

# Héritage, sous-typage et classes abstraites

Virginia Aponte

Département Informatique  
CNAM-Paris

September 14, 2017

# 1. Sous-typage (polymorphisme objet)

# Types d'objets en Java

Trois sortes de **types** pour les objets:

- les interfaces
- les classes
- les classes abstraites.

Leur (seul) point commun:

- spécifier les méthodes (avec ou sans corps),
- que «contiennent» les objets de ce type.

## sous-typage

Permet de comparer les types selon les méthodes que ces types «contiennent».

# Sous-typage: qu'es-ce que c'est?

## Sous-typage

- Le `type B` est un **sous-type** du `type A` si
  - ▶ B «peut exécuter» tout ce que A «peut»  
(autrement dit: *si B contient «autant de méthodes» que A*)
- On dit: *A est le **super-type** de B.*
- Appelé aussi: *«**polymorphisme objet**»*

Exemple : une **classe** qui implante une **interface**  
est un sous-type de l'interface.

# Exemple de sous-typage

Aire: surface d'une figure géométrique.

---

```
interface Aire{  
    public double getAire ();  
}  
public class Cercle implements Aire {  
    private int radius;  
    private Point p;  
    ....  
    public Double getAire() {  
        return 3.14159 * radius * radius;  
    }  
}
```

---

La classe Cercle est un sous-type de l'interface Aire

## Point n'est pas sous-type de Aire

```
public class Point implements Deplacable {
    private int x;
    private int y;
    public Point(int initx , inity ){...}
    public int getX() {...}
    public int getY() {...}
    public void move(int dx, int dy) {...}
}
```

- La classe `Point` ne possède pas de méthode `getAire()`
- `Point` ne «contient pas» autant que `Aire`;
- donc, `Point` n'est pas sous-type de `Aire`.

# Compatibilité par sous-typage

Une *variable* déclarée de type A, peut stocker un objet *sous-type* de A.

```
Aire a = new Cercle (...);
```

super-type de Cercle

sous-type de Aire

## Compatibilité par sous-typage (2)

```
Aire a = new Cercle (...);
```

super-type de Cercle      sous-type de Aire

Méthode avec **paramètres déclarés** de type A  $\Rightarrow$  peut être invoquée

avec **paramètres effectifs** :  $\left\{ \begin{array}{l} \text{de type A} \\ \text{d'un sous-type de A} \end{array} \right.$  **ou**

```
static double deuxFois(Aire x){ //param. declare.5: Aire
    return (x.getAire() * 2);
}
....
Circle c = new Cercle (...);
deuxFois(c); // invocation sur sous-type
```



# Sous-typage entre classes et interfaces

Attention: en Java on doit déclarer explicitement les relations de sous-typage (via **implements** ou **extends**)

Rappel: une classe peut implanter plusieurs interfaces.

Toutes ces classes sont déplaçables, mais Point n'a pas d'aire.

---

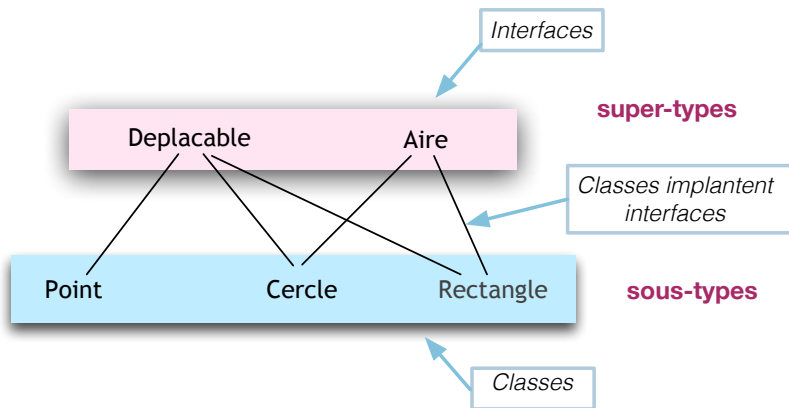
```
public class Point implements  
    Déplacable { ...}
```

```
public class Cercle implements  
    Déplacable , Aire { ...}
```

```
public class Rectangle implements  
    Déplacable , Aire { ...}
```

---

# Hiérarchie de sous-typage entre classes et interfaces



## 2. Types statiques et types dynamiques

# Type statique (ne change pas)

- Le type **statique** d'une variable, est celui *donné à sa déclaration*:

```
Deplacable x = new Point(1,2);
```

type statique

type dynamique

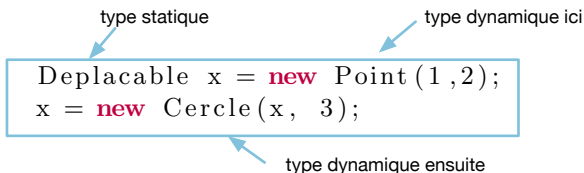
Ici, **Deplacable** est le type statique de x.

- ▶ c'est aussi son type **connu** à la compilation;
- ▶ il ne **change jamais** pendant l'exécution du programme

statique  $\approx$  ne change pas .

# Type dynamique (peut changer)

- Le type **dynamique** d'une variable, est celui de la classe utilisée pour construire *l'objet référencé par la variable à un point précis de l'exécution*:



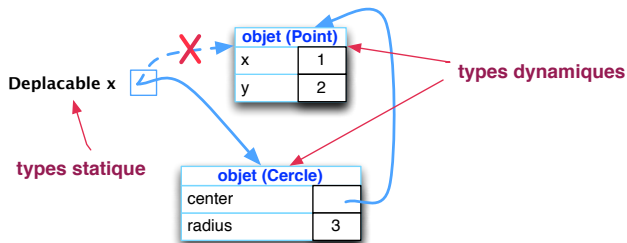
x a les types dynamiques **Point** puis **Cercle**.

- ▶ il n'est pas forcément **connu** à la compilation;
- ▶ il peut **changer** pendant l'exécution du programme

dynamique  $\approx$  changeant.

# Type dynamique et mémoire

```
Deplacable x = new Point(1,2);  
x = new Cercle(x, 3);
```



- type dynamique  $\Rightarrow$  structure précise de l'objet pointé;
- type statique  $\Rightarrow$  donne un «point de vue» de l'objet pointé.

# Type statique vs. type dynamique

Pour toute variable  $x$ :

son type dynamique est **toujours un sous-type** de son type statique.

Rappel:  $B$  est un sous-type de  $A$ , si  $B$  «sait faire au moins autant» que  $A$ .

Que signifie «faire autant» ?  $\approx$  *avoir au moins autant de méthodes que.*

## Conclusion:

- l'objet pointé ( $\approx$  type dynamique)
- «sait toujours faire autant que» ( $\approx$  est un sous-type de)
- le type déclaré de la variable ( $\approx$  type statique).

## 2.1 Utilité du sous-typage



# A quoi sert le sous-typage?

Ou «pourquoi diable se compliquer la vie?»

Le sous-typage est une forme de *polymorphisme*

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

# Compatibilité entre types statique et dynamique

Le sous-typage est utile car il autorise la **compatibilité** («mélange») de types différents, sans risque d'erreurs à l'exécution.

## Compatibilité par sous-typage entre types statique et dynamique

- une variable de type statique  $A$ ,
- peut pointer vers un objet de type  $B$ ,
- si  $B$  est un sous-type de  $A$ .

Cela arrive en 3 situations:

- affectation variable  $A \ x = o$ ; avec  $o$  de type  $B$ ;
- affectation «case» d'une structure de données  $A \ [] \ t$ ;  $..t[0] = o$ ;
- appel à une méthode  $m(A \ x)$  avec argument de type  $B \Rightarrow m(o)$ ;

# Utilité du sous-typage: exemple (1)

- `moveItAll` «attend» arraylist d'objets
- ces objets ont le type (statique) `Deplacable`

---

```
public void moveItAll (ArrayList<Deplacable> s,  
                      int dx, int dy) {  
    for (int i=0; i < s.size(); i++) {  
        s.get(i).move(dx,dy);  
    }  
}
```

---

`moveItAll` accepte arraylist avec objets **sous-types** de `Deplacable`.

## Utilité du sous-typage: exemple (2)

Variable `d`  $\Rightarrow$  de type statique (TS) `ArrayList<Deplacable>`:

---

```
Deplacable p =                //TS=Deplacable , TD=Point
    new Point(5,5);
Cercle c =                    //TS=Cercle ,      TD=Cercle
    new Cercle(new Point(..),100);
ArrayList<Deplacable> d = new ArrayList<Deplacable>();
d.add(p);
d.add(c);                      // TDs : Point et Cercle
moveltAll(d,5,10);
```

---

- les «cases» de `d`  $\Rightarrow$  de TS `Deplacable`;
- elles pointent sur objets de TDs divers,
  - ▶ **tous** sous-types de `Deplacable`.

### 3. Sous-typage et oubli

## Sous-typage $\Rightarrow$ oubli

```
Point p = new Point(5,5);
Cercle c = new Cercle(new Point(0,0),100);
ArrayList<Deplacable> d =
    new ArrayList<Deplacable>();
d.add(p); d.add(c); // ok
c.getRadius(); // (1) ok
d.get(1).getRadius(); // (2) erreur!
}
```

- Compilation: connaît **uniquement** les types statiques
  - ▶ (1)  $\Rightarrow$  ok car TS = Cercle, qui contient `getRadius()`.
  - ▶ Tentative d'accès à d'autres méthodes que celles du TS  $\Rightarrow$  **rejetée à la compilation**;
  - ▶ (2)  $\Rightarrow$  rejeté par le compilateur.

### Conclusion

- Un objet dont le type dynamique contient plus de méthodes que son type statique, **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 statique.

## 5. Héritage



# Qu'est-ce que l'héritage?

## Héritage

- Définition d'une **nouvelle** classe ou interface **B** par **extension** d'une classe ou interface **A existante**:
- Syntaxe `B extends A` (à la déclaration)
- Sémantique:
  - ▶ B « hérite » des membres (non privés) de A;
  - ▶ +1 étage dans **hiérarchie de types** (A au-dessus de B);
  - ▶ B devient (en général) un **sous-type** de A.

# Extension de la classe Compte

Rappel:

---

```
class Compte{
    double solde;
    Compte(double init){...}
    double getSolde() {... }
    void depot(double m) {... }
    void retrait(double m) throws ProvisionInsuffisante {
        if (solde < m){ throw new ProvisionInsuffisante ();
        } else {solde = solde - m ;} }
    void virement(Compte c, double m) throws ProvisionInsuffisante
    void afficher(){ System.out.println("Solde_courant:_solde")
}
}
```

---

retrait et virement **échouent** si le solde est insuffisant.

# Compte avec découvert ou rémunéré

2 nouvelles classes: `CompteDecouvert` et `CompteRemunere`.

- **Ce sont des comptes**: doivent partager les caractéristiques des `Compte`.
- **Fonctionnalités nouvelles**: rémunérer un compte, fixer le taux de rémunération, fixer le seuil du découvert autorisé.
- **Comportements à redéfinir**: on doit adapter le comportement de certaines méthodes. Ex: la méthode retrait ne doit plus échouer si le solde est insuffisant (dans la limite du découvert autorisé).

# Définir CompteDecouvert

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

```
class CompteDecouvert extends Compte{
    private double decMax;

    public void fixeDecMax(double m){decMax = m;}

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

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

---

# Sous-classes et super-classes

---

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

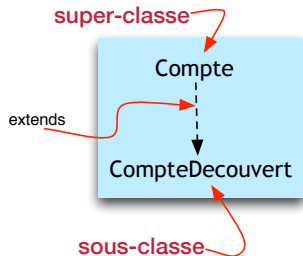
---

- `CompteDecouvert` est une *sous-classe* de `Compte`.
- `Compte` est *la super-classe* de `CompteDecouvert`.
- Une classe peut être définie par extensions successives.  
`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 decMax;  
    void fixeDecMax(double dm){...}  
    void retrait(double m) throws..{...}  
}
```

redéfinition (overriding)



# Héritage entre classes

---

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

---

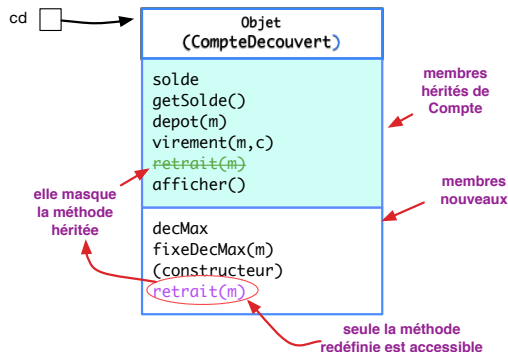
La nouvelle classe B:

- *hérite* variables + méthodes **non privés** de A, **sauf** ses constructeurs.
- peut **déclarer nouvelles variables/méthodes** qui lui sont *propres*.
- peut *surcharger* méthodes de A: si elle les définit avec d'autres signatures.
- peut *re-définir* méthodes de A (*override*).



# Membres de la sous-classe

```
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..*

# La sous-classe «peut accéder/sait faire»

---

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

---

- variables/méthodes **propres** à la sous-classe:

```
cd.fixeDecMax(500);
```

- variables/méthodes **héritées** avec **même comportement** que dans la super-classe, si non redéfinies dans la sous-classe;

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

- méthodes avec **comportement redéfini** dans la sous-classe:

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

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

# retrait: 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.fixeDecMax(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*  $\approx$  *overriding* (en anglais)

## 5.1 Héritage et typage

# Héritage et sous-typage

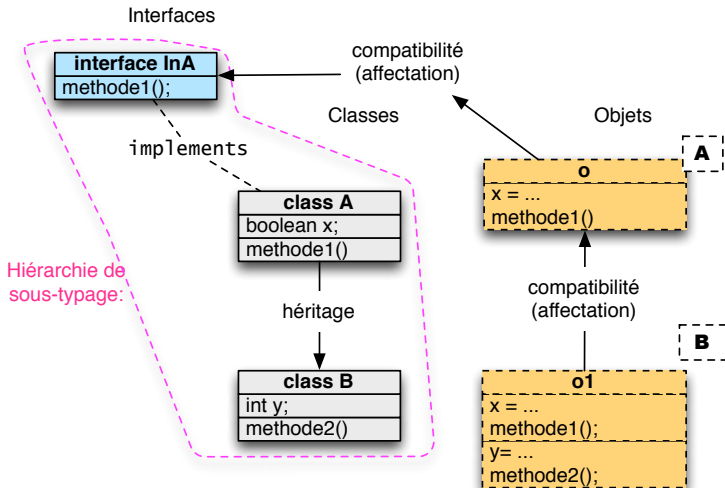
---

```
class B extends A { ... } // héritage entre classes
interface I { ... }
class C implements I { ... // implantation d'une interface
```

---

- une sous-classe B est un sous-type de sa super-classe A
- une classe C est un sous-type d'une interfaces I qu'elle implante.
- ces relations sont **transitives**:
  - ▶ si C sous-type de B et B sous-type de A  $\Rightarrow$  C sous-type de A;
  - ▶ si C sous-classe de B qui implante I  $\Rightarrow$  C sous-type de I; etc.

# extends, implements et compatibilité



B est un **sous-type** de A et de InA  
car B contient au moins tout ce que A et InA contiennent

# Rappel: compatibilité par sous-typage

Si B est un sous-type de A, alors tout objet instancé de B peut être:

- "mis" dans une variable ou case de type statique A;
- passé à une méthode dont l'argument attendu est de type A;

Le code suivant est donc correcte:

---

```
Compte c1 = new CompteDecouvert (...);  
Compte c2 = new Compte (...);  
CompteRemunere cr = new CompteRemunere (...);  
c1 = cr;  
c1.retrait(50);  
c.virement(100, c2);
```

---

Car `CompteRemunere` est un sous-type de `Compte`.

## Exemple: ce code est-il correcte?

---

```
Compte c = new Compte (...);
CompteRemunere cr = new CompteRemunere (...);
CompteDecouvert cd = new CompteDecouvert (...);
cd = cr;
c = cd;
cd.virement(100, cr);
cd.retrait(50);
```

---

Quelles sont les lignes érronées?

Quel sorte d'erreur?

**Transitivité:** si on a `class CompteRemunereAvecDecouvert extends CompteRemunere ...`



## 5.2 Héritage et constructeurs : super.

# Construction d'instance d'une sous-classe

```
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(...) ⇒ initialisation parties propre + héritée:

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

# Constructeurs implicites

```
class B extends A{
    public B(int i){
        // pas d'appel à super() ici --> 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 constructeur 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()` ?

## 5.2 Héritage et visibilité: `protected`, `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  $\Rightarrow$  **non visibles** dans les sous-classes;
- les rendre visibles dans sous-classes (et pas ailleurs)  $\Rightarrow$  **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 accéder aux membres redéfinis dans la sous-classe.
- Ici, incr() est redéfinie ⇒ **super.incr()** est la méthode de la super-classe.

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

- 2 nouvelles variables d'instance: `taux` et `interets`
- 2 nouvelles méthodes:
  - ▶ `fixeTaux(double m)`
  - ▶ `et accrediterInterets()`
- 1 constructeur propre.

---

```
class CompteRemunere extends Compte{
    private double taux ;
    private double interets;

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

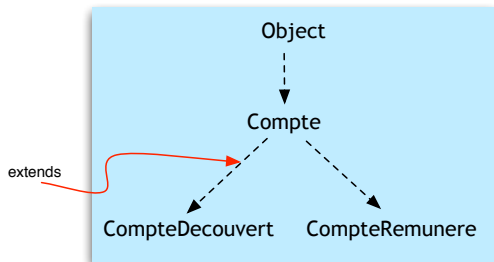
    public CompteRemunere (double m, double t){
        super(m) ; fixeTaux(t); interets = 0;
    }
    public void crediterInterets (){
        interets = getSolde()*taux/100;
        depot(interets);
    }
}
```

---

## 5.3 Hiérarchie de classes : `Object`, méthodes prédéfinies.

# Hiérarchie des Comptes et le type Object

En Java toute classe **hérite implicitement** du type Object:



- à la racine de toutes les hiérarchies de classes!
- possède méthodes prédéfinies: `equals`, `toString`, etc
- $\Rightarrow$  héritées par tout objet de toute classe!
- leur code est donné par défaut: il faut souvent le re-définir. Ex:  
`o1.equals(o2)` teste l'égalité d'adresses de `o1` et `o2`!

# Héritage simple (entre classes)

Peut-on définir `CompteRemunereDecouvert` par héritage simultané de `CompteRemunere` et `CompteDecouvert` ? (héritage multiple)

---

```
class CompteRemunereDecouvert
    extends CompteRemunere, CompteDecouvert ????
```

---

Réponse  $\Rightarrow$  Non!

## Héritage entre classes : simple

En Java, seul l'héritage **simple** entre classes est autorisé.

Mais, l'**implantation de multiples interfaces** est possible...

## 6. Liaison dynamique

# De quoi s'agit-il?

Quel code est exécuté lors de cet appel?

```
o. retrait(100);
```

- si `retrait` est dans la classe ou héritée d'une super-classe?
- si `retrait` est re-définie dans la classe ou dans une super-classe?



# Rappel: types statique et dynamique

Quels sont les type statique et dynamique de o?

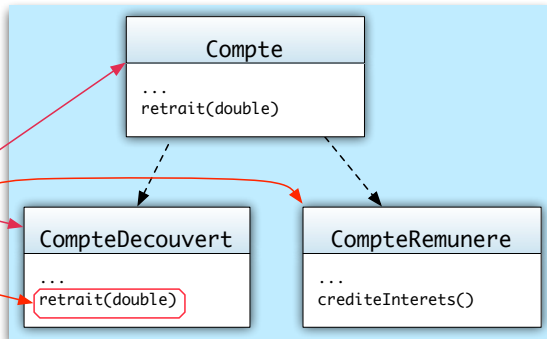
```
Compte o = ...;  
o.retrait(100); // Quelle est la methode executee??
```

type statique

Compte o  ??

types  
dynamiques  
possibles

re-définition



# Liaison dynamique

Liaison dynamique  $\approx$  répond à la question: «quel code est exécuté?»

## Liaison dynamique

`o.m()` exécute la méthode correspondant au **type dynamique** de l'objet `o`.

Exemple:

---

```
Compte o = new CompteDecouvert (...);  
o.retrait(100); // Quelle est la methode executee??
```

---

- Ici, le type dynamique de `o` est `CompteDecouvert`
- $\Rightarrow$  exécution méthode **redéfinie** dans cette classe.

# Liaison dynamique : exemples (1)

Quelle méthode exécutée ici?

---

```
Compte o = new CompteRemunere (...);  
o.retrait(100);
```

---

- Type dynamique de `o = CompteRemunere`  
⇒ exécution de la méthode **héritée** de `Compte`.

## Liaison dynamique : exemples (2)

Pour chaque méthode invoquée quel est le code exécuté:

```
Compte o = new CompteDecouvert (...);  
Compte r = new CompteRemunere (...);  
o.virement(100,r); // Quelle methode retrait executee??
```

TD de `o = CompteDecouvert`  $\Rightarrow$  `virement` **héritée** de `Compte`

```
/** Dans la classe Compte */  
void virement(Compte c, double m) throws ... {  
    this.retrait(m); c.depot(m);  
}
```

- `this.retrait(m)`  $\Rightarrow$  `retrait` **redéfini** par TD de `this(CompteDecouvert)`
- `r.depot(m)`  $\Rightarrow$  `depot` (**hérité**) par TD de `r (CompteRemunere)`.

Ici, une méthode héritée (`virement`) exécute une méthode redéfinie (`retrait`).

# Liaison dynamique: un test

---

```
Compte c;  
Terminal.ecrireString  
  ("Creation_d'un_compte_decouvert(O/N)?_");  
char rep = Terminal.lireChar();  
if ((rep != 'O') && (rep != 'o')){  
  Terminal.ecrireString("Creation_compte_ordinaire");  
  c = new Compte(10000.00) ;  
} else {  
  Terminal.ecrireStringln("Creation_compte_decouvert");  
  Terminal.ecrireString("Decouvert_maximal?_");  
  double max = Terminal.lireDouble();  
  c = new CompteDecouvert(10000.00, max) ;  
}
```

---

# Liaison dynamique: suite du test

- TS de  $c \Rightarrow$  `Compte`,
- à l'exécution, son TD sera `Compte` ou `CompteDecouvert`,
- à la compilation on ne connaît pas son TD.

---

```
Terminal.ecrireStringln
    ("solde_avant_retrait_"+c.getSolde());
try { c.retrait(11000.00);
    Terminal.ecrireStringln
        ("retrait_OK;_nouveau_solde=_"+ c.getSolde());
} catch (Exception e)
    {Terminal.ecrireStringln
        ("Probleme_lors_du_retrait");}
```

---

Résultat dépendra du type dynamique de `c`