

# L'héritage

Cours 8 / GL / LP DA2I

---

Cédric Lhoussaine

October 20, 2017

# Principes

---

après le **polymorphisme**, mécanisme caractéristique des langages à objets:

- du point de vue **implémentation**: permet de réutiliser le code d'une classe;
- du point de vue **modélisation**: une autre façon de réaliser la relation *sorte-de*.

mécanisme permettant de faire profiter *automatiquement* une classe dite **dérivée** D des *attributs* et des *méthodes* d'une autre classe, dite classe **de base** B.

- D dérive de B
- D hérite de B
- D étend B
- D est une sous-classe de B
- B est une superclasse de D

La classe dérivée D:

- possède les **attributs** de la classe de base B (et peut y accéder sauf s'ils sont privés);
- possède les **méthodes** de B (idem);
- peut déclarer de **nouveaux attributs**;
- peut définir de **nouvelles méthodes**;
- peut **redéfinir des méthodes** héritées de B

*La relation de dérivation est transitive*

# L'héritage en Java

- mot-clef extends :

```
class D extends B
```

# L'héritage en Java

- mot-clef `extends` :

```
class D extends B
```

- toute classe dérive, directement ou non (par transitivité), de la classe `Object`

```
class Toto {...}
```

équivalent en fait à:

```
class Toto extends Object {...}
```

# L'héritage en Java

- mot-clef `extends` :

```
class D extends B
```

- toute classe dérive, directement ou non (par transitivité), de la classe `Object`

```
class Toto {...}
```

équivalent en fait à:

```
class Toto extends Object {...}
```

- l'arbre des dérivations est visible dans la documentation générée par `javadoc`;



# L'héritage en Java

- mot-clef `extends` :

```
class D extends B
```

- toute classe dérive, directement ou non (par transitivité), de la classe `Object`

```
class Toto {...}
```

équivalent en fait à:

```
class Toto extends Object {...}
```

- l'arbre des dérivations est visible dans la documentation générée par `javadoc`;
- une classe ne peut dériver directement que **d'une seule superclasse** (*héritage simple*).

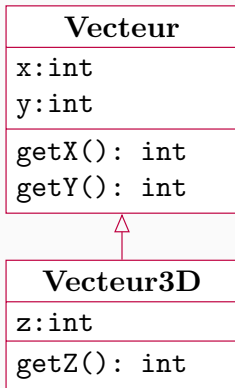
# Exemple

```
public class Vecteur {  
    protected int x, y;  
    public Vecteur() {  
        x = 0;  
        y = 0;  
    }  
    public Vecteur(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}  
public class Vecteur3D extends Vecteur {  
    protected int z;  
  
    public Vecteur3D(int x, int y, int z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
}
```

# Example

```
public class Vecteur {
    protected int x, y;
    public Vecteur() {
        x = 0;
        y = 0;
    }
    public Vecteur(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
}

public class Vecteur3D extends Vecteur {
    protected int z;
    public Vecteur3D(int x, int y, int z){
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public int getZ() { return z; }
}
```



*Toute instance de Vecteur3D est aussi de type Vecteur*  
→ *polymorphisme*

# Protection d'accès : rappel

Modificateurs d'attributs et de méthodes:

	classe	package	sous-classes	partout
private	X			
(rien)	X	X		
protected	X	X	X	
public	X	X	X	X

## Héritage $\neq$ Instanciation

- l'**instanciation** crée une relation d'appartenance d'un objet à une classe  $\rightarrow$  relation *est-un*
  - $(4, 3)$  est un Vecteur
- l'**héritage** est une relation d'inclusion entre classes  $\rightarrow$  relation *sorte-de* (*toute instance de D est aussi un B*)
  - un Vecteur3D est une sorte de Vecteur

## Héritage $\neq$ Encapsulation

- un objet peut en **encapsuler** d'autres  $\rightarrow$  relation *partie-de* (agrégation, composition...)
  - un Polygone est *composé* de plusieurs Points.
- **l'héritage** est une spécialisation ou un enrichissement
  - un Chat est une sous-espèce de Vertébré
  - un Vecteur3D rajoute une dimension au Vecteur (2D)

## Héritage $\neq$ Implémentation d'interfaces

- les **interfaces** définissent de façon abstraite un **comportement contractuel**
  - la classe Chat *s'engage* à posséder des méthodes énumérées dans l'interface Compagnon
- **l'héritage** sert à réutiliser du code, le spécialiser, etc.
  - la classe Chat possède toutes les méthodes de Vertébré + une méthode ronronner().



# Les identifiants `this` et `super`

- `this` et `super` sont des *pseudo-variables*
- `this`:
  - désigne *l'instance courante* (celle qui est en train d'exécuter le code)
  - a comme *type* celui de la **classe d'appartenance** de l'instance.
- `super` :
  - désigne aussi *l'instance courante*;
  - a pour *type* celui de la **superclasse**;
  - sert à faire explicitement appel au code (attributs et méthodes) *défini dans la superclasse*.

# Redéfinition

---

*TRÈS IMPORTANT:*

- *possible mais à éviter car source de bugs*
- *interdit dans de nombreux langages*

## *TRÈS IMPORTANT:*

- *possible mais à éviter car source de bugs*
- *interdit dans de nombreux langages*

## Définition

déclaration dans une classe dérivée d'un attribut *de même nom* que dans la classe de base

- de type différent ou identique!
- l'attribut de la superclasse est *seulement masqué*: il ne disparaît pas mais n'est plus accessible directement;
- `this.attribut` vs. `super.attribut`.

## Définition

déclaration dans la classe dérivée d'une méthode de *même signature* que dans la classe de base

- même nom et type de retour;
- mêmes arguments (type et nombre).

## Conséquences

- le nouveau code sera utilisé à **la place** du code hérité;
- le code hérité reste accessible par `super.laMethode(...)`

# Redéfinition des méthodes

```
public class Vecteur {
    protected int x, y;
    public Vecteur() {
        x = 0;
        y = 0;
    }
    public Vecteur(int x, int y){
        this.x = x;
        this.y = y;
    }
    public String toString(){ return "X=" + x + ", Y=" + y; }
}

public class Vecteur3D extends Vecteur {
    protected int z ;

    public Vecteur3D(int x, int y, int z){
        this.x = x;
        this.y = y;
        this.z = z;
    }
    // Redéfinition de la méthode toString()
    public String toString() { return "X="+x+", Y="+y+", Z="+z; }
}
```

# Redéfinition des méthodes

En général on cherche à réutiliser le code des superclasses et à le compléter :

```
public class Vecteur3D extends Vecteur {  
    protected int z ;  
  
    public Vecteur3D(int x, int y, int z){  
        this.x = x;  
        this.y = y;  
        this.z = z;  
    }  
  
    public int getZ(){  
        return z;  
    }  
  
    public String toString(){  
        return super.toString() + ", Z=" + z;  
    }  
}
```

# Redéfinition des méthodes

Le code exécuté dépend de la **classe** de l'objet (pas de son type)

```
Vecteur v1, v2;  
v1 = new Vecteur(314, 159);  
v2 = new Vecteur3D(3, 4, 5); // polym. d'affectation  
System.out.println(v1);  
System.out.println(v2);
```

X=314, Y=159

X=3, Y=4, Z=5

→ **polymorphisme de méthode**



Le polymorphisme est possible puisque toute classe dérivée:

- peut utiliser le **type** de ses superclasses
- possède les **méthodes** de ses superclasses (soit héritées, soit redéfinies)
- le mécanisme de liaison retardée (*late binding* ou *dynamic dispatch*) assure l'exécution du code approprié pour une *signature donnée*, en fonction de la classe de l'objet, en remontant dans l'arborescence des superclasses.

# Héritage et polymorphisme

```
public class Vecteur {  
    protected int x, y;  
  
    public Vecteur() {  
        x = 0;  
        y = 0;  
    }  
  
    public Vecteur(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
  
    public String toString(){  
        return "X=" + x + ", Y=" + y;  
    }  
  
    public void afficher(){  
        System.out.println(this.toString());  
    }  
}
```

# Héritage et polymorphisme

La classe Vecteur3D reste inchangée, et on écrit:

```
Vecteur v1, v2;  
v1 = new Vecteur(314, 159);  
v2 = new Vecteur3D(3, 4, 5);  
v1.afficher();  
v2.afficher();
```

Résultat:

X=314, Y=159

X=3, Y=4, Z=5

→ liaison retardée

# Constructeurs

---

## Quelques spécificités de Java

- un objet *sans constructeur* peut être instancié au moyen du *constructeur sans paramètres*
- en Java, les constructeurs **ne sont pas hérités!!!**
- lors d'une instanciation, le code des constructeurs des *superclasses* est utilisé (explicitement ou non).

## 1ère conséquence

Il faut **explicitement** définir les constructeurs dont on a besoin (ou dont les sous-classes auront besoin).

```
public class Vecteur3D extends Vecteur {  
    /** Constructeur qui crée un vecteur nul en 3D */  
    public Vecteur3D()  
    {  
        this.x = 0;  
        this.y = 0;  
        this.z = 0;  
    }  
    ...  
}
```

## 2nde conséquence

Tout constructeur doit *commencer*:

- soit par une appel explicite à un constructeur de la superclasse  
→ *toujours préférable*
- soit par rien de particulier si la superclasse possède un constructeur sans paramètres → *appel implicite à ce constructeur*

## super(...) et les constructeurs

on peut (doit) faire appel, dans un constructeur, à un constructeur de la superclasse `super(...)`.

```
public class Vecteur3D extends Vecteur {  
    public Vecteur3D(int x, int y, int z) {  
        super(x,y) ;  
        this.z = z ;  
    }  
    // ...  
}
```

*ATTENTION: `super(...)` doit être la première instruction du constructeur!*



au moment de l'instanciation :

1. les attributs reçoivent immédiatement une valeur par défaut (selon leur **type**);
2. un constructeur de la classe est choisi (selon la **signature** de l'appel);
3. un constructeur de la **superclasse** est appelé;
4. les attributs sont éventuellement initialisés;
5. les instructions d'initialisation sont exécutées;
6. les instructions du constructeur sont exécutées.

# Exemple

```
class A { public A() { ... } }

class B extends A {
    int i = 1; // 1) valeur par défaut i = 0
               // 4) initialisation i = 1
    i = 2;      // 5) instructions d'initialisation i = 2
    public B(int x) { // 2) choix de B(int)
                   // 3) appel implicite à super()
                   // 6) instructions constructeur i = 5
        i = x;
    }
}

new B(5); // instruction d'instanciation
```

# Problèmes de conception

---

## Définition

mécanisme permettant de définir **plusieurs superclasses** pour une même classe dérivée.

## Exemple

un Carré peut être défini comme:

- un Rectangle dont les longueur et largeur sont égales
- un Losange dont les côtés sont perpendiculaires

→ le Carré est à la fois une sorte de Rectangle et une sorte de Losange

## Héritage multiple et langages

- autorisé par certains langages : C++, Ada95, Python, Ocaml. . .
- interdit en Java !
- permet de représenter l'appartenance d'objets à plusieurs espèces (points de vue) et de factoriser plus de code
- pose des difficultés pour la liaison retardée si méthodes de même signature dans les superclasses: quel code exécuter ?

# Héritage entre interfaces

L'héritage **même multiple** est possible entre interfaces

- par exemple, dans `java.util`: `Set`, `List` dérivent de `Collection`
- syntaxe: `interface I1 extends I0 {...}`
- toutes les signatures de `I0` sont implicitement dans `I1`
- si une classe `C` implémente `I1`, alors `C` implémente forcément **aussi** `I0`

# Héritage entre interfaces

Héritage multiple entre interfaces:

```
import java.util.*;  
public interface EnsembleOrdonne extends Set, List {}
```

L'interface EnsembleOrdonne contient:

- toutes les signatures de Set
- toutes les signatures de List
- toutes les signatures de Collection

# Que signifie l'héritage ?

## Factoriser du code

Qui doit hériter de qui ?

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



# Que signifie l'héritage ?

## Solution ?

il semble logique de choisir comme **superclasse** celle dont les attributs sont présents partout (réutilisation de code !)

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

on ne s'intéresse pas qu'à la factorisation du code:

- approche **opérationnelle** de l'héritage: on raisonne sur l'extension des données et des traitements encapsulés;
- approche **conceptuelle** de l'héritage: on raisonne sur le degré plus ou moins grand de **généralité** ou de **spécificité** des objets.

*dans le doute : privilégier l'approche conceptuelle.*

# Exceptions

---

Mécanisme d'interruption d'un traitement en cas d'anomalie → rôle de **signalisation** et **d'arrêt**

- en Java : objet dérivé de `java.lang.Exception`
- doit être "levée" par `throw uneException`
- la méthode susceptible de lever une exception doit l'indiquer dans sa signature: `throws TypeException`
- peut être "capturée" par un bloc spécifique `try ... catch(TypeException) ...`

# Exceptions

Exemple d'empilement sans exception:

```
public class PileTableau implements Pile {  
    // ...  
    public void empiler(Object o)  
    {  
        if (!estPleine()){  
            contenu[nbElements] = o;  
            nbElements++;  
        }  
        // sinon ne rien faire  
    }  
    // ...  
}
```

# Exceptions

Exemple d'empilement avec exception:

```
public class PileException extends Exception {  
    public PileException(String msg) {  
        super(msg);  
    }  
}  
  
public class PileTableau implements Pile {  
    // ...  
    public void empiler(Object o) throws PileException {  
        if (estPleine())  
            throw new PileException("Pile Pleine") ;  
        contenu[nbElements] = o ;  
        nbElements++ ;  
    }  
    // ...  
}
```

# Propagation d'une exception

Effets de la levée d'une exception:

- **sortie immédiate** de la méthode
- **propagation** d'un nouvel objet (en général instancié au moment du `throws`)
- **remontée** dans l'arbre d'appel à la recherche d'une méthode capable de *capturer* l'exception
- si pas de capture: sortie du programme avec un message d'erreur

# Capture d'une exception

On peut intercepter des objets de type `Exception` lors de la remontée dans l'arbre d'appel.

```
Pile p = new PileTableau();
try {
    System.out.println(p.sommet());
} catch (PileException e) {
    System.out.println(e.getMessage());
}
```

- le code du bloc `try` est exécuté;
- si aucune exception n'est levée, les blocs `catch` sont ignorés;
- sinon le **premier** bloc `catch` avec un paramètre du **bon type** est exécuté **dès la levée de l'exception**;
- on peut avoir plusieurs blocs `catch` avec des paramètres de types différents;
- on peut avoir un bloc `finally`.

- la possibilité de lever une exception **fait partie de la signature** de la méthode (`throws`) → attention aux interfaces et à l'héritage!



- la possibilité de lever une exception **fait partie de la signature** de la méthode (`throws`) → attention aux interfaces et à l'héritage!
- l'appel d'une méthode levant une exception doit se faire
  - soit dans un bloc `try... catch...` capable de **capturer** ce type d'exception
  - soit dans le corps d'une méthode capable de **propager** cette exception (`throws`)

- la possibilité de lever une exception **fait partie de la signature** de la méthode (`throws`) → attention aux interfaces et à l'héritage!
- l'appel d'une méthode levant une exception doit se faire
  - soit dans un bloc `try... catch...` capable de **capturer** ce type d'exception
  - soit dans le corps d'une méthode capable de **propager** cette exception (`throws`)

*Une exception doit **toujours** être capturée ou propagée*

# Capture ou propagation

```
public class TestException extends Exception {
    public TestException(String msg) {
        super(msg);
    }
}

public class Test {
    public void method() throws TestException {
        throw new TestException("bouh!");
    }
    // propagation
    public void context1() throws TestException {
        method();
    }
    // ou capture
    public void context2() {
        try {
            method();
        } catch (TestException e) {
            System.out.println(e.getMessage()+" attrapé!");
        }
    }
}
```

Mécanisme **préférable aux réponses par défaut**:

- contrôle par le **compilateur** : signatures et capture
- interruption intelligible lors de l'exécution
- veiller à avoir **peu de code** dans les blocs try
- veiller à avoir autant de blocs catch que de types d'exceptions possibles

# Les classes abstraites

---

# Problème de factorisation de code

Blocs cubique:

```
public class Cube {  
    protected double cote;  
    protected double masse_volumique;  
    public Cube(double mv, double c) {  
        masse_volumique = mv;  
        cote = c;  
    }  
    public double masse() {  
        return masse_volumique * volume() ;  
    }  
    public double volume() {  
        return cote * cote * cote ;  
    }  
}
```

# Problème de factorisation de code

Blocs sphériques:

```
public class Boule {  
    protected double r ;  
    protected double masse_volumique ;  
    public Boule(double mv, double r) {  
        masse_volumique = mv ;  
        this.r = r ;  
    }  
    public double masse() {  
        return masse_volumique * volume() ;  
    }  
    public double volume() {  
        return Math.PI * r * r * r * 4 / 3;  
    }  
}
```

# Problème de factorisation de code

→ écrire une superclasse Bloc pour regrouper les portions de code redondantes:

```
public class Bloc {  
    protected double masse_volumique;  
    public Bloc(double mv) {  
        masse_volumique = mv;  
    }  
    public double masse() {  
        return masse_volumique * volume();  
    }  
}
```



# Problème de factorisation de code

→ Réécriture des classes Cube et Boule:

```
public class Cube extends Bloc {
    protected double cote;
    public Cube(double mv, double c) {
        super(mv);
        cote = c;
    }
    public double volume() {
        return cote * cote * cote;
    }
}

public class Boule extends Bloc {
    protected double r;
    public Boule(double mv, double r) {
        super(mv);
        this.r = r;
    }
    public double volume() {
        return Math.PI * r * r * r * 4 / 3;
    }
}
```

# Mauvaise solution

```
public class Bloc {  
    protected double masse_volumique ;  
    public Bloc(double mv) {  
        masse_volumique = mv;  
    }  
    public double masse() {  
        return masse_volumique * volume();  
    }  
    public double volume() {  
        return 0;  
    }  
}
```

la méthode `volume()` est maintenant définie dans `Bloc` mais elle n'a **aucun sens**!

La bonne solution:

- placer dans Bloc la **signature** de la méthode `volume()`;
- mais indiquer qu'on ne peut pas définir son code.

→ `volume()` est une **méthode abstraite** = méthode dont on ne définit que la **signature** (comme dans les interfaces)

# Solution correcte

```
public abstract class Bloc {  
    protected double masse_volumique;  
    public Bloc(double mv) {  
        masse_volumique = mv;  
    }  
    public double masse() {  
        return masse_volumique * volume();  
    }  
    public abstract double volume(); // signature seule  
}
```

- les sous-classes de Bloc définissent le code de volume()
- Bloc devient une *classe abstraite*

## Définition

Une classe abstraite est une classe qui **ne peut pas avoir d'instances**. Elle peut contenir :

- du code:
  - des **attributs** (variables de classe ou d'instance)
  - des **méthodes concrètes**
  - des **constructeurs**
- des **méthodes abstraites** (= des signatures)

Utilisations multiples :

- factorisation de code  $\rightarrow$  plus grande généricité
- représentation d'abstractions à différents niveaux
- contrôle des objets instanciables par l'utilisateur

# Classe abstraite vs. interface

- *points communs:*
  - peuvent contenir de simples **signatures**;
  - **pas instanciables**;
  - définition d'un **type commun** pour plusieurs objets → implémentation de la relation *sorte-de*.
- *différences:*
  - **jamais de code** dans les interfaces
  - **méthodes concrètes** et **constructeurs** autorisés dans les classes abstraites
  - les interfaces concernent ce qui est **public**
  - notion de contrat vs. réutilisation de code

# Abstraction et héritage

que se passe-t-il lorsqu'on hérite de méthode abstraites ?

```
public abstract class AbstractToto {  
    public abstract int maMethode();  
}  
public class Toto extends AbstractToto {}
```

- la classe Toto hérite de toutes les méthodes de AbstractToto;
- elle hérite en particulier de `abstract int maMethode();`
  - soit elle la **redéfinit** en indiquant du **code**
  - soit elle ne fait rien → **classe abstraite**, il faut donc la déclarer **abstract**!

*Une classe qui possède des méthodes abstraites doit être abstraite.*



Remarques importantes:

- une classe peut être déclarée *abstract* même si elle n'a pas de méthodes *abstraites*. Elle ne doit simplement pas être instanciée.
- une classe *concrète* peut dériver:
  - soit d'une autre classe *concrète*
  - soit d'une classe *abstraite* (et implémenter les éventuelles méthodes *abstraites*).
- une classe *abstraite* peut dériver:
  - soit d'une autre classe *abstraite*
  - soit d'une classe *concrète*.

adapté d'un cours de Sébastien Picault