

Suite sur l'héritage et polymorphisme

Christelle CAILLOUET
(christelle.caillouet@unice.fr)

IE du jeudi 16 mars

- Durée 1h30
- **Programme** : tout depuis le début jusqu'à cette semaine incluse
 - Savoir écrire une classe en Java
 - Règles d'écriture et d'indentation
 - Types primitifs, objets, tableaux, fichiers, exceptions
 - Instanciation, encapsulation, composition, héritage, polymorphisme
- Seul document autorisé : **1 feuille A4 manuscrite recto/verso non photocopiee**

Retour sur le TD4

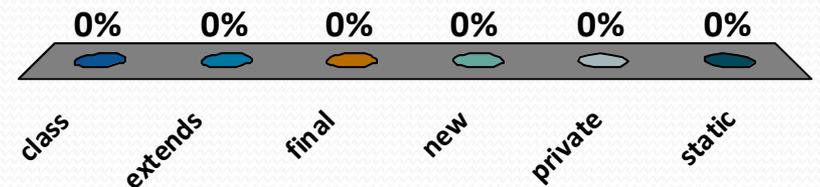
Eclipse et l'héritage

- Créer des classes dérivées directement
- Paramétrage à la création
- Génération des constructeurs dérivés avec appel à **super**

En Java, quel mot clé est relié à l'héritage ?

- A. class
- ✓ B. extends
- C. final
- D. new
- E. private
- F. static

30



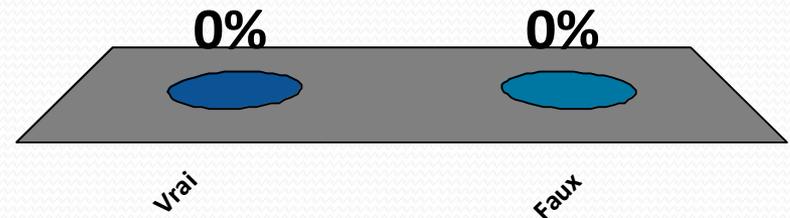
Ce constructeur dérivé est-il correct ?

```
public ColoredPoint(double x, double y, Color c) {  
    couleur = c;  
    super(x, y);  
}
```

A. Vrai

✓ B. Faux

30



Suite sur l'héritage

Héritage des méthodes

- Le code des méthodes de la classe de base peut ne plus être correct dans la classe dérivée
 - Pour l'exemple de `Point` et `ColoredPoint` :
 - La méthode `distanceP` est toujours valide
 - La méthode `toString` ne parle pas de la couleur

```
public static void main(String[] args) {  
    ColoredPoint cp = new ColoredPoint();  
    System.out.println(cp);  
    System.out.println(cp.couleur);  
}
```

```
C:\Users\cmolle\Documents\COURS\2016-2017\M213\2017\Code>java ColoredPoint  
<0.0 , 0.0>  
java.awt.Color[r=0,g=0,b=0]
```

Redéfinition et surcharge de membres

- Réécrire/Modifier des méthodes existantes
- On l'a déjà vu avec les méthodes de la classe `Object`
- Généralisation à toutes les classes avec l'héritage

Redéfinition de méthode

- Fournir une nouvelle définition pour la **même méthode**
 - Même nom
 - Mêmes arguments (nombre et types)
 - Code différent
- L'annotation *@Override* demande au compilateur de vérifier (facultative)

Principes de la redéfinition

La redéfinition d'une méthode (dans la classe dérivée) :

- N'est pas possible sur une méthode *static*
 - Ne doit pas diminuer les droits d'accès par rapport à celle de la classe de base
 - Par contre, elle peut les augmenter : *protected* → *public*
 - Ne doit pas lever plus d'exceptions
 - Peut en lever moins
 - Doit avoir le même type de retour ou un type dérivé
-
- La résolution est faite en 2 temps :
 1. **Compilation** : on vérifie que c'est possible sur le type déclaré
 2. **Exécution** : on cherche la + précise étant donné le type « réel » (cf. polymorphisme)

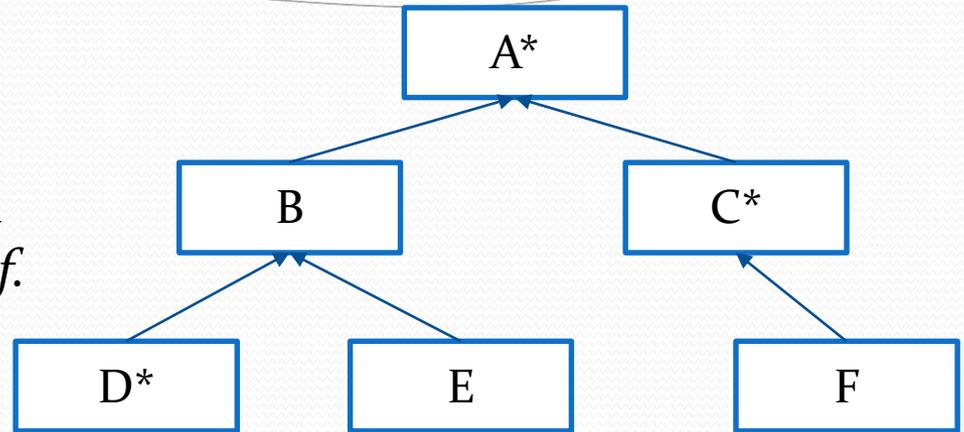
Exemple : méthode *toString()* de `Object`

```
public String toString() {  
    return "Point "+couleur+" "+super.toString();  
}
```

- L'objectif est de lui donner une définition + précise (mieux adaptée aux objets de la classe dérivée que de la classe de base) de sorte qu'elle soit appelée à l'exécution

Soit l'arborescence d'héritage suivante :

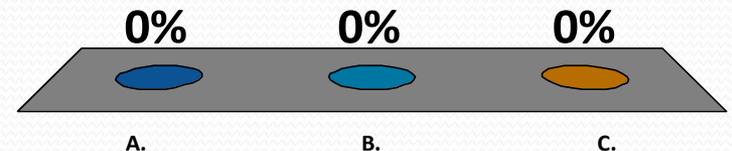
L'astérisque * signale la définition ou la redéfinition d'une méthode f .



Quelle méthode f est appelée pour une instance de la classe B ?

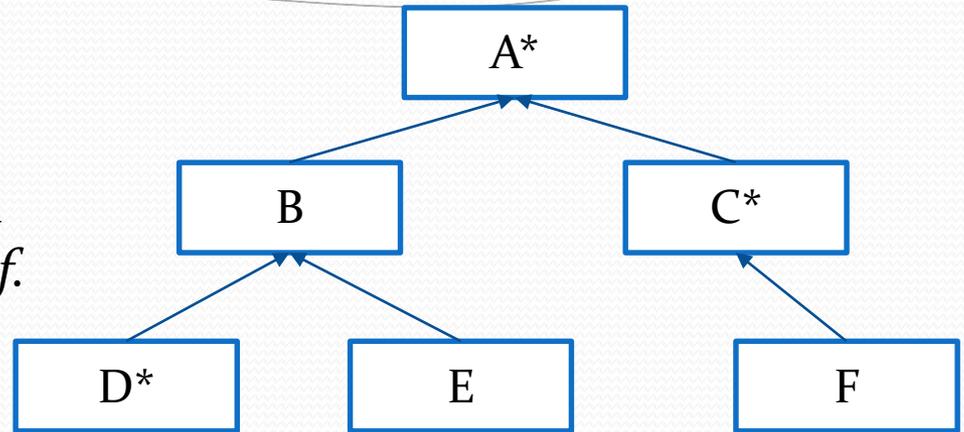
- ✓ A. Méthode f de A
- B. Méthode f de C
- C. Méthode f de D

30



Soit l'arborescence d'héritage suivante :

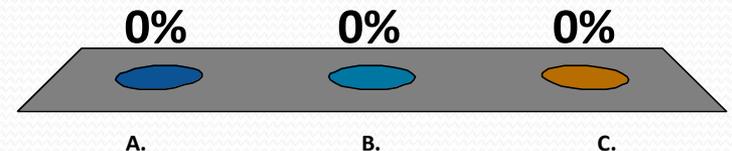
L'astérisque * signale la définition ou la redéfinition d'une méthode f .



Quelle méthode f est appelée pour une instance de la classe E ?

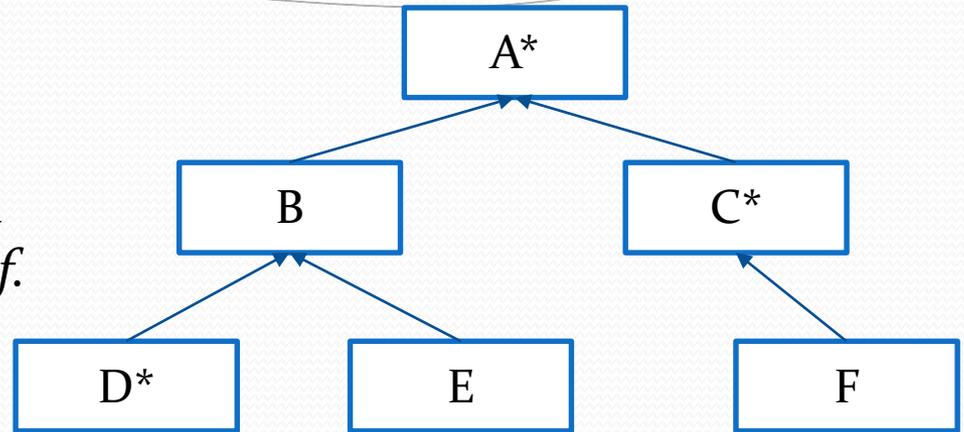
- ✓ A. Méthode f de A
- B. Méthode f de C
- C. Méthode f de D

30



Soit l'arborescence d'héritage suivante :

L'astérisque * signale la définition ou la redéfinition d'une méthode f .



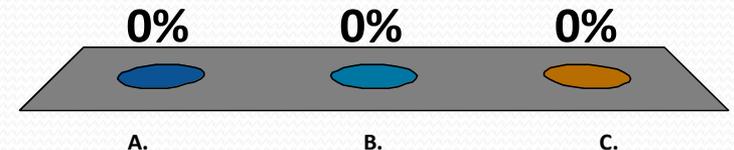
Quelle méthode f est appelée pour une instance de la classe F ?

A. Méthode f de A

✓ B. Méthode f de C

C. Méthode f de D

30

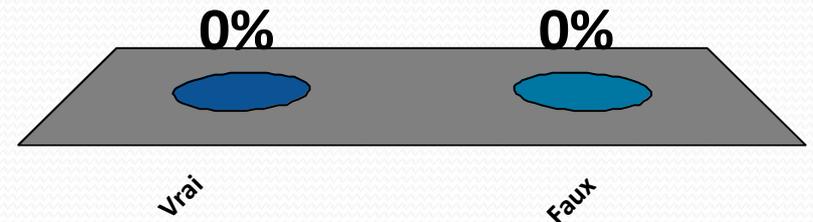


Peut-on redéfinir une méthode de classe *static* dans une classe dérivée ?

A. Vrai

✓ B. Faux

30



Les classes et méthodes *final*

- Le mot-clé *final* existe pour les méthodes :
 - Il signifie que la méthode ne pourra pas être redéfinie dans une classe dérivée
 - Peut être utile pour garantir qu'aucune autre définition ne pourra être donnée pour cette méthode (sécurité)
- Le mot-clé *final* existe pour les classes (vu cours précédent) :
 - Impossible d'hériter de cette classe
 - Les méthodes se comportent comme si elles étaient *final*

Surcharge dans une classe

- Surcharger une méthode (dans une même classe), c'est avoir plusieurs définitions possibles pour cette méthode :
 - Même nom
 - Signatures différentes : nombre d'arguments ou types des arguments
- Détection de la bonne méthode à la compilation :
 - Le compilateur doit être capable de faire un choix

```

public class Point {
    private double abscisse;
    private double ordonnee;

    public Point(double x, double y){
        abscisse = x;
        ordonnee = y;
    }

    public void deplace(double dx, double dy)
    {
        abscisse += dx;
        ordonnee += dy;
    }

    public void deplace(double dx) {
        abscisse += dx;
    }

    public void deplace(float dx) {
        abscisse += dx;
    }
}

```

Surcharge de la méthode
deplace

```

public class Surdef1 {
    public static void main(String[] args) {
        Point a = new Point(1.0, 2.0);
        a.deplace(1, 3); // appelle deplace(double, double)
        a.deplace(2.0); // appelle deplace(double)
        float b = 1.5f;
        a.deplace(b); // appelle deplace(float)
    }
}

```

Ambiguïté

```
public void deplace(double dx, float dy) {
    abscisse += dx;
    ordonnee += dy;
}
public void deplace(float dx, double dy) {
    abscisse += dx;
}
```

Relié à la notion de transtypage (conversion de types) automatique et polymorphisme

```
public class Surdef1 {
    public static void main(String[] args) {
        Point a = new Point(1.0, 2.0);
        float b = 1.5f;
        double c = 2.0;
        a.deplace(b, c); // OK
        a.deplace(c, b); // OK
        a.deplace(b, b); // erreur : ambiguïté
    }
}
```

```
Surdef1.java:8: error: reference to deplace is ambiguous
    a.deplace(b, b); // erreur : ambiguïté
    ^
    both method deplace(double,float) in Point and method deplace(float,double) in
    Point match
1 error
```

Surcharge et héritage

- **Surcharge** vs. Redéfinition :
 - La signature de la méthode dérivée n'est pas la même que dans la classe de base

⚠ Les 2 méthodes cohabitent dans la classe dérivée

Le polymorphisme

Le polymorphisme

- **Définition :**
 - Faculté d'un objet d'être une instance de plusieurs classes
- Concept extrêmement puissant en POO, qui complète l'héritage
- **Intérêt :**
 - Manipuler des objets sans en connaître (*tout à fait*) le type

Exemple

Gestion de tableaux
hétérogènes :

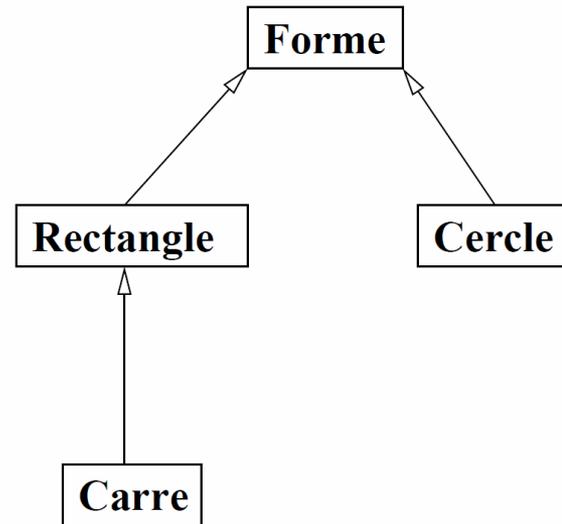


FIGURE 4.1 – Exemple de relations d'héritage

```
Forme[] tableau = new Forme[4];  
tableau[0] = new Rectangle(10, 20);  
tableau[1] = new Carre(15);  
tableau[2] = new Rectangle(5, 30);  
tableau[3] = new Carre(10);
```

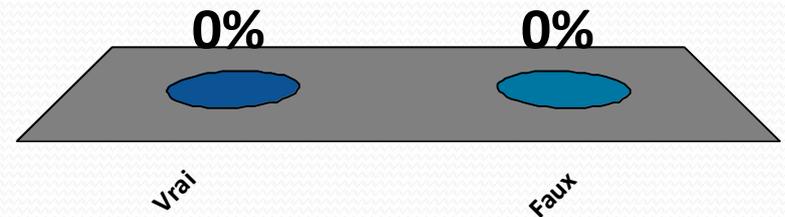
Un ColoredPoint est avant tout un Point

Cette instantiation est-elle autorisée en Java ?

```
Point p;  
p = new ColoredPoint(1.5, 2.5, Color.red);
```

- ✓ A. Vrai
- B. Faux

30



Transtypage implicite (ascendant)

- Capacité d'une variable de classe de base à recevoir une référence sur un objet de sa descendance.
- Compatibilité par affectation entre un type classe et un type ascendant (conversion implicite légale)

```
Capitale paris = new Capitale("Paris", 2250000, "France");  
// Capitale hérite de Ville  
Ville v = paris;  
  
// Capitale hérite de Ville qui hérite de Object  
Object o = paris;
```

Transtypage explicite (descendant)

```
ColoredPoint p;  
p = new Point(1.5, 2.5);
```

Erreur de compilation !



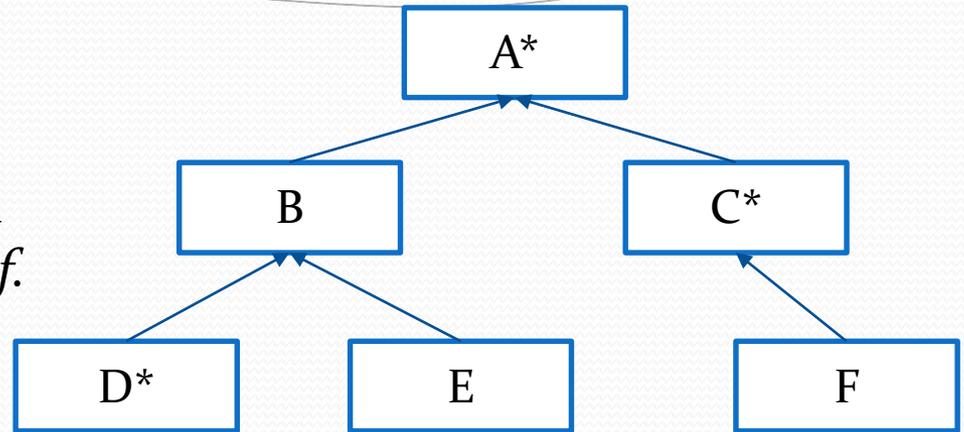
```
ColoredPoint cp = new ColoredPoint(1.5, 2.5, Color.red);  
Point p = cp;  
  
// Transtypage explicite  
ColoredPoint c = (ColoredPoint)p;
```

Soit l'arborescence d'héritage suivante :

L'astérisque * signale la définition ou la redéfinition d'une méthode *f*.

Soient les déclarations :

```
A a = new A(); B b = new B(); C c = new C();  
D d = new D(); E e = new E(); F f = new F();
```

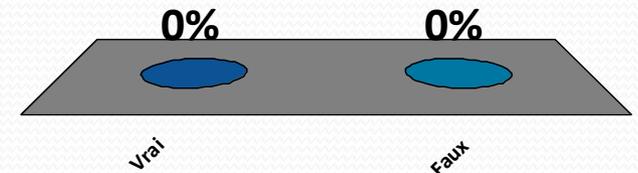


Les affectations suivantes sont-elles légales ?

```
a=b; a=c; a=d ; a=e; a=f;  
b=d; b=e;  
c=f;
```

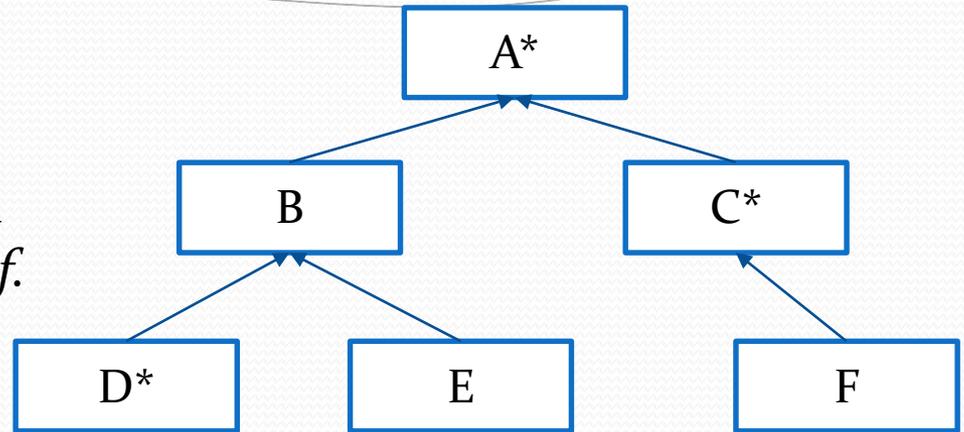
- ✓ A. Vrai
- B. Faux

30



Soit l'arborescence d'héritage suivante :

L'astérisque * signale la définition ou la redéfinition d'une méthode *f*.



Soient les déclarations :

```
A a = new A(); B b = new B(); C c = new C();
```

```
D d = new D(); E e = new E(); F f = new F();
```

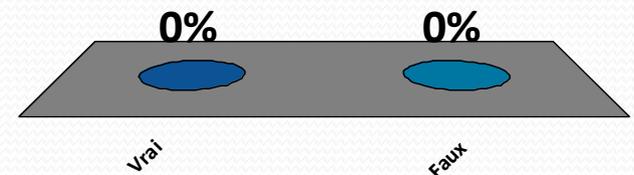
Les affectations suivantes sont-elles légales ?

```
b=a; d=c; c=d;
```

A. Vrai

✓ B. Faux

30

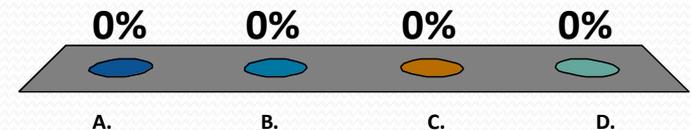


Quelle est la séquence d'appel des méthodes *toString()* ?

```
Point p;  
p = new Point(5, 0.5);  
System.out.println(p);  
p = new ColoredPoint(1.5, 2.5, Color.red);  
System.out.println(p);
```

- A. ColoredPoint - ColoredPoint
- B. ColoredPoint - Point
- ✓ C. Point - ColoredPoint
- D. Point - Point

30



Ligature dynamique

```
Point p;  
p = new Point(5, 0.5);  
System.out.println(p);  
p = new ColoredPoint(1.5, 2.5, Color.red);  
System.out.println(p);
```

- L'instruction se fonde :
 - Sur le type effectif de l'objet référencé par p
 - Et non sur le type de la variable p
- ➔ Permet d'obtenir un comportement adapté à chaque type d'objet sans avoir besoin de tester sa nature

Opérateur *instanceof* et méthode *getClass()*

- *instanceof* : teste l'appartenance d'un objet à une classe (réponse à la question *est instance de ... ?*)
- Pour connaître la classe exacte de l'objet à l'exécution (type effectif) :
getClass().getName()

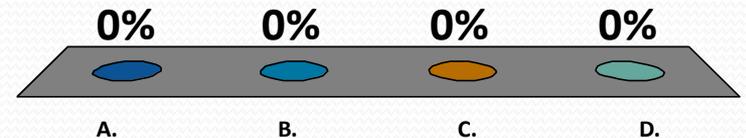
```
Point p;  
p = new ColoredPoint(1.5, 2.5, Color.red);  
System.out.println(p instanceof Point);  
System.out.println(p.getClass().getName());
```

Quel résultat produit le code suivant ?

```
Point p;  
p = new ColoredPoint(1.5, 2.5, Color.red);  
System.out.println(p instanceof Point);  
System.out.println(p.getClass().getName());
```

- A. false - ColoredPoint
- B. false - Point
- C. true - ColoredPoint
- D. true - Point

30



Polymorphisme, redéfinition et surcharge

- **A la compilation :**

- Des vérifications statiques sont effectuées sur le type de la variable
- a est de type $A \rightarrow$ on ne peut lui appliquer que les méthodes de A

- **A l'exécution :**

- La sélection du code à exécuter est effectuée dynamiquement en fonction du type effectif de la variable

Exemple

```
public class A{
    public void f(float x) {...}
}

public class B extends A{
    public void f(float x) {...} // redéfinition de f de A
    public void f(int n) {...} // surcharge de f pour A et B
}

A a = new A();
B b = new B();
int n;
a.f(n); // appelle f(float) de A
b.f(n); // appelle f(int) de B
a = b; // a contient une référence sur un objet de type B
a.f(n); // appelle f(float) de B et non f(int)
```

Exemple

1. Le compilateur recherche la meilleure méthode **parmi toutes celles s'appliquant au type de a** (à savoir A ou ses classes ascendantes)

→ La signature de la méthode et son type de retour sont figés
2. Lors de l'exécution, on se fonde sur le type de l'objet référencé par a pour trouver une méthode ayant la signature et le type de retour voulus

→ On aboutit à *void f(float x)* de **B**