CompteRemunere(..);
```



- cr *est-un* `CompteRemunere`
- et cr *est-un* `Compte`.

Règles de compatibilité pour l'affectation

Affectation

Une variable de type déclaré A, peut recevoir un objet qui *est-un* A.
Ex : une instance de A ou d'un type plus bas que A dans la hiérarchie.

```
Compte a = new CompteDecouvert(100, 1500);
```

super-type de CompteDecouvert

sous-type de Compte

- a déclarée de type `Compte` peut recevoir un objet qui *est-un* `Compte`.
- et `CompteRemunere` *est-un* `Compte`

Règle de compatibilité (appel de méthode)

Appel de méthode avec argument

Une méthode

- dont un paramètre est déclaré de type A
- peut être invoquée avec un paramètre :

$$\text{soit} \begin{cases} \text{instance de A} \\ \text{d'un sous-type de A (plus bas que)} \end{cases}$$

```
static void afficheSolde(Compte c){  //type déclaré
  System.out.println("Solde :" + c.getSolde());
}
....
CompteDecouvert cd = new CompteDecouvert (100, 1500);
afficheSolde (cd);          // invocation avec sous-type
```

A quoi sert le sous-typage ?

Le sous-typage est une forme de *polymorphisme*

- polymorphisme $\approx$ plusieurs formes ;
- ce qui «change de forme» est le type pointé par les variables,
- autrement dit, une variable de type déclaré A, pourra pointer vers des objets instances différents de A :
  - affectation de «plusieurs formes» d'objet ;
  - stocker dans même structure (tableau, etc.) objets instances divers mais de type déclaré commun !
  - passer en argument objets instances de B pour une méthode qui «attend» un super-type déclaré A ;

Utilité du sous-typage : exemple (1)

```
public double bilan(ArrayList<Compte> lc){
    double res = 0;
    for (int i=0; i < lc.size(); i++) {
        res= res+ lc.get(i).getSolde();
    }
    return res;
}
```

- `bilan()` : déclare une liste d'éléments de type `Compte`
- mais accepte une liste d'éléments sous-types de Compte.
- pourquoi ? : tous les éléments de la liste à passer possederont forcément une méthode `getSolde()`
- pourquoi ? : car tout sous-type de `Compte` hérite cette méthode.

Utilité du sous-typage : exemple (suite)

```
Compte c = new Compte(...);
CompteDecouvert cd = new CompteDecouvert(...);
Compte cr= new CompteRemunere(...);
ArrayList<Compte> lc = new ArrayList<Compte>();
lc.add(c); lc.add(cd);
lc.add(cr);      // TDs cases: Compte, CompteDecouvert, Comp
double b = bilan(lc);
```

- les «cases» de `lc` $\Rightarrow$ de type declaré `Compte` ;
- elles pointent sur des instances de type divers,
  - **tous** sous-types de `Compte`, donc tous possédant `getSolde()`.
- très utile de mélanger divers types de comptes dans une même structure !

Utilité du sous-typage : exemple (fin)

Peut-on écrire ?

```
public void setDecouvertMax(double d, ArrayList<Compte> lc)
    for (int i=0; i < lc.size(); i++) {
        lc.get(i).setdecMax(d);
    }
}
```

- Cette méthode ne compile pas !
- les «cases» de `lc` $\Rightarrow$ sont de déclarés de type `Compte` ;
- dans `Compte` $\Rightarrow$ pas de méthode `setDecMax()`
- la règle de sous-typage rejette cette méthode.
- autrement, on aurait pu lui passer une liste composée d'objets sans `setDecMax()`, ce qui fait aurait échouer l'exécution.

3. Sous-typage et oubli

Sous-typage ⇒ oubli

```
  Compte c = new Compte(...);
  CompteDecouvert cd = new CompteDecouvert(...);
  cd.setDecMax(1500);   // (1) ok
  c = cd;               // (2) ok
  c.setDecMax(1500);    // (3) erreur!
}
```

Le compilateur connaît uniquement les types déclarés :

- (1) ⇒ ok car le type déclaré de cd est `CompteDecouvert` et ce type contient `setDecMax()`.
- (2) ok, car compatibilité par sous-typage.
- le type déclaré de c est Compte :
  - (3) est une tentative d'accès à d'autres méthodes que celles du déclaré de c ⇒ rejetée à la compilation ;

Sous-typage $\Rightarrow$ oubli (2)

Conclusion

- Une variable qui pointe vers une instance qui contient plus de méthodes que son type déclaré, ne peut pas les employer, car le typage (compilateur) l'interdit.

- Soustypage $\approx$ dans le code d'utilisation de l'objet, on «oublie» les méthodes de l'objet qui sont «en plus» p/r à son type déclaré.