

# L3. Programmation orientée objet. Cours 10

Marie-Pierre Béal (Cours de Rémi Forax)

Héritage



Héritage

Héritage

java.lang.Object

java.lang.Object

# java.lang.Object

Classe "mère" de toutes les autres classes

```
public class Author { }  
...  
Object o = new Author(); // ok
```

Marche aussi avec les interfaces

```
public interface I { }  
...  
I i = ...  
Object o = i; // ok
```

Ne marche pas avec les types primitifs (on doit utiliser les wrappers)

## Object.getClass()

Renvoie un objet de type `java.lang.Class` qui représente la classe d'un objet à l'exécution.

- La notation `Foo.class` permet d'obtenir le même objet

### Exemple

```
"hello".getClass() // String.class
String s = "hello";
s.getClass()        // String.class
Object o = "hello";
o.getClass()         // String.class
```

# toString()/equals()/hashCode()

java.lang.Object fournit une implantation par défaut

- equals(Object o)
  - return this == o;
- hashCode()
  - renvoie un nombre tiré au hasard, une seule fois!
- String toString()
  - return getClass().getName() + "@" + hashCode()

Donc un objet est égal (equals) à lui même et hashCode est une valeur qui peut être utilisée pour le debug

Héritage

Héritage

# Idée de Modula (1960+)

Et si on pouvait définir une classe comme une sorte de classe déjà existante avec des champs supplémentaires ?

- J'ai déjà la classe `Point`, si j'ajoute une couleur, je peux définir `ColoredPoint`

Comme c'est une "sorte de", je peux appeler toutes les méthodes qui prennent un `Point` et passer un `ColoredPoint` à la place.

- L'héritage implique le sous-typage

## extends

On hérite en utilisant le mot-clé **extends**

```
public class Point {  
    private final int x;  
    private final int y;  
    ...  
}  
  
public class ColoredPoint extends Point {  
    private final String color;  
    // cette classe a trois champs x, y et color  
}
```

On hérite des champs même s'ils sont privés.

On ne peut pas y accéder dans la sous-classe mais ils sont là.

On peut appeler des méthodes de la super-classe pour y accéder.

# Héritage et java.lang.Object

Lorsque l'on déclare une classe, le compilateur ajoute `extends Object` s'il n'y a pas de `extends`

```
public class Author extends Object {  
    ...  
}
```

Toutes les classes héritent de java.lang.Object

- Soit directement
- Soit indirectement, une classe hérite d'une autre classe, qui hérite directement de Object
  - Par exemple un record hérite de java.lang.Record qui hérite de Object.

## Hérite des méthodes d'instances

Les méthodes d'instance de la super-classe visibles dans la sous-classe sont accessibles.

```
public class Point {  
    ...  
    public double distanceToOrigin() {  
        ...  
    }  
}  
  
public class ColoredPoint extends Point {  
    ...  
}  
  
...  
var coloredPoint = new ColoredPoint(...);  
coloredPoint.distanceToOrigin(); // ok
```

# On n'hérite pas ...

On n'hérite pas :

- des constructeurs
  - Ils ne savent initialiser que la classe sur laquelle ils ont été définis
- des méthodes statiques
  - Elles sont définies sur une classe
- des méthodes privées
  - Elles ne sont pas visibles

## Appel au constructeur de la super-classe

Un constructeur doit appeler un constructeur de sa super-classe.  
Il vérifie les pré-conditions donc on ne peut pas passer outre.

```
public class Point {  
    ...  
    public Point(int x, int y) { ... }  
}  
public class ColoredPoint extends Point {  
    ...  
    public ColoredPoint(int x, int y, String color) {  
        super(x, y); // appelle le constructeur de Point  
        this.color = Objects.requireNonNull(color);  
    }  
}
```

L'appel à `super()` doit être la première instruction du constructeur

# Redéfinition

On **doit** remplacer/redéfinir (override) les méthodes dont on hérite et qui n'ont pas le bon code

```
public class SmallTax {  
    ...  
    public long computeTax() { /* 1 */ }  
}  
public class BigTax extends SmallTax {  
    ...  
    @Override  
    public long computeTax() { /* 2 */ }  
}  
...  
SmallTax tax = new BigTax(...);  
tax.computeTax(); // appelle 2
```

## Redéfinition et `super.m()`

La méthode redéfinie peut appeler la méthode de base avec la notation "`super.method()`"

■ `super` est équivalent à "`this`" mais avec le type de la super-classe

```
public class SmallTax {  
    ...  
    public long computeTax() { ... }  
}  
public class BigTax extends SmallTax {  
    ...  
    @Override  
    public long computeTax() {  
        return 10 * super.computeTax();  
        // appelle SmallTax::computeTax  
    }  
}
```

# Héritage

L'héritage ce sont trois choses :

- Le sous-typage
  - Si `B extends A`, alors on peut écrire `A a = new B();`
- On récupère les membres d'instances
  - Si `B extends A` et il existe `A::m()`, `b.m()` appelle `A::m()`
  - Si `B extends A` et il existe `A::x`, `b.x` appelle `A::x` (beurk!)
- Le polymorphisme (late binding)
  - Si `B::m()` redéfinit `A::m()` et `A a = new B()`, alors `a.m()` appelle `B::m()` à l'exécution

# Héritage vs Interface

L'héritage ce sont trois choses :

| Héritage                             | Interface                           |
|--------------------------------------|-------------------------------------|
| sous-typage                          | sous-typage                         |
| On hérite des champs et des méthodes | On récupère les méthodes par défaut |
| On redéfinit les méthodes            | On implante les méthodes            |

La grosse différence est que les interfaces ne gèrent pas les champs

# Héritage et maintenance

Hériter pose de vrais problèmes

## Modularité

- L'implantation n'est plus à un seul endroit, il faut regarder toute la hiérarchie
  - Gros problème si on hérite d'une classe du JDK/d'une librairie, on ne contrôle plus l'implantation.

## Correction du code

- Il faut redéfinir toutes les méthodes qui n'ont pas la bonne sémantique.
  - Gros problème si on hérite d'une classe du JDK/d'une librairie. On doit redéfinir des méthodes qui n'existent pas encore.

# Héritage et maintenance (2)

Hériter **des champs** pose de vrais problèmes

## Mutabilité

- La classe qui hérite peut avoir un champ non final, donc aucun type n'est vraiment non mutable

## Egalité

- Si on a un `equals()` dans la super-classe, `coloredPoint.equals(point)` et `point.equals(coloredPoint)` n'exécutent pas le même code

# Interface et délégation

Interface et délégation  
(comment ne pas faire d'héritage)

# On utilise une interface

## On utilise une interface pour le sous-typage

```
public interface Point {  
    double distanceToOrigin();  
}  
  
public class Point2D implements Point {  
    private final int x;  
    private final int y;  
    @Override  
    public double distanceToOrigin() { /* 1 */ }  
}  
  
public class ColoredPoint implements Point {  
    private final int x;  
    private final int y;  
    private final Color color;  
    @Override  
    public double distanceToOrigin() { /* 2 */ }  
}
```

Mais est-ce que cela veut dire que l'on va dupliquer du code ?

# On utilise la délégation

On utilise la délégation pour réutiliser le code

```
public interface Point {  
    double distanceToOrigin();  
}  
  
public class Point2D implements Point {  
    private final int x;  
    private final int y;  
    @Override  
    public double distanceToOrigin() { /* 1 */ }  
}  
  
public class ColoredPoint implements Point {  
    private Point2D point; // délégation  
    private final Color color;  
    @Override  
    public double distanceToOrigin() {  
        return point.distanceToOrigin(); // délégation  
    }  
}
```

# Réutilisation par délégation

La délégation permet de choisir les méthodes que l'on expose dans l'API

- Avec l'héritage (beurk!)

```
public class Library extends ArrayList<Book> {} // mauvais
```

Il faut redéfinir toutes les méthodes qui permettent d'ajouter un livre pour faire un test à null : `add()`, `addAll()`, `ListIterator.add()`, plus celles qui n'existent pas encore.

- Avec la délégation

```
public class Library {  
    private final ArrayList<Book> books; // délégation  
}
```

Ici, on est libre d'exposer uniquement les méthodes que l'on veut et de faire uniquement les tests à null nécessaires, on contrôle l'API.

Empêcher l'héritage et mutation

Empêcher l'héritage et mutation

# Classe final

Si une classe est déclarée **final**, alors on ne peut pas en hériter

- Toutes les classes non mutables doivent être **final** sinon une sous-classe peut avoir un champ non **final**
- Tous les records sont **final**

```
public final class Point {  
    ...  
}
```

# Classe sealed

On liste (**permits**) les classes qui peuvent hériter

On contrôle que toutes les classes sont non mutables

```
public sealed class Point permits ColoredPoint {  
    private final int x;  
    private final int y;  
    ...  
}  
public final class ColoredPoint extends Point {  
    private final Color color;  
    ...  
}
```

Héritage et égalité

# Héritage et égalité

(attention, c'est technique ...)

# Instanceof vs héritage

On doit écrire un equals/hashCode dans ColoredPoint pour prendre en compte la couleur, mais utiliser instanceof dans la classe de base ne fait pas ce que l'on veut :(

```
public class Point {  
    private final int x;  
    private final int y;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        // ce code est faux à cause de l'héritage  
        return o instanceof Point p && x == p.x && y == p.y;  
    } // + hashCode()  
}
```

```
public final class ColoredPoint extends Point {  
    private final String color;  
    ...  
    @Override  
    public boolean equals(Object o) {  
        // ce code est Ok, la classe est final  
        return o instanceof ColoredPoint colorPoint  
            && super.equals(o) && color.equals(colorPoint.color);  
    } // + hashCode()  
}
```

# InstanceOf vs héritage suite

```
new Point(1, 2).equals(new Point(1, 2)) // appelle Point::equals
new ColoredPoint(1, 2, "red").equals(new ColoredPoint(1, 2, "blue"))
                                     // appelle ColoredPoint::equals
new Point(1, 2).equals(new ColoredPoint(1, 2, "red"))
                                     // appelle Point::equals donc true, ahhhh
```

## getClass()

En cas d'héritage, on doit tester si les deux classes sont identiques avec getClass().

```
public class Point {
    private final int x;
    private final int y;
    ...
    @Override
    public boolean equals(Object o) {
        if (o == null || getClass() != o.getClass()) { return false; }
        var p = (Point) o;
        return x == p.x && y == p.y;
    } // + hashCode()
}

public final class ColoredPoint extends Point {
    private final String color;
    ...
    @Override
    public boolean equals(Object o) {
        return o instanceof ColoredPoint colorPoint
            && super.equals(o) && color.equals(colorPoint.color);
    } // + hashCode()
}
```

## getClass() suite

```
new Point(1, 2).equals(new Point(1, 2)) // appelle Point::equals
new ColoredPoint(1, 2, "red").equals(new ColoredPoint(1, 2, "blue"))
                                     // appelle ColoredPoint::equals
new Point(1, 2).equals(new ColoredPoint(1, 2, "red"))
                                     // appelle Point::equals donc false
```

# Utiliser une interface (version 1)

La vraie solution est de ne pas utiliser l'héritage, mais une interface.

```
public interface Point { } // on utilisera surement un sealed + permits ici
public final class Point2D implements Point {
    private final int x;
    private final int y;
    ...
    @Override
    public boolean equals(Object o) { // ok classe final
        return o instanceof Point2D p && x == p.x && y == p.y;
    } // + hashCode()
}
public final class ColoredPoint implements Point {
    private final int x;
    private final int y;
    private final String color;
    public ColoredPoint(int x, int y, String color) {...}
    @Override
    public boolean equals(Object o) { // ok classe final
        return o instanceof ColoredPoint p
            && x == p.x && y == p.y && color.equals(p.color);
    } // + hashCode()
}
```

# Utiliser une interface suite

```
new Point2D(1, 2).equals(new Point2D(1, 2)) // appelle Point2D::equals
new ColoredPoint(1, 2, "red").equals(new ColoredPoint(1, 2, "blue"))
                                     // appelle ColoredPoint::equals
new Point2D(1, 2).equals(new ColoredPoint(1, 2, "red"))
                                     // appelle Point2D::equals donc false
```

## Utiliser une interface (version 2)

La vraie solution est de ne pas utiliser l'héritage, mais une interface.

```
public interface Point { } // on utilisera surement un sealed + permits ici
public final class Point2D implements Point {
    private final int x;
    private final int y;
    ...
    @Override
    public boolean equals(Object o) { // ok classe final
        return o instanceof Point2D p && x == p.x && y == p.y;
    } // + hashCode()
}

public final class ColoredPoint implements Point{
    private final Point2D point;
    private final String color;
    public ColoredPoint(int x, int y, String color) {
        ...
    }
    @Override
    public boolean equals(Object o) { // ok classe final
        return o instanceof ColoredPoint p
            && point.equals(p.point) && color.equals(p.color);
    } // + hashCode()
}
```



# Package et Visibilité

# Package

Un package est une suite de répertoires (dans le système de fichiers ou dans un jar) dans lequel on range une classe

- Évite les collisions de noms : deux classes nommées List par exemple.

Java utilise la notation reverse DNS

- Si vous êtes chez Google, votre package commence par `com.google.nomdeproject`

Par convention les noms de package sont en minuscules et ne finissent pas par 's' ou un nombre

# Package dans le code source

On place les classes du package dans le bon répertoire

- Par exemple `fr.uge.licence` est dans le répertoire `fr/uge/licence`

On écrit "`package fr.uge.licence;`" en première instruction des fichiers `.java` du package

- Si le chemin et la première instruction ne sont pas les mêmes, le compilateur râle

# Quatre visibilités de Java

Java possède quatre visibilités mais trois modificateurs

"package" est une visibilité sans modificateur

- **private** signifie visible que entre '`{`' et '`}`' de la classe
- **package** signifie visible par les classes du même package
- **protected** signifie visible par les classes du même package et les sous-classes
- **public** signifie visible par tout le monde (si la classe est elle-même public)

# Visibilité package

Relaxe la notion de private, les membres sont accessibles aux classes du même package

Sert à partager des champs et méthodes d'implantation entre deux classes

- Par exemple, une table de hachage et une classe qui stocke les couples clé/valeur

# Visibilité protected

Accessible par les classes du même package ou les sous-classes

- Cela veut dire qu'une classe peut hériter d'une classe d'un autre package
- Gros problème de maintenance car les deux packages sont pas forcément gérés par la même société/personne

Copier/coller du C++

- pas utilisé en pratique sauf dans les vieux codes (< 2000)

Surcharge vs Redéfinition

# Surcharge vs Redéfinition

# Surcharge vs Redéfinition

(si on hérite ou implante une interface)

La redéfinition (override), c'est le fait de remplacer une méthode par une autre dans un sous-type (entre 1 et 2 dessous)

La surcharge (overload), c'est le fait d'avoir plusieurs méthodes ayant le même nom (entre 2 et 3 dessous)

```
public class Point {  
    public void m(Point p) { .. } // 1  
}  
  
public class ColoredPoint extends Point {  
    @Override  
    public void m(Point p) { .. } // 2  
    public void m(ColoredPoint p) { .. } // 3  
}
```

# Surcharge

Dans une classe (ou avec les méthodes héritées), plusieurs méthodes peuvent avoir

- le même nom et
- un nombre et des types de paramètres différents

Deux méthodes ne peuvent pas différer uniquement par le type de retour

```
public class Point {  
    public void m(int i) { ... }  
    public int m(int i) { ... } // ne compile pas  
                                // Pas le même type de retour, pas suffisant  
}
```

# Algorithme de résolution de la surcharge

## À la compilation

- Pour un appel de méthode donné, le compilateur trouve toutes les méthodes appelables
  - Si aucune méthode n'existe, il plante
- Puis, parmi les méthodes appelables, il sélectionne la méthode la plus spécifique (celle qui a les meilleurs sous-types)
  - Si aucun n'est plus spécifique que les autres, il plante

# Exemples

Avec l'appel `point.m("hello")`

```
public class Point {  
    public void m(CharSequence seq) { ... }  
    public void m(Object o) { ... }  
}
```

Appelle `m(CharSequence)` car les deux sont appelables, et `CharSequence` est plus spécifique que `Object`.

Avec l'appel `point.m("hello", "hello")`

```
public class Point {  
    public void m(Object o, String s) { ... }  
    public void m(String s, Object o) { ... }  
}
```

Le compilateur plante, les deux sont appelables, aucune n'est plus spécifique que l'autre.

# Redéfinition

Il y a redéfinition entre une méthode d'un supertype et une méthode d'un sous-type si l'une est substituable à l'autre par le **dynamic dispatch** lors de l'exécution.

```
public class Point {  
    public double distanceToPoint(Point p) { ... }  
}  
  
public class ColoredPoint extends Point {  
    @Override  
    public double distanceToPoint(Point p) { ... }  
}
```

# Condition du *Dynamic Dispatch*

Site d'appel

Point p = new ColoredPoint();

retour result =

p.method(paramètres);

Compilation

Execution

```
public class Point {  
    visibilité retour method(paramètres)  
    throws exceptions {  
        ...  
    }  
}
```

```
public class ColoredPoint extends Point {  
    visibilité2 retour2 method(paramètres2)  
    throws exceptions2 {  
        ...  
    }  
}
```

Il y a redéfinition, si la substitution à l'exécution est possible

# Visibilité

Site d'appel

```
Point p = new ColoredPoint();
```

```
retour result =
```

```
    p.method(paramètres);
```

Compilation

Execution

```
public class Point {  
    visibilité retour method(paramètres)  
    throws exceptions {  
        ...  
    }  
}
```

```
public class ColoredPoint extends Point {  
    visibilité2 retour2 method(paramètres2)  
    throws exceptions2 {  
        ...  
    }  
}
```

Il faut que la visibilité soit au moins aussi grande  
`package` < `protected` < `public`

Sinon cela veut dire que l'on a moins de droits  
si on appelle directement `ColoredPoint::method`

# Type de retour

Site d'appel

Point p = new ColoredPoint();

*retour* result =

p.method(paramètres);

Compilation

Execution

```
public class Point {  
    visibilité retour method(paramètres)  
    throws exceptions {  
        ...  
    }  
}
```

```
public class ColoredPoint extends Point {  
    visibilité2 retour2 method(paramètres2)  
    throws exceptions2 {  
        ...  
    }  
}
```

Le type de retour peut être un sous-type

Car la valeur de retour doit être assignable dans la variable "result"

# Type des paramètres

Site d'appel

```
Point p = new ColoredPoint();
```

```
retour result =
```

```
p.method(paramètres);
```

Compilation

Execution

```
public class Point {  
    visibilité retour method(paramètres)  
    throws exceptions {  
        ...  
    }  
}
```

```
public class ColoredPoint extends Point {  
    visibilité2 retour2 method(paramètres2)  
    throws exceptions2 {  
        ...  
    }  
}
```

Les types des paramètres pourraient être des super-types

Mais la signature d'une méthode est stockées sous forme d'une String dans le fichier .class, pour une recherche rapide, donc le type des paramètres doit être exactement le même en Java

# Type des exceptions checkées

Site d'appel

```
Point p = new ColoredPoint();
```

```
try {  
    retour result =  
        p.method(paramètres);  
} catch(exceptions) {  
    ...  
}
```

```
public class Point {  
    visibilité retour method(paramètres)  
    throws exceptions {  
        ...  
    }  
}  
  
public class ColoredPoint extends Point {  
    visibilité2 retour2 method(paramètres2)  
    throws exceptions2 {  
        ...  
    }  
}
```

Compilation

Execution

Le type d'une exceptions checkée peut être  
un sous-type d'une exception checkée existante  
(**exceptions2** peut aussi être vide)

Il faut que les exceptions soient attrapables par le catch(**exceptions**)

# Condition de redéfinition

Une méthode d'un sous-type redéfinit une méthode d'un super-type si

- La méthode est au moins aussi visible
- Le nom est identique
- Les paramètres sont identiques
- Les types de retour sont covariants  
(covariant  $\Rightarrow$  sont des sous-types)
- Les exceptions checkées sont covariantes  
(covariantes  $\Rightarrow$  sont des sous-types)

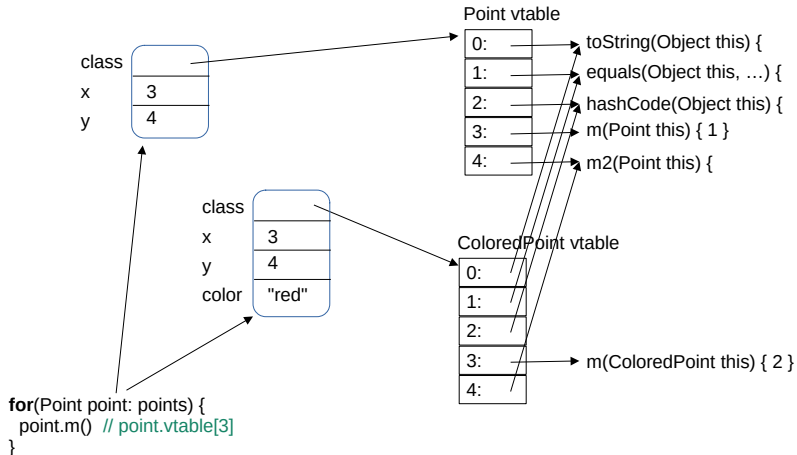
En mémoire

En mémoire

```

public class Point { private final int x, y; public void m() { 1 } public void m2() { ... } }
public class ColoredPoint extends Point { private final String color; public void m() { 2 } }

```



En résumé

En résumé

# En résumé

L'héritage est le goto de la POO

## Ne pas faire d'héritage

- Préférer les rateaux aux arbres (les interfaces à l'héritage)
- Si une librairie utilise l'héritage, comment on fait ?
  - On utilise l'héritage, on n'a pas le choix

Ne pas confondre la surcharge et la redéfinition

- La redéfinition demande le même nom, les mêmes paramètres, une visibilité au moins aussi grande et un type de retour et des exceptions covariantes