

Polymorphisme

Programmation objet - II

L2 informatique

F. Bertrand

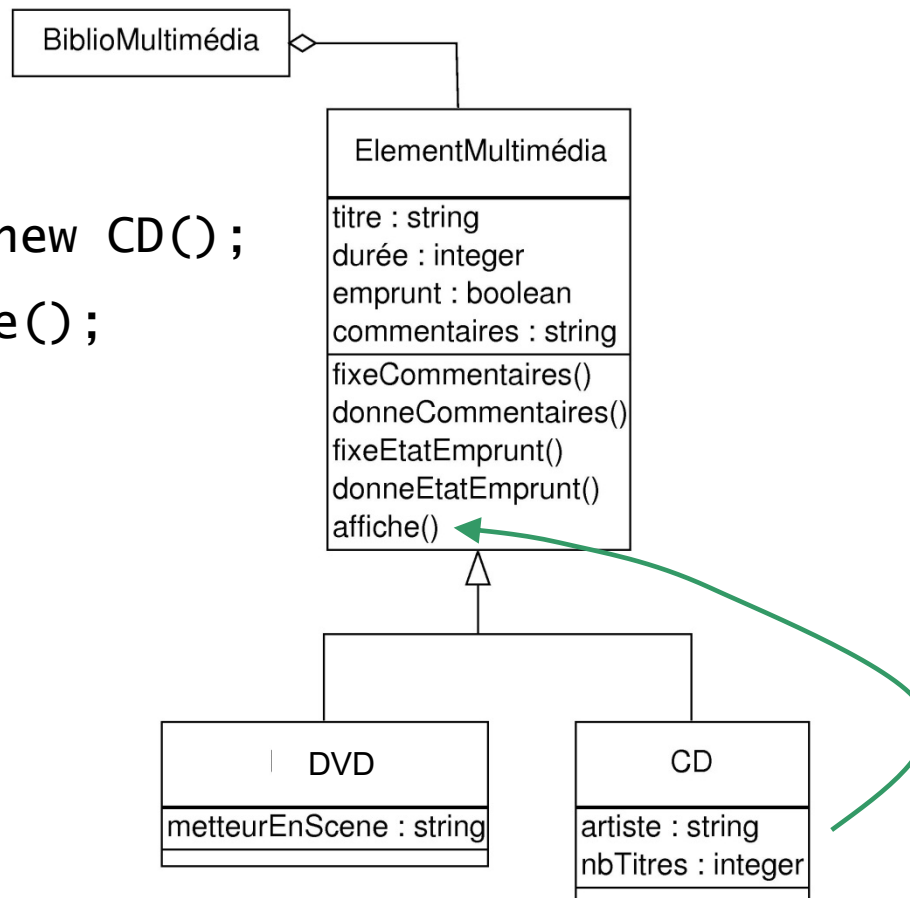
Au programme d'aujourd'hui...

- Appel de méthode dans une hiérarchie de classes
- Héritage de type et notion de sous-type
- Notion de polymorphisme
- Résolution dynamique des appels
 - Type statique et type dynamique
- Transtypage dans une hiérarchie de classes

Appel d'une méthode dans une hiérarchie de classes

- Exemple : appel de la méthode `affiche()` dans `CD`

```
CD unCD = new CD();  
unCD.affiche();
```



Appel d'une méthode dans une hiérarchie (suite)

- La recherche de la méthode va s'effectuer selon les étapes suivantes :
 1. Recherche dans la classe de l'objet appelé (CD)
 2. Puis recherche dans sa super-classe (ElementMultimedia)
 3. Puis, si la méthode n'est toujours pas trouvée, recherche dans Object (cas avec la méthode toString()).
- C'est donc la version **la plus proche** (dans la hiérarchie) du type de l'objet appelé qui sera utilisée.
- Il existe cependant une possibilité de masquage...

Appel d'une méthode dans une hiérarchie (suite)

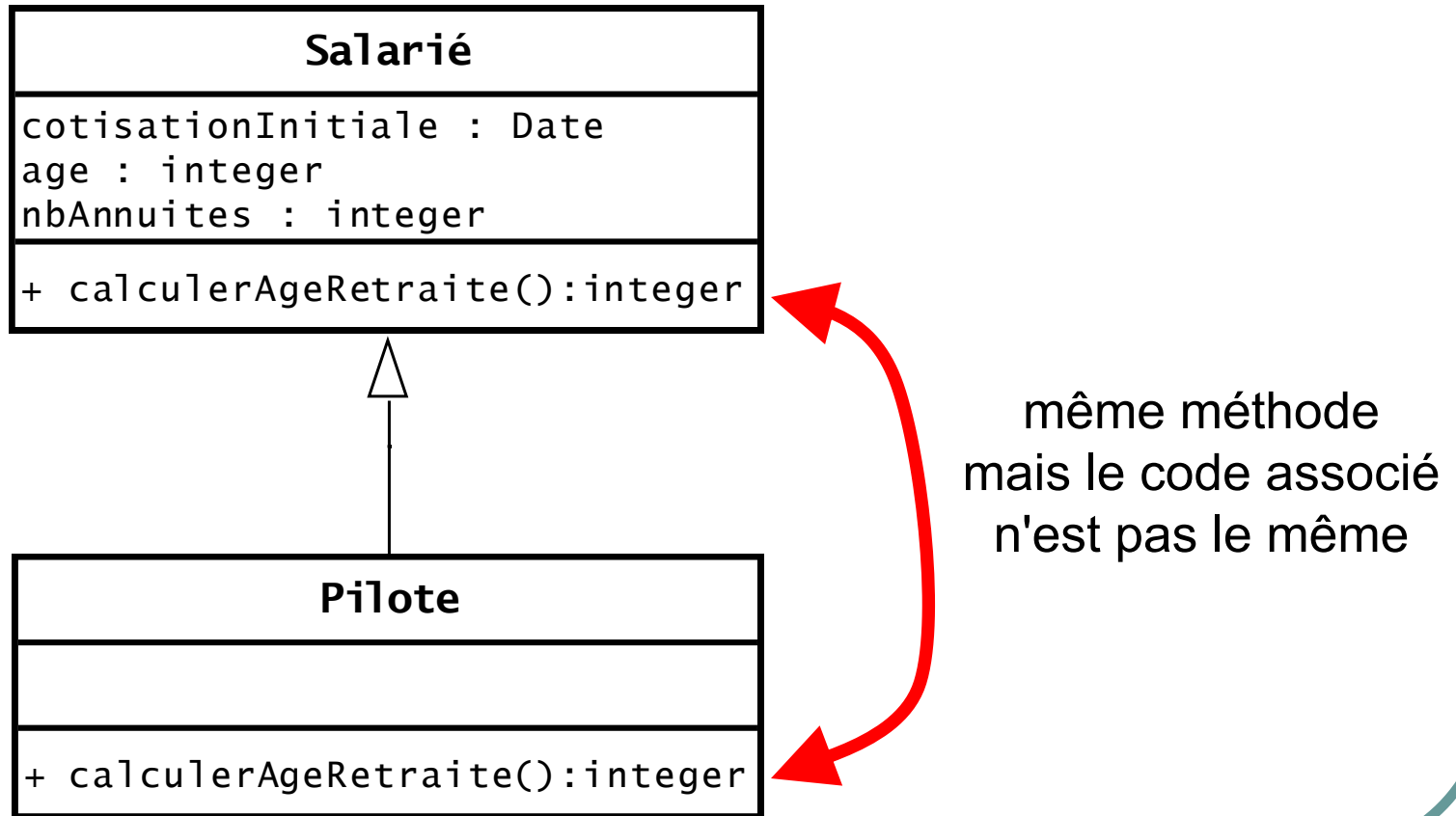
- La redéfinition (*overriding*), ou masquage, correspond à une situation où un identificateur (d'attribut ou de méthode) est présent **à la fois** dans la classe et dans sa super-classe :
 - Ceci est peu fréquent (et peu souhaitable !...) pour les attributs car le risque de confusion est important. Néanmoins ces derniers étant (normalement) toujours déclarés privés, ils ne sont pas accessibles dans la sous-classe.
 - Assez fréquent et utile pour les méthodes car cela permet de conserver le même nom de méthode mais en adaptant son code à la sous-classe...

Appel d'une méthode dans une hiérarchie (suite)

- Pratiquement la redéfinition s'effectue en conservant la même signature c'est-à-dire :
 - Le même nom de méthode
 - Le même type de retour
 - Les mêmes paramètres (même nombre et mêmes types)
- Le non-respect de ce principe conduit à la création d'une méthode non pas redéfinie mais **surchargée** !... (avec des règles d'appel différentes)

Appel d'une méthode dans une hiérarchie (suite)

- Exemple avec réutilisation de la méthode redéfinie :



Appel d'une méthode dans une hiérarchie (suite)

- Exemple (suite) : si on souhaite calculer l'âge de départ à la retraite d'un pilote en utilisant la méthode présente dans `Salarié`, comment procéder ?...

Redéfinition de la méthode héritée

```
public class Pilote extends Salarié {  
    public int calculerAgeRetraite() {  
        int ageNormal = super.calculerAgeRetraite();  
        return ageNormal - 5;  
    }  
}
```

Appel de la méthode héritée

Appel d'une méthode dans une hiérarchie (suite)

```
public class Salarié {  
    ...  
    public int calculerAgeRetraite() { ... }  
    public float calculMontantRetraite(int annee) { ... }  
}  
public class Pilote extends Salarié {  
    ...  
    public int calculerAgeRetraite() {  
        int ageNormal = calculerAgeRetraite();  
        return ageNormal - 5;  
    }  
    public float calculMontantRetraite(int annee, float primes) {  
        /* calcul */  
    }  
    public static void main(String[] args) {  
        Pilote p = new Pilote();  
        int age = p.calculerAgeRetraite();  
        float montant = p.calculMontantRetraite(2016);  
    }  
}
```

Peut être marquée
@overrides

Appel d'une
méthode redéfinie

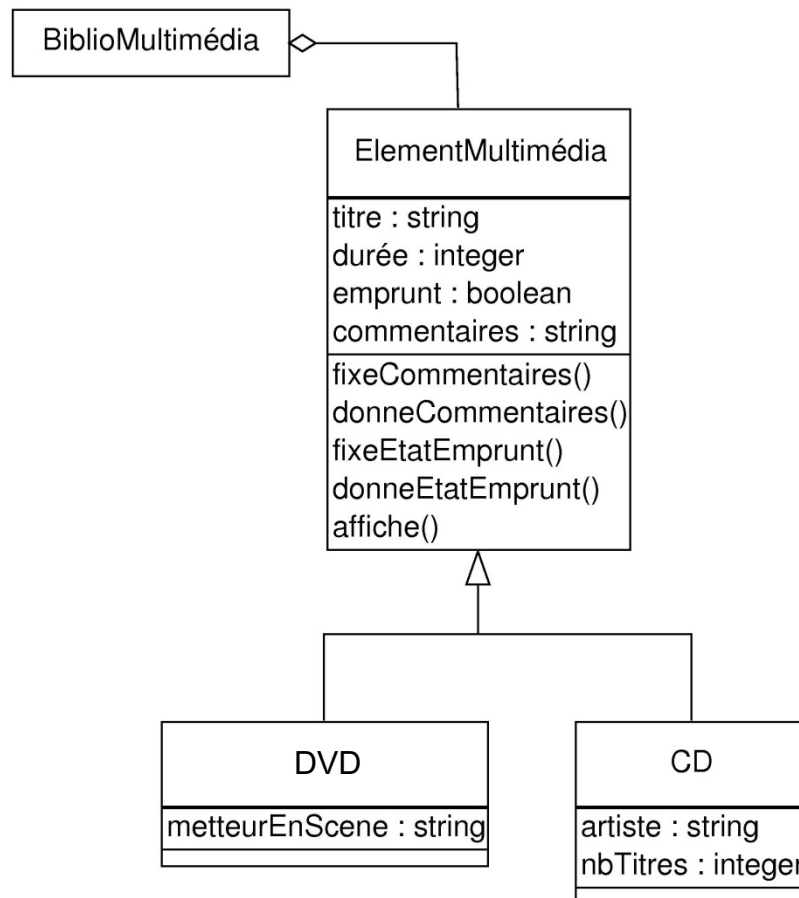
Appel d'une
méthode surchargée

Appel d'une méthode dans une hiérarchie (suite)

- En résumé :
 - Pour appeler une méthode (ou un attribut) redéfinie (masquée), il est nécessaire d'utiliser le mot-clé **super** comme **objet cible** de l'appel de méthode.
 - Son oubli conduit à un appel récursif (en général fatal !...) à l'exécution.
 - Une méthode ne peut être redéfinie qu'en conservant **strictement** sa signature...
 - L'appel est limité **uniquement à la super-classe** (pas d'accès direct aux classes supérieures)...

Héritage du type des super-classes

- Reprenons l'exemple du cours précédent :



Héritage du type des super-classes (suite)

- Avec l'héritage et la présence de la super-classe `ElementMM`, il devient possible d'utiliser une seule liste car :
 - un objet d'une classe hérite du type de sa super-classe
 - l'héritage est transitif
 - un objet appartient à **une seule classe** (précisée lors de l'appel à `new`) mais peut posséder **plusieurs types**

Sous-types et affectation

- La création d'une super-classe `ElementMM` permet de stocker des CD et des DVD dans une liste d'`ElementMM`
- Jusqu'à présent une référence de type T ne pouvait recevoir que l'adresse d'une instance de T (même type)
Par exemple : `CD unCD = new CD(...);`

- Avec l'introduction de la relation d'héritage, il est possible d'affecter à une variable de type T l'adresse d'une instance de type T' à **condition que T' soit un sous-type (sous-classe) de T** :

```
ElementMM elt = unCD; // OK
```

```
Object o = unCD;      // OK
```

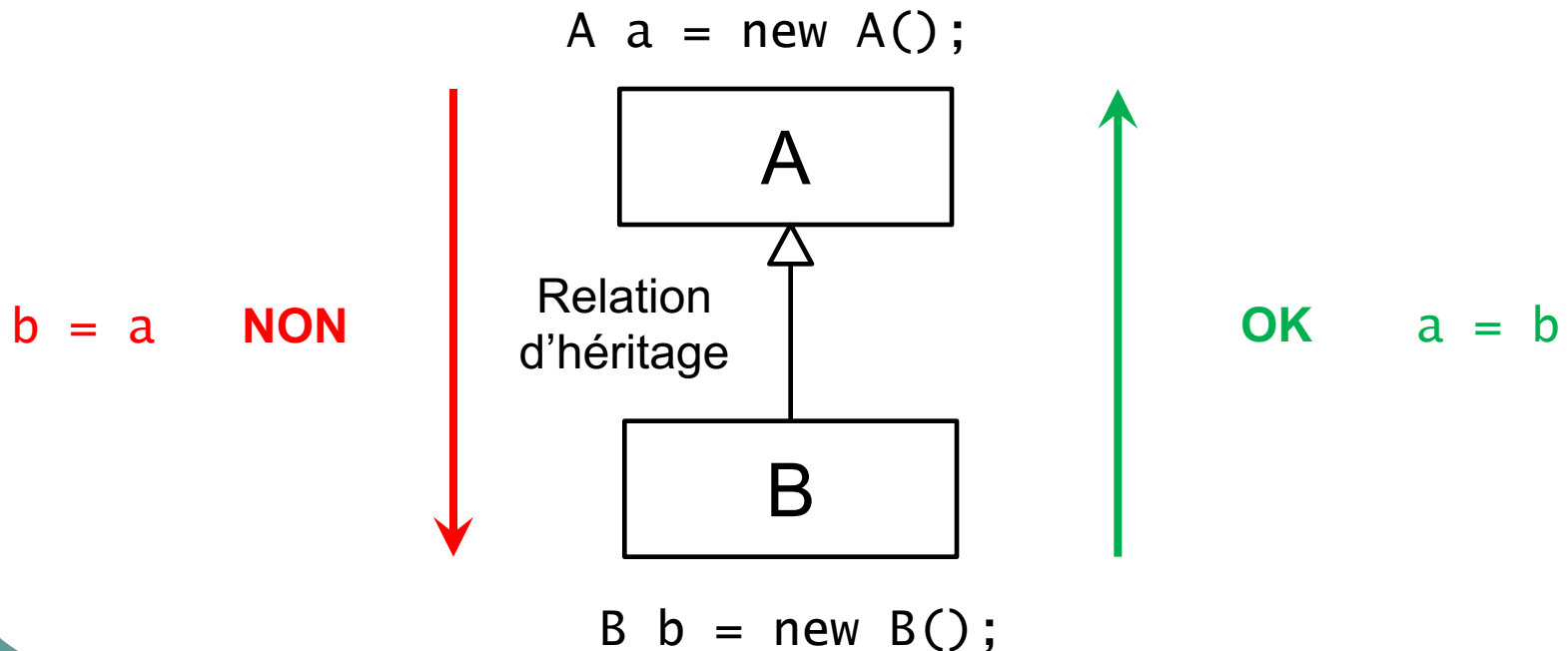
```
DVD unDVD = elt;      // FAUX ! (condition fausse)
```

Sous-types et affectation (suite)

- Un moyen assez simple pour se rappeler de la condition d'affectation est d'utiliser la relation « *est-un* » entre le type à droite de l'affectation et le type à gauche.
- Exemple :
 - `ElementMM elt = new CD(...);`
→ Un CD est un élément multimédia ? **Oui...**
 - `CD unCD = elt;`
→ Un élément multimédia est un CD ? **Non, cela peut être un DVD**
 - `Object o = ... ;`
→ **Toujours vrai car, dans le langage Java, toute classe hérite directement (ou indirectement) de la classe Object**

Sous-types et affectation (suite)

- Un autre moyen, par rapport au diagramme de classes, est de considérer que l'affectation ne peut s'effectuer que dans le sens de la relation d'héritage (généralement présentée de bas en haut)

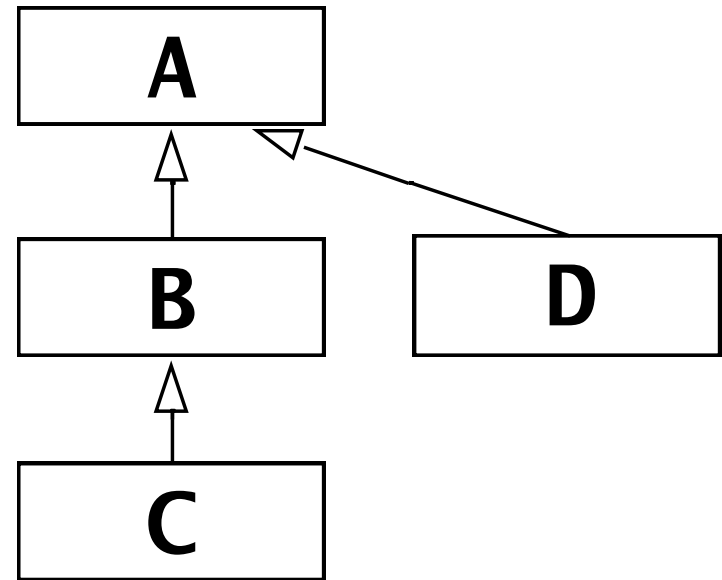


Sous-types et affectation (suite)

- D'autres exemples :

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

```
A a2 = new D(); // ?  
b = a; // ?  
a = c; // ?  
c = d; // ?
```



Sous-types et passage de paramètres

- La relation d'héritage influe sur l'affectation mais également sur le passage de paramètres.
- Le passage de paramètres peut être vu comme une affectation entre **le paramètre formel** (utilisé dans la définition) et **le paramètre réel** (utilisé lors de l'appel)

- Exemple :

Définition : `public void ajouterElt(ElementMM unElt) { }`

Appel :

```
CD unCD = new CD(...);  
BiblioMM bib = new Biblio(...);  
bib.ajouterElt(unCD);
```

Vérification :

`unElt = unCD ?...`

Sous-types et retour de valeur

- De manière similaire au passage de paramètres, lorsqu'une méthode retourne une référence à un objet, les mêmes règles s'appliquent : **le type de l'objet retourné doit être compatible avec le type de retour déclaré**

- Exemple :

Définition :

```
public ElementMM obtenirCD(String titre) {  
    CD unCD = new CD(...);  
    return unCD;  
}
```

OK car CD hérite d'ElementMM

Notion de polymorphisme

- On peut distinguer plusieurs types de polymorphisme (= *plusieurs formes*) :
 - **polymorphisme de données** :
 - d'inclusion : un même variable peut être utilisée pour référencer des objets de types différents liés par une relation d'héritage
 - paramétrique : une même classe peut être utilisée pour désigner un ensemble de types (ex. `ArrayList<T>`)
 - **polymorphisme de traitement** : une même signature de méthode peut désigner des séquences d'instruction différentes selon la classe où elle est définie

Polymorphisme d'inclusion

- Exemple avec une variable de type `ElementMM` pouvant désigner différents types d'objets :
 - Soit des instances de `CD`
 - Soit des instances de `DVD`

```
ElementMM element = new CD(...);  
element = new DVD(...);
```

Polymorphisme paramétrique

- Exemple d'une pile paramétrée définie avec un type générique :

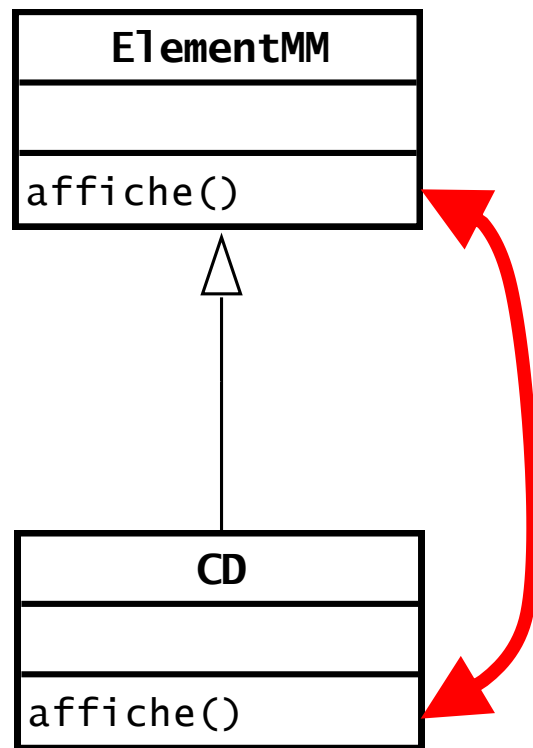
```
public class Pile<E> {  
    private E[]  tableau;  
    public void empiler(E element) { ... }  
    public E     depiler() { ... }  
}
```

Cette classe désigne un ensemble de classes :

- pile d'entiers (Pile<Integer>)
- pile de chaînes de caractères (Pile<String>)
- etc...

Polymorphisme de traitement

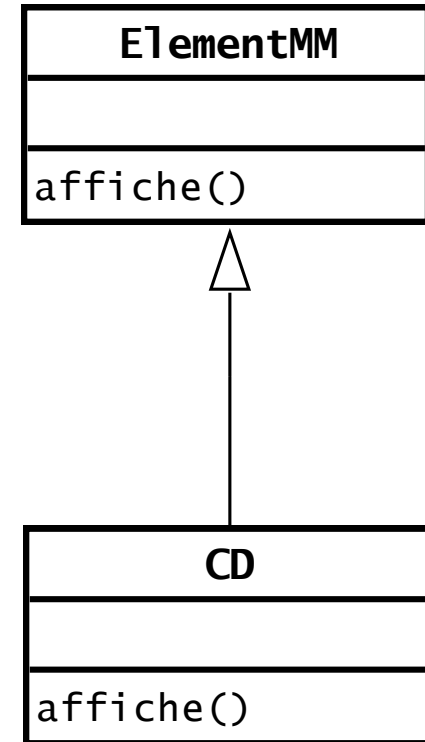
- Exemple de la méthode afficher :



même signature de méthode
mais le code associé
n'est pas le même

Résolution dynamique des appels

- Supposons qu'il existe une méthode `affiche()` dans `ElementMM` et une autre version, **redéfinie**, dans `CD` :
`ElementMM e = new CD(...);`
`e.affiche();`
→ Quelle version de `affiche()` va être exécutée ?...



Résolution dynamique des appels

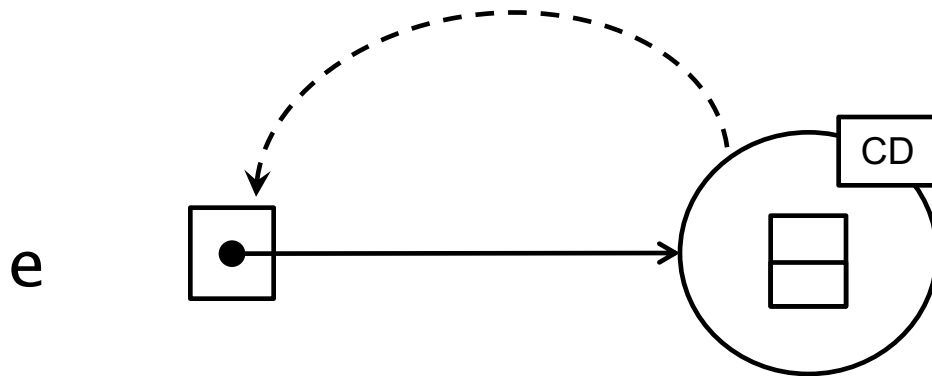
- Pour comprendre ce comportement il est nécessaire de définir la notion de **type statique** et de **type dynamique** :
 - Le type *déclaré* d'une variable représente son type **statique**
 - Le type de l'objet *référéncé* par une variable représente son type **dynamique**→ illustration...

Résolution dynamique des appels

```
ElementMM e = new CD(...);
```

type statique
=
ElementMM

type dynamique
=
CD



Résolution dynamique des appels (suite)

- Avec Java, l'exécution des méthodes se fonde sur le **type dynamique** de l'objet qui reçoit l'appel :

```
ElementMM e = new CD(...);  
e.affiche();
```

type dynamique = CD
donc appel de la méthode
affiche() de la classe CD

- Lors de l'appel d'une méthode Java, le type de l'objet référencé **est plus important** que le type de la référence...

Résolution dynamique des appels (suite)

- Comment le compilateur peut-il déterminer la version de la méthode à appeler ?...
→ en fait le compilateur **ne peut pas le savoir** car cela peut dépendre de valeurs fournies à l'exécution

- Exemple :

```
public class Test {  
    public static void main(String[] args) {  
        Object o = null;  
        if (args[0].equals("1"))  
            o = new Integer(1);  
        if (args[0].equals("2"))  
            o = new String("A");  
        System.out.println(o);  
    }  
}
```

Quelle version de toString sera appelée ?... Celle d'Integer, de Boolean ou bien celle d'Object ?...
→ Cela dépend de la valeur fournie à l'exécution...

```
> java Test 1  
1  
> java Test 2  
A
```

Résolution dynamique des appels (suite)

- Lors qu'il rencontre un appel de méthode, le compilateur génère des instructions (tests) permettant une recherche de la méthode lors de l'exécution (*liaison dynamique*) qui s'effectue en plusieurs étapes :
 1. Détermination de la classe de l'instance référencée par la variable
 2. Recherche de la méthode dans cette classe
 3. Si la méthode est absente, recherche dans la super-classe jusqu'à remonter à la classe Object

Cas des appels de méthodes de classe

- Les méthodes de classes, même si elles possèdent le même nom, ne sont pas redéfinissables car leur appel (`nomDeClasse.methodeDeClasse()`) est lié à la classe :

```
public class A {  
    public static void m() {}  
}  
public class B extends A {  
    public static void m() {}  
}
```

Cas des appels de méthodes de classe (suite)

- Cependant il est possible d'appeler une méthode de classe à partir d'une instance

```
A unObjA = new B();  
unObjA.m(); // quelle méthode m est appelée ?...
```

- Contrairement aux méthodes d'instance, l'appel d'une méthode de classe est déterminé par le type **statique** de la référence appelée
- Ici le type statique de `unObjA` est `A` (type donné lors de la déclaration de la référence) donc la méthode de classe appelée est la méthode `m` de la classe `A`.

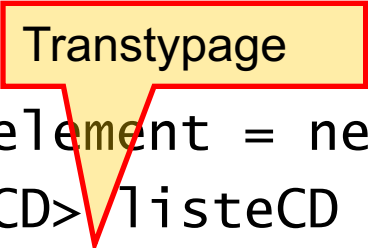
Transtypage dans une hiérarchie de classes

- On a pu constater qu'une variable d'un type T pouvait contenir l'adresse d'un objet d'un type T'
- Il peut parfois être nécessaire d'indiquer au compilateur la nature réelle de l'objet référencé par une variable
- Exemple :

```
ElementMM element = new CD(...);  
ArrayList<CD> listeCD = new ArrayList<CD>();  
listeCD.add(element); // ERREUR à la compilation
```

Pourtant l'objet référencé par `element` est bien du type `CD` !...
Mais le compilateur ne vérifie que les types statiques !...

Transtypage dans une hiérarchie de classes (suite)

- Dans ce cas-là, la solution est d'effectuer un **transtypage** (*cast*) pour indiquer au compilateur que l'objet est réellement du type attendu...
- Exemple : 

```
ElementMM element = new CD(...);  
ArrayList<CD> listeCD = new ArrayList<CD>();  
listeCD.add((CD) element); // OK...
```
- Un transtypage s'effectue en faisant précéder une variable par le type souhaité entre () ...
- Cela indique au compilateur qu'il doit prendre en compte le type indiqué par le transtypage et pas le type statique de la variable...

Transtypage dans une hiérarchie de classes (suite)



- **ATTENTION !....** Si, à l'exécution, la machine virtuelle détecte que le type dynamique de la variable ne correspond pas du type indiqué par le transtypage, l'exécution s'arrête (levée d'une exception)
- Exemple :

```
ElementMM element = new DVD(...);  
ArrayList<CD> listeCD = new ArrayList<CD>();  
listeCD.add((CD) element); // OK à la compilation
```


Mais erreur à l'exécution : ClassCastException !... Car le type dynamique d'element est DVD alors que celui du transtypage est CD !...

Transtypage dans une hiérarchie de classes (suite)

- Avec Java :
 - Le **compilateur** (javac) contrôle les violations de **type statique**
 - La machine virtuelle (java), à **l'exécution**, contrôle les violations de **type dynamique**

```
ElementMM e = new CD(...);
```

```
DVD dvd = (DVD) e;
```



compilation OK
mais pb à
l'exécution !...

Transtypage dans une hiérarchie de classes (suite)

- Concernant le transtypage, il faut savoir que le compilateur exerce un certain nombre de contrôle pour déterminer si son utilisation ne conduit pas à des utilisations incohérentes...

- Exemples :

```
Integer i = (Integer) "bonjour";
```

Problème à la compilation
car une référence de type
Integer ne pourra jamais
référencer une instance de
type String

```
Object o = new String("bonjour")
```

```
String s = (String) o;
```

OK car une référence de
type Object peut
référencer une instance
de type String

Le cas particulier de la classe Object

- L'architecture retenue les classes Java est d'avoir une **unique classe** racine (la classe Object) dont héritent toutes les autres classes. Elle représente la **base commune** à l'ensemble des classes.
- Ce choix permet de fournir un **comportement par défaut** à toutes les classes Java.
- Elle contient notamment la définition des méthodes :
 - `public boolean equals(Object o)`
 - `public String toString()`

Le cas particulier de la classe `Object` (suite)

- Par exemple, la méthode `toString()` est définie pour afficher le nom de la classe de l'instance sur laquelle elle s'exécute suivi de l'adresse de cette instance en mémoire

```
Point p = new Point(2,3);
```

```
System.out.println(p.toString()) → Point@3ef5600
```

- Si on redéfinit cette méthode dans la classe `Point` alors c'est la nouvelle version qui sera appelée :

```
System.out.println(p.toString()) → [x=2,y=3]
```

Héritage et polymorphisme paramétrique

- Un problème fréquemment rencontré avec les conteneurs est le cas où, bien que B soit un sous-type de A, l'écriture du code suivant est cependant impossible :

```
List<A> listeA = new ArrayList<B>();
```

- Pourquoi ?... Autoriser cette affectation permettrait d'insérer dans la liste d'éléments B des éléments sous-type de A (ex. C sous-type de A) ou même des éléments de type A.
- Pour permettre une telle affectation une notation existe...

- `List` : définit une liste d'objets pouvant être de types différents
- `List<T>` : définit une liste d'objets devant être tous du type `T`
- `List<? extends T>` : ajoute une limite **supérieure** au type des objets de la liste qui doit être `T` ou un **sous-type** de `T`

`List<? extends Number> li = new ArrayList<Integer>();`
car `Integer` est un sous-type de `Number`

- C'est cette notation qu'il faut utiliser lorsqu'on veut passer une liste d'un sous-type de celui attendu.

Héritage et polymorphisme paramétrique (suite)

- En contrepartie, la nouvelle référence à la liste ne permet pas de la modifier... Un exemple est donné ci-dessous :

```
class A {  
    static void m1(ArrayList<A> listeA) { }  
    static void m2(ArrayList<? extends A> listeA) { }  
}  
  
class B extends A {  
    public static void main(String[] args) {  
        ArrayList<B> l1 = new ArrayList<B>();  
        ArrayList<? extends A> l2 = l1; // OK  
        l2.add(new B()); // erreur !...  
        A.m1(l1); // erreur !...  
        A.m2(l1); // OK...  
    }  
}
```


Droits d'accès et héritage

- Une sous-classe n'a pas de droits d'accès aux membres (attributs et méthodes) privés hérités.
→ Passage obligatoire par les accesseurs...
- Si une classe souhaite offrir à ses sous-classes l'accès direct à un membre, elle peut le déclarer `protected`
- Cependant l'utilisation du mot-clé `protected` nuit au principe d'encapsulation
→ Donc à utiliser avec modération...

Pour résumer...

- Une instance appartient à **une seule classe** mais peut avoir **plusieurs types**
- Une variable d'un type T peut référencer des objets d'un sous-type de T
- Le polymorphisme permet de choisir dynamiquement (lors de l'exécution) la méthode d'instance à exécuter et il utilise le **type dynamique** de la référence
- Cependant l'appel des méthodes de classe est fondé sur le **type statique** de la référence
- Le **transtypage** permet, dans certains cas, des affectations (ou des passages de paramètres) si le type dynamique correspond au type statique attendu
- La classe `Object` contient des méthodes (version par défaut) héritées par l'ensemble des classes Java