

Encapsulation et héritage

Programmation objet - I

L2 informatique

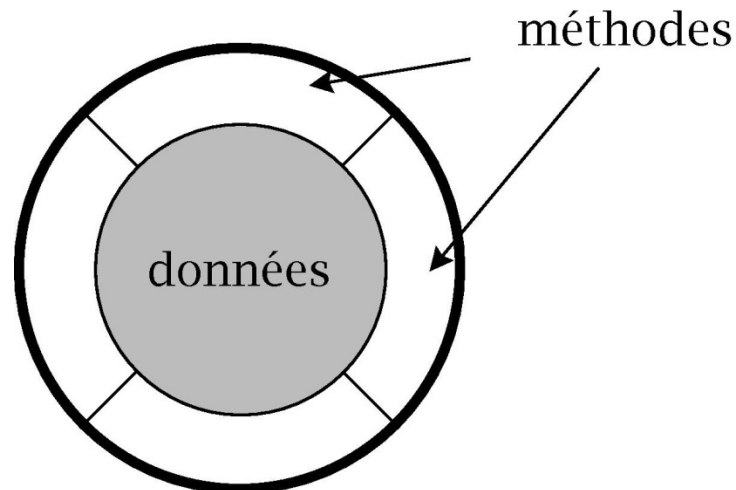
F. Bertrand

Au programme d'aujourd'hui...

- L'encapsulation en programmation objet
- Attributs et méthodes de classe (`static`)
- Définition et utilisation de l'héritage
- Héritage et construction d'objets

Le principe d'encapsulation

- Principe d'encapsulation :
« Séparer les choses qui **peuvent changer** de celles qui **ne doivent pas...** »
- **Seules les méthodes** ont accès aux attributs



Le principe d'encapsulation (suite)

- Pourquoi ?...

1. La plus importante : masquer les détails de mise en œuvre pour être libre de les modifier...
2. Protéger la cohérence des données (dépendance)
3. Faciliter la mise au point des programmes (le code modifiant les attributs se situe uniquement dans les méthodes de la classe)
4. Simplifie la compréhension du fonctionnement de l'objet (moins de membres visibles)

- Deux exemples...

Le principe d'encapsulation (suite)

- Supposons qu'on souhaite créer une classe `NombreComplexe` en utilisant, dans un premier temps, une représentation cartésienne publique :

```
public class NombreComplexe {  
    public double x, y;  
    ...  
}
```

Pour l'exemple !!!...

$$z = x + iy$$

- Un utilisateur de la classe peut écrire :

```
NombreComplexe c = new NombreComplexe(0,1);  
System.out.println(c.x);
```

Le principe d'encapsulation (suite)

- Maintenant, supposons qu'on s'aperçoive que pour les calculs à réaliser, une représentation trigonométrique soit mieux adaptée.

```
public class NombreComplexe {  
    public double rho, theta;  
    ...  
}
```

$$z = \rho \cdot (\cos\theta + i \cdot \sin\theta)$$

- Le code précédent ne se compile plus :

```
NombreComplexe c = new NombreComplexe(0,1);  
System.out.println(c.x); // ERREUR !...
```

Le principe d'encapsulation (suite)

- La bonne approche est de qualifier les attributs **privés** (private) et de fournir des méthodes accesseurs...(encore appelées get/set)

```
public class NombreComplexe {  
    private double x, y;  
    public double donnePartieReelle() {  
        return this.x;  
    }  
    public double donnePartieImaginaire() {  
        return this.y;  
    }  
}
```

Respect du 1^{er} principe :
masquer la mise en œuvre

Le principe d'encapsulation (suite)

- L'utilisation devient alors :

```
NombreComplexe c = new NombreComplexe(0,1);  
System.out.println(c.donnePartieReelle());
```

- En cas de **changement de représentation interne**, le programmeur (utilisateur de la classe) **n'en subit aucune conséquence**, seul le développeur de la classe `NombreComplexe` doit réécrire le code des méthodes accesseurs pour calculer les parties réelle et imaginaire à partir de la représentation trigonométrique...

Le principe d'encapsulation (suite)

- Dans le cas précédent, cela donnerait...

```
public class NombreComplexe {  
    private double rho, theta;  
    public double donnePartieReelle() {  
        return Math.cos(this.theta) * this.rho;  
    }  
    public double donnePartieImaginaire() {  
        return Math.sin(this.theta) * this.rho;  
    }  
}
```

Le principe d'encapsulation (suite)

- Autre exemple : la surface du rectangle...

```
public class Rectangle {  
    private int largeur, longueur, surface;  
    ...  
    public void changeLargeur(int lr) {  
        this.largeur = lr;  
        this.calculeSurface(); // recalcul de la surface  
    }  
    public void changeLongueur(int lg) {  
        this.longueur = lg;  
        this.calculeSurface(); // recalcul de la surface  
    }  
    private void calculeSurface() {  
        this.surface = this.largeur * this.longueur;  
    }  
}
```

Cette
méthode
est à usage
« interne »
(à la
classe)

Respect du 2^{ème} principe :
protéger la cohérence interne

Le principe d'encapsulation (suite)

- Si on qualifie les attributs de `Rectangle` en `public`, il devient **impossible** de s'assurer que la surface du rectangle correspond à sa largeur et sa hauteur :

```
Rectangle r = new Rectangle(5,2);  
r.largeur = 1; // possible si les  
                // attributs sont publics  
System.out.println(r.donneSurface());
```

La valeur affichée est erronée !...

Le principe d'encapsulation (suite et fin)

- Quelques règles à respecter :
 1. Toujours interdire aux autres classes l'accès aux attributs en les qualifiant **privés** (`private`)
 2. Les méthodes, manipulant ces attributs, sont, sauf raison particulière, **publiques** (`public`)
 3. Pour accéder aux valeurs des attributs, fournir des méthodes permettant de lire et de modifier les valeurs de ces attributs (**méthodes accesseurs**)
 4. Si des attributs ne doivent pas être modifiés après la création d'un objet (immutable), alors ne fournir que des méthodes permettant de lire uniquement les valeurs.

Les attributs de classe

- Ces membres se distinguent des attributs d'**instance** par le fait qu'ils n'appartiennent pas à un objet mais sont rattachés directement à la classe.
- Lorsqu'on définit un attribut de classe, il est donc :
 - **Commun** (accessible) à toutes les instances de cette classe.
 - Présent qu'à **un seul** emplacement mémoire, initialisé lors du chargement de la classe (début du programme).

Les attributs de classe (suite)

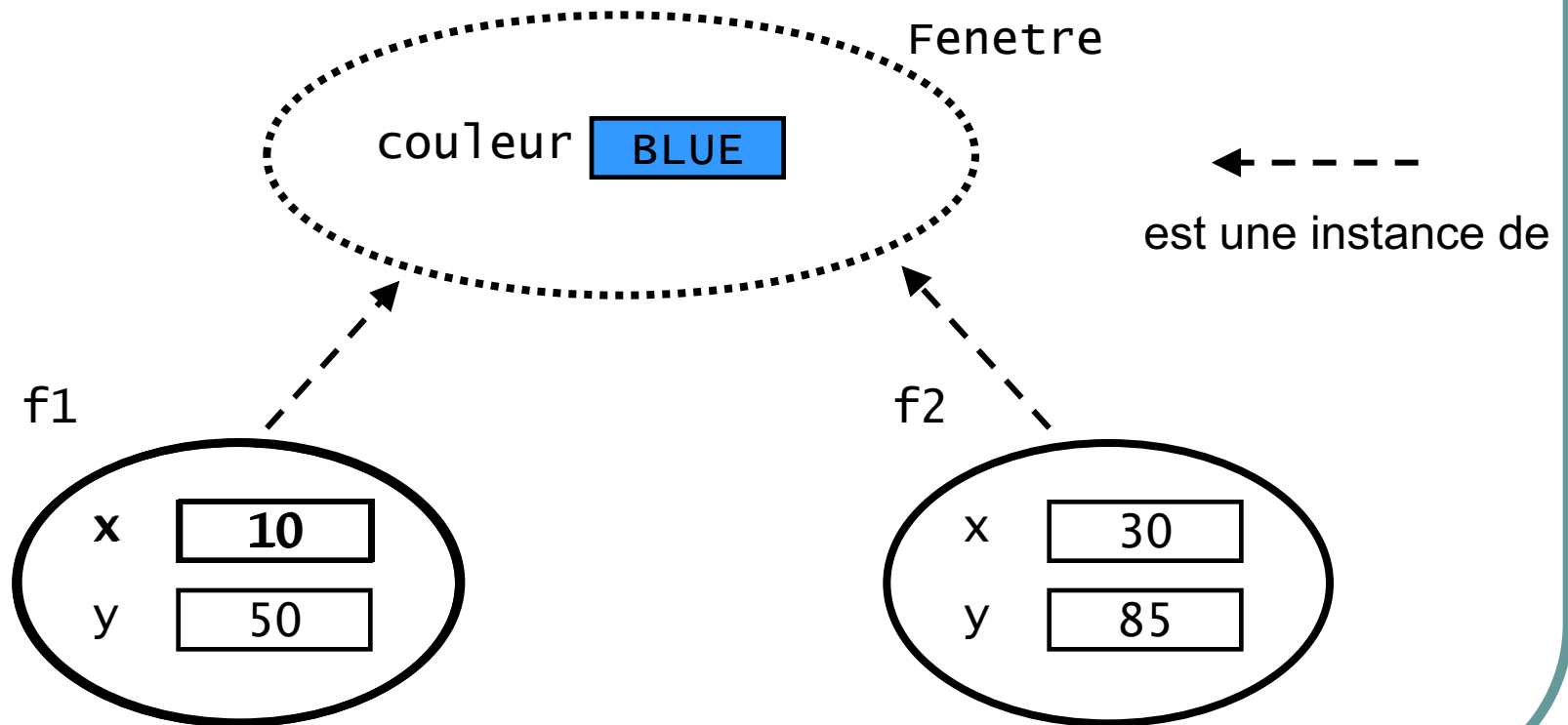
- Pour définir un attribut de classe on doit préfixer sa définition avec le mot-clé `static`
- Cet attribut est initialisé avant même l'exécution de la méthode `main` (lors du chargement de la classe).

```
public class Fenetre {  
    private static Color couleur = Color.BLUE;  
    private int x, y;  
    ...  
}
```

Exemple d'utilisation
du type énuméré
`java.awt.Color...`

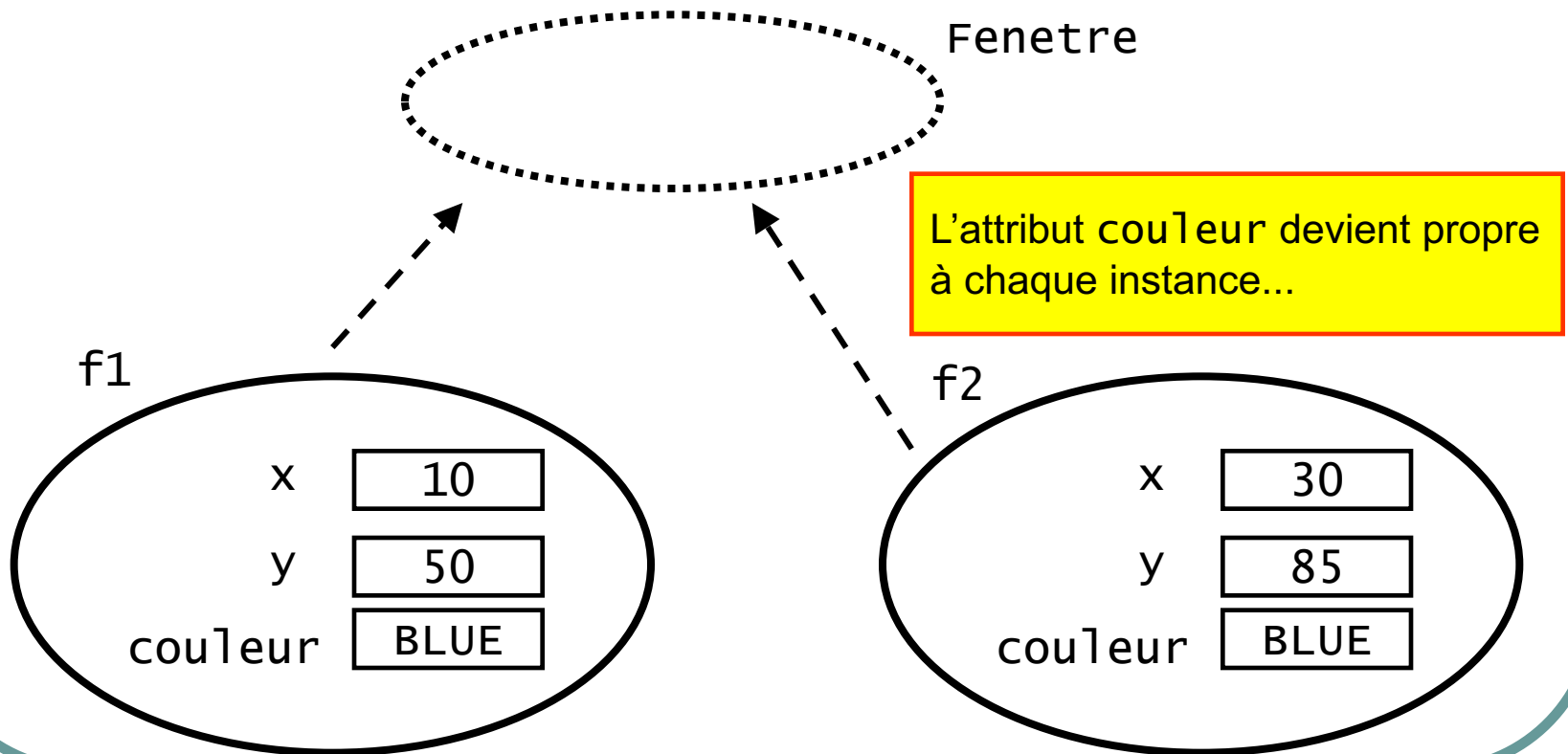
Les attributs de classe (suite)

- Représentation en mémoire (création de 2 fenêtres f1 et f2)

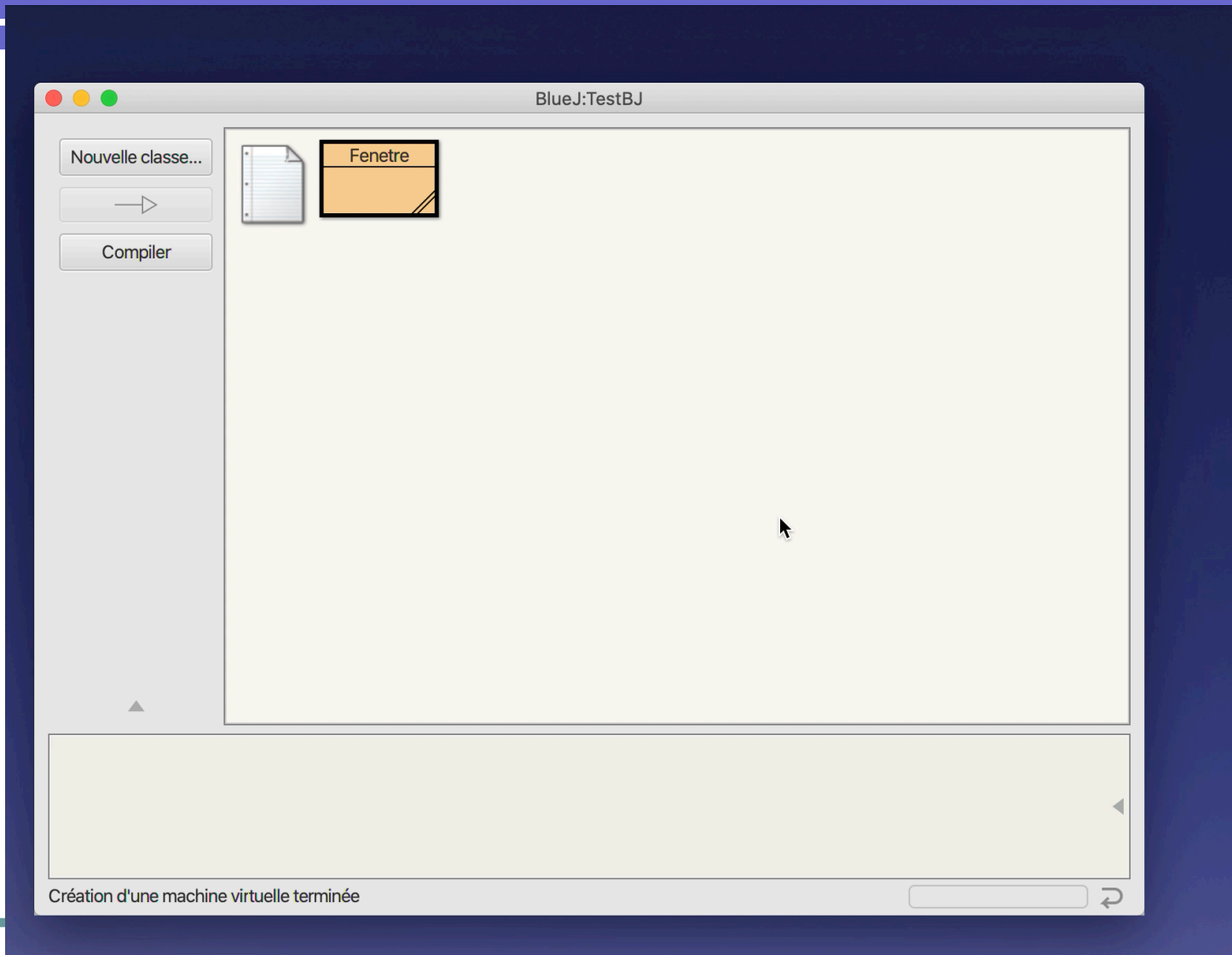


Les attributs de classe (suite)

- Représentation en mémoire avec couleur défini comme un attribut d'instance (pas de static)



Les attributs de classe (suite)



Les attributs de classe (suite)

- Contrairement aux variables/attributs d'instance qui doivent être initialisé(e)s dans le (ou les) constructeur(s), les variables de classe doivent être initialisées lors de leur déclaration :

```
public class BienDeConsommation {  
    private static TypeTVA taux = TypeTVA.NORMAL;  
    private static ArrayList<Vendeur> liste =  
        new ArrayList<Vendeur>();  
    ...  
}
```

Les attributs de classe (fin)

- Récapitulatif des différentes catégories existantes de variables :
 - Variables locales (déclarées à l'intérieur d'une méthode)
 - Paramètres (s'utilisent comme des variables locales)
 - Variables d'instances (attributs d'un objet)
 - Variables de classes (`static`)

Les méthodes de classe

- Une méthode qui peut s'exécuter **sans être associée à une instance**, est appelée **méthode de classe**
- De ce fait, l'utilisation de `this` dans une méthode de classe est **interdit** !...
- Une méthode de classe est définie en la préfixant avec le mot-clé `static`
- Elle s'exécute directement sur une classe et n'a accès **uniquement qu'aux attributs de classe**

Les méthodes de classe (suite)

- Exemple la méthode `main` :

```
public class MaClasse {  
    ...  
    public static void main(String[] args) {  
        ...  
    }  
}
```

- Lors de l'exécution, la machine virtuelle (java) appelle directement la méthode `main` via sa classe d'appartenance

`java MaClasse` → appel de `MaClasse.main()`

Les méthodes de classe (suite)

- Autre exemple, les méthodes de la classe `Math` : `Math.sqrt(4)` ou `Math.cos(3.14)`
- Ces méthodes ont été définies comme méthodes de classe car les types primitifs n'ont pas de méthodes associées...
- Certains langages objet ne manipulent qu'un seul type : le type objet (pas de types primitifs). Dans ce cas, cela donne le code suivant (non autorisé en Java) :
`4.sqrt()` ou `(3.14).cos()`

Les méthodes de classe : exemple d'utilisation

- Exemple : on souhaite compter le nombre d'instances créées pour une classe A...

- Première solution (**fausse**)



```
class A {  
    private int nbInstances;  
    A() { this.nbInstances++; }  
    int nbDeA() { return this.nbInstances; }  
}
```

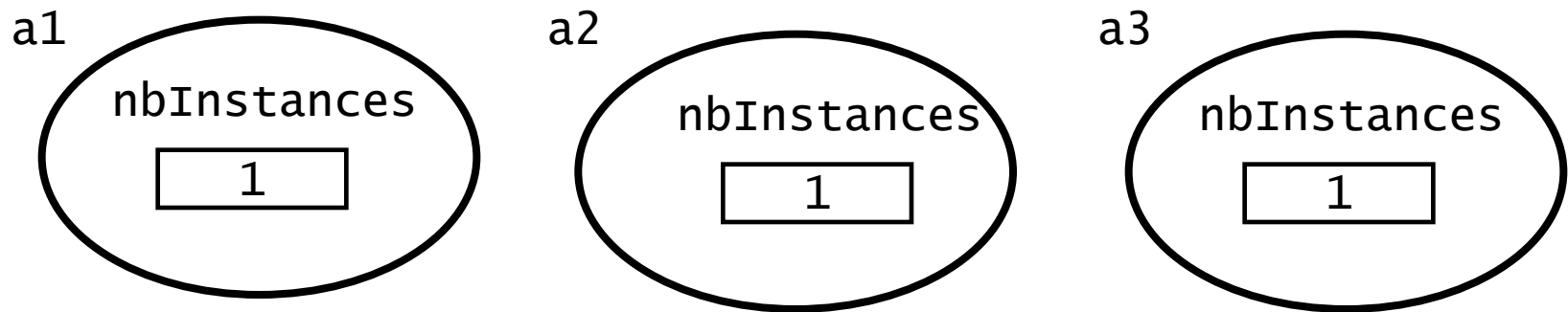
Quelle valeur est affichée ?...

...

```
A a1 = new A(); A a2 = new A();  
System.out.println("Nb = " + a1.nbDeA());
```

Les méthodes de classe : exemple d'utilisation (suite)

- Dans cette première version, chaque instance possède son propre compteur...



- Or on souhaite avoir un compteur **partagé** (commun) par l'ensemble des instances...
- Ceci n'est possible qu'en utilisant une variable de classe (`static`)

Les méthodes de classe : exemple d'utilisation (suite)

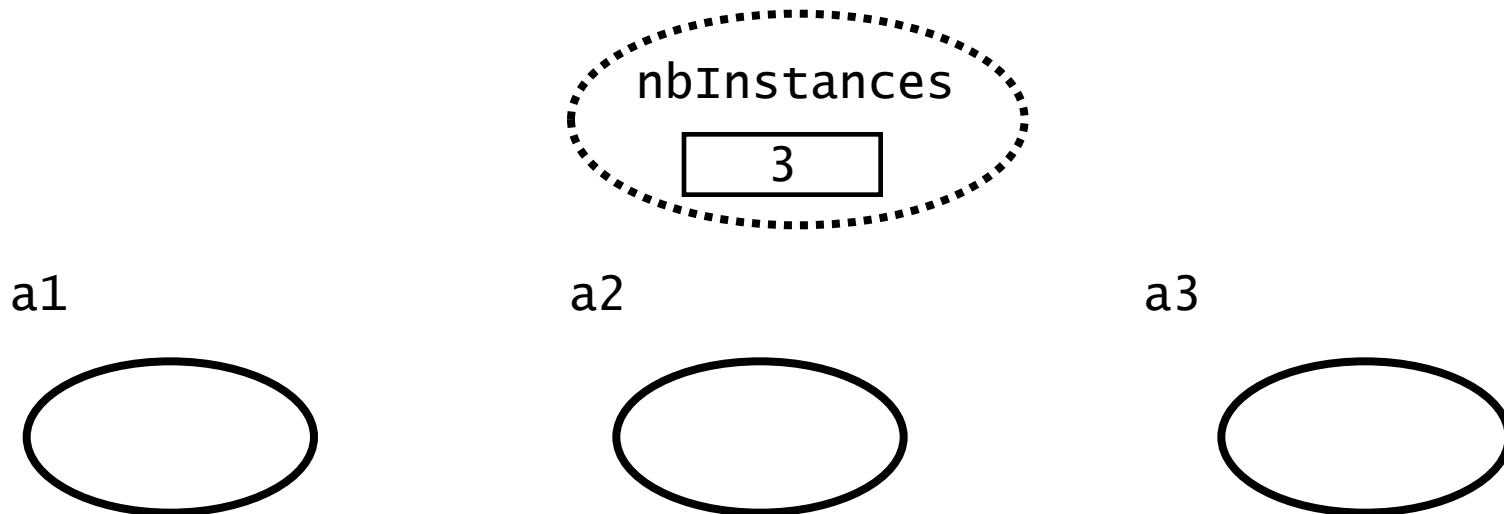
● 2^{ième} solution (OK)



```
class A {  
    private static int nbInstances;  
    A() { A.nbInstances++; }  
    static int nbDeA() {  
        return A.nbInstances;  
    }  
}  
...  
A a1 = new A(); A a2 = new A();  
System.out.println("Nb = " + A.nbDeA());
```

Les méthodes de classe : exemple d'utilisation (suite)

- Dans cette seconde version, l'attribut `nbInstances` est une variable de classe et donc **commune** et **unique** à l'ensemble des instances de A...



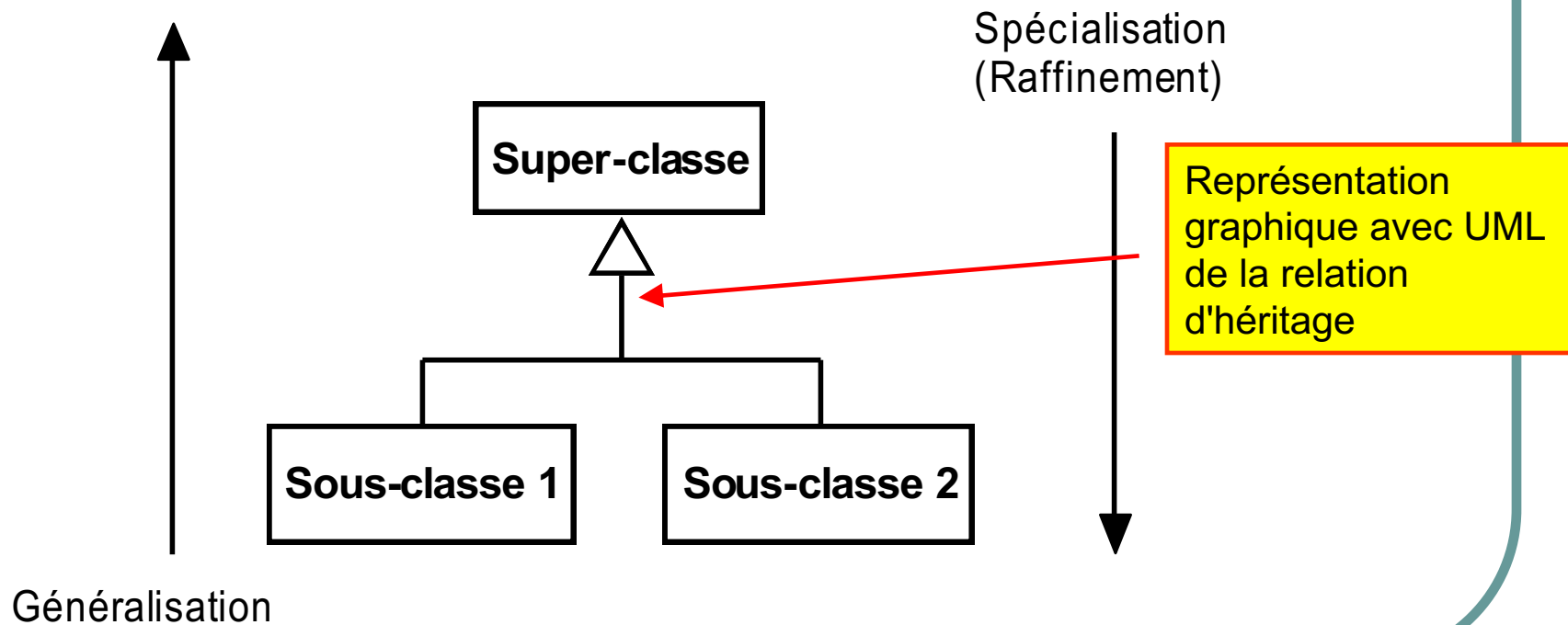
Pour résumer...

- Pourquoi utiliser des variables de classe ?
 - Une **variable de classe** représente une **valeur partagée par toutes les instances** de la classe où elle est déclarée...
 - Une méthode de classe permet d'être appelée sans qu'aucune instance de la classe ne soit créée
 - Exemple : contrôle de l'accès au constructeur
- Mais attention un mauvais emploi de `static` peut amener à programmer **de manière non objet**



Le concept d'héritage

- L'héritage est une technique permettant de **réutiliser** du code existant ou de **factoriser** des éléments de code communs à plusieurs classes



Le concept d'héritage (suite)

- Lorsqu'une classe A hérite d'une classe B, elle va hériter :
 - des attributs de B
 - des méthodes de B (sauf des constructeurs)
- Les attributs et les méthodes de la super-classe sont ainsi disponibles dans la sous-classe **sans qu'on ait besoin de les définir explicitement**
- Les éléments hérités de B **s'ajoutent** (enrichissent) à ceux déclarés dans A.

Le concept d'héritage (suite)

- Les différents liens d'héritage entre les classes forment un graphe orienté *acyclique* appelé **hiérarchie de classes**.
- La relation d'héritage est **transitive**
- Nous nous limiterons à des exemples d'héritage simple (une sous-classe est liée à **une seule super-classe**), Java ne permettant pas l'héritage multiple.

Le concept d'héritage (suite)

- Exemple extrait de la documentation Java :

[Overview](#) [Package](#) **Class** [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

javax.swing

Class JCheckBox

[java.lang.Object](#)

└ [java.awt.Component](#)

└ [java.awt.Container](#)

└ [javax.swing.JComponent](#)

└ [javax.swing.AbstractButton](#)

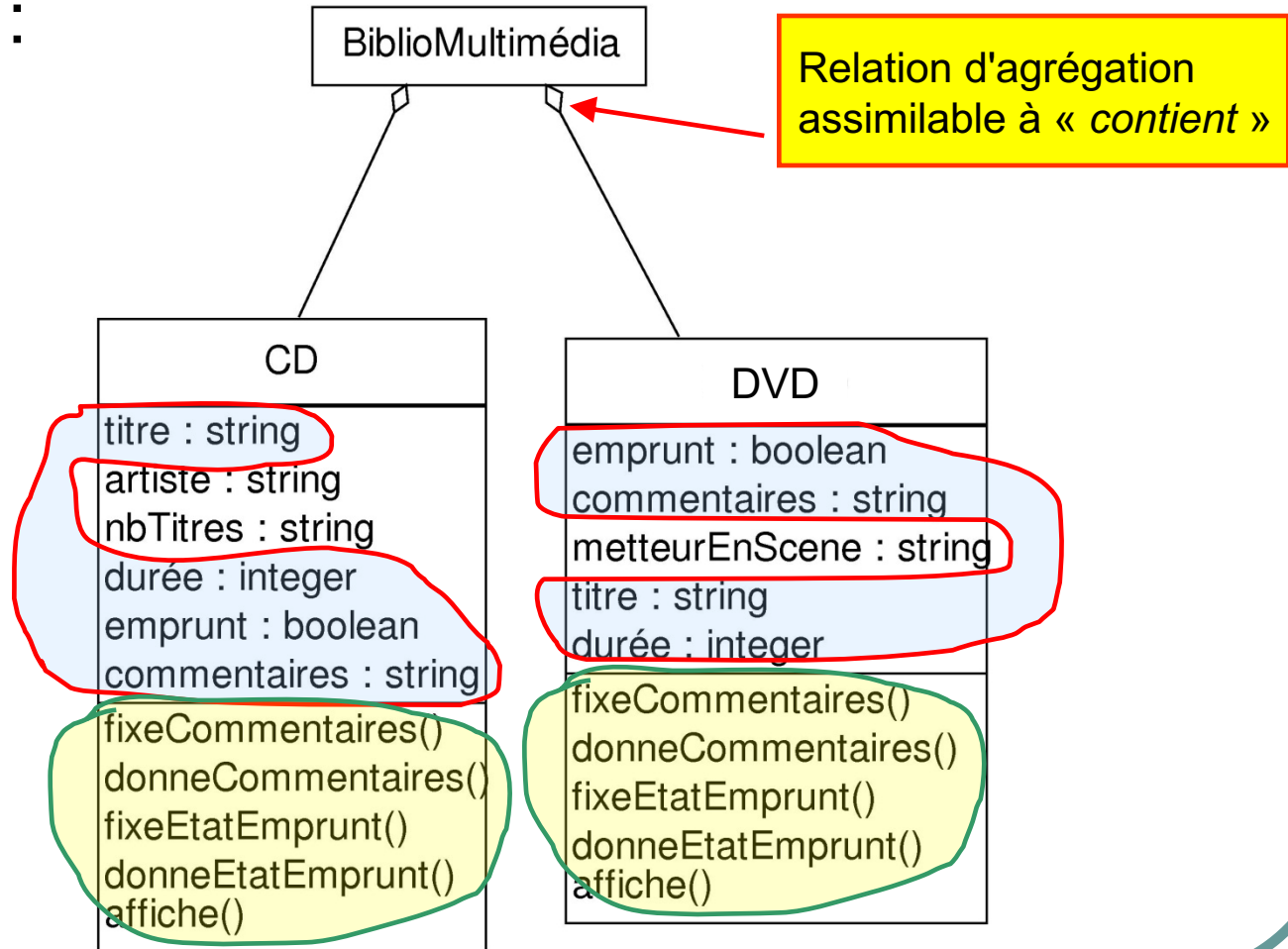
└ [javax.swing.JToggleButton](#)

└ javax.swing.JCheckBox

Hiérarchie
d'héritage

Utilisation de l'héritage

- Exemple :



Utilisation de l'héritage (suite)

- Structure de la classe BiblioMM :

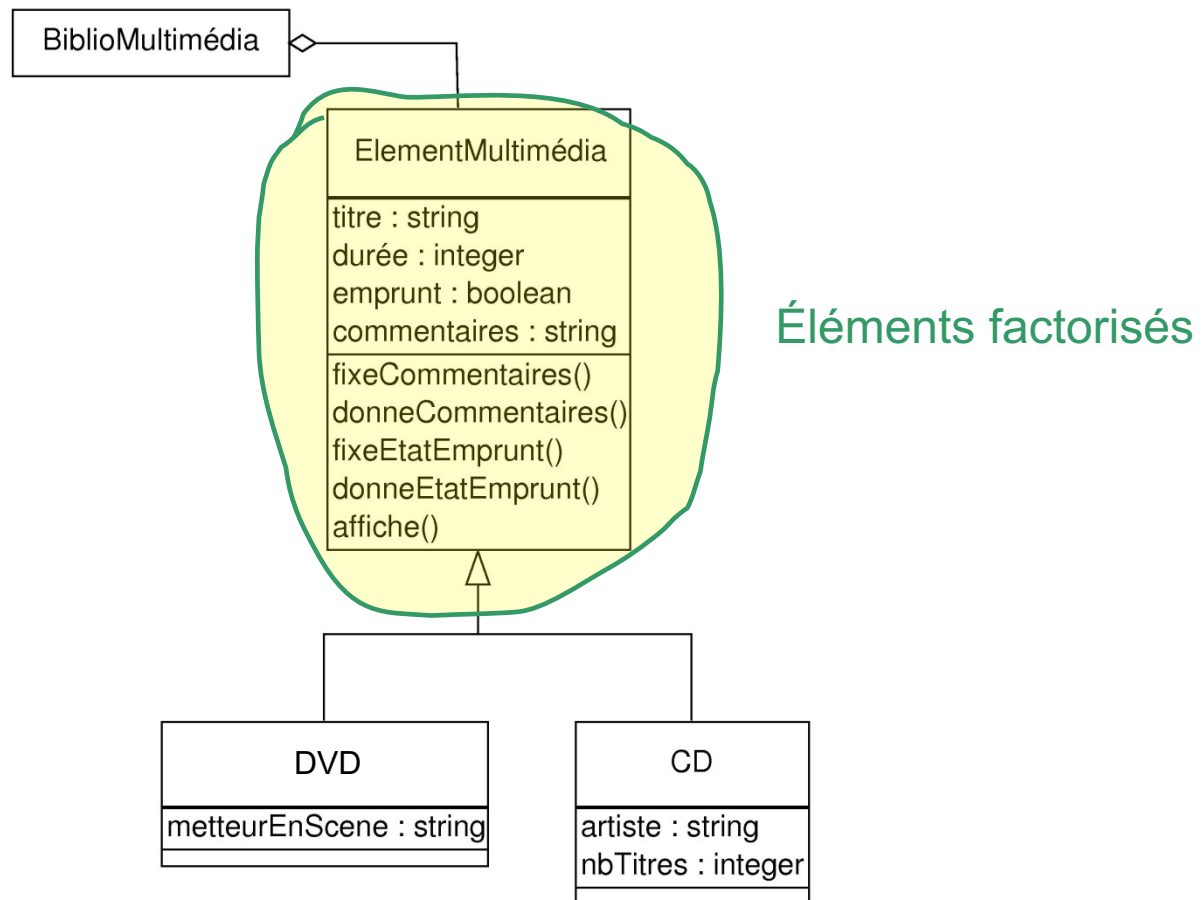
```
public class BiblioMM {  
    private ArrayList<CD> cds;  
    private ArrayList<DVD> dvds;  
  
    public void ajouter(CD unCD) { ... }  
    public void ajouter(DVD unDVD) { ... }  
}
```

Utilisation de l'héritage (suite)

- Une analyse rapide montre que les classes CD et DVD sont très semblables
- Ces similarités peuvent être considérées, au niveau du code, comme de la duplication.
- Cela entraîne :
 - une maintenance plus laborieuse et plus difficile
 - une augmentation des risques d'introduire des erreurs lors de la maintenance du code

Utilisation de l'héritage (suite)

- L'exemple revisité avec la relation d'héritage :



Héritage du type des super-classes

- Nouvelle structure de la classe BiblioMM :

```
public class BiblioMM {  
    private ArrayList<ElementMM> eltsMM;  
}
```
- Comme CD et DVD héritent de ElementMM, ils possèdent ce type
- Ainsi une instance de CD possède 3 types :
 - Object
 - ElementMM
 - CD
- Attention : elle n'est instance que d'une seule classe ! ... CD en l'occurrence

Héritage du type des super-classes (suite)

- En fait une instance possède le type de sa classe et les types de ses super-classes.
- Cela permet :

- De remplacer les deux listes présentes dans BiblioMM par une seule :

`ArrayList<ElementMM> elements;`

- De remplacer les deux méthodes :

`public void ajouter(CD unCD)`

`public void ajouter(DVD unDVD)`

par une seule méthode :

`public void ajouterEltMM(ElementMM unElt)`

Substitution de type

- Le principe de substitution de type est simple : à chaque fois qu'il est nécessaire de fournir un objet d'un type T, il est possible d'utiliser à la place un objet d'un autre type à condition qu'il possède **également** ce type T.
- Si on considère la méthode
`public void ajouterEltMM(ElementMM unElt)`
il est possible, lors de son appel, de lui passer en paramètre une instance de la classe CD car celle-ci possède le type `ElementMM` du fait que CD hérite de `ElementMM`

Représentation de l'héritage avec Java

- La relation d'héritage est exprimée lors de la déclaration d'une classe en précisant sa super-classe avec le mot-clé `extends`.

```
class A extends B { ... } // A hérite de B
```

- Avec Java, **toutes les classes possèdent une super-classe** (`Object`) même si `extends` n'apparaît pas explicitement lors de la déclaration de la classe :

```
class A extends Object  
{ ... }
```

Représentation de l'héritage avec Java (suite)

- L'exemple précédent correspond au code suivant :

Implicitement : extends Object

```
public class ElementMultimedia {  
    private String titre;  
    private int durée;  
    private boolean emprunt;  
    private String commentaires;  
    // constructeur et méthodes...  
}
```


Représentation de l'héritage avec Java (suite)

```
public class DVD extends ElementMultimedia {  
    private String metteurEnScene;  
    // constructeur et méthodes...  
}
```

```
public class CD extends ElementMultimedia {  
    private String artiste;  
    private String nbTitres;  
    // constructeur et méthodes...  
}
```

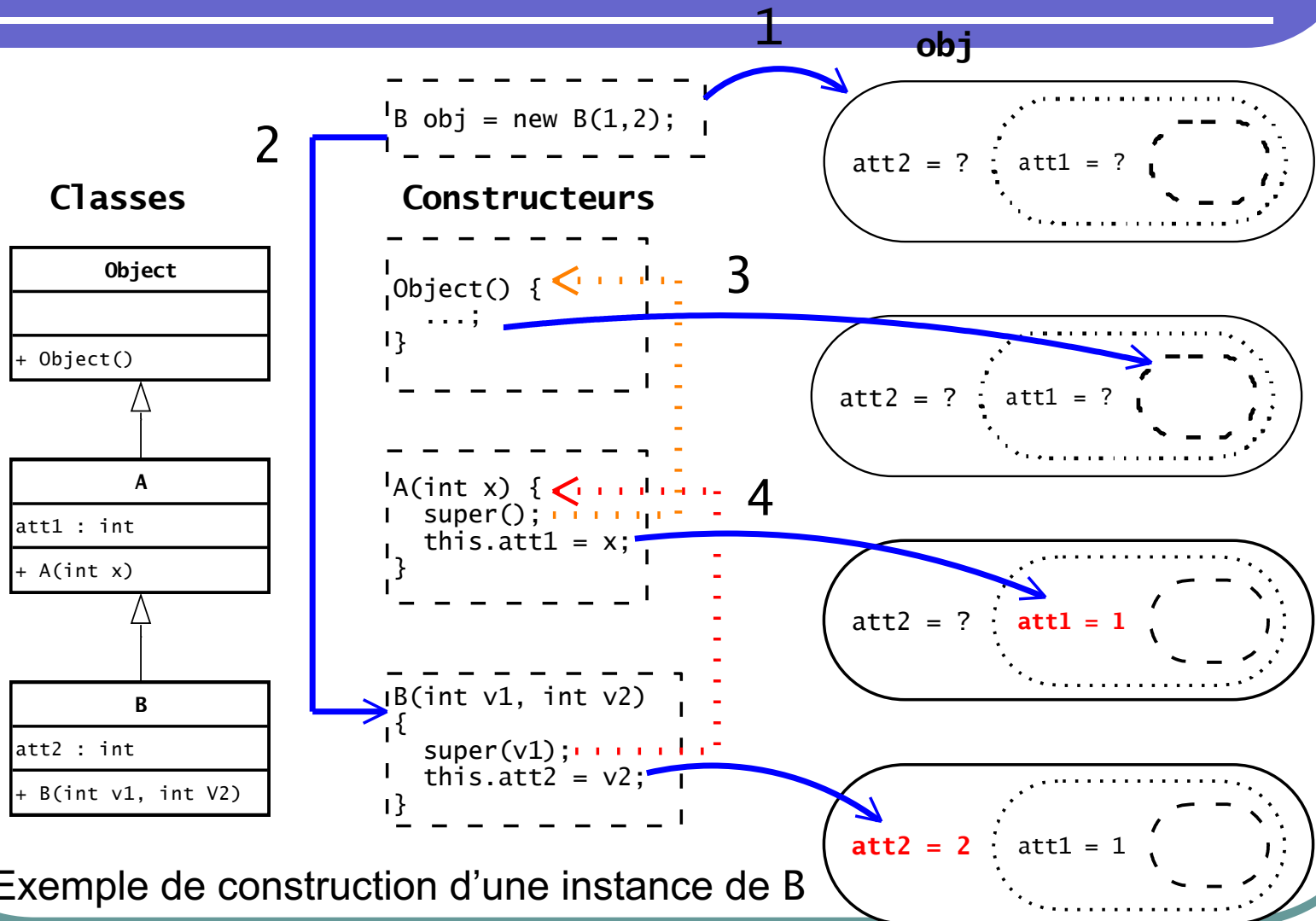
Héritage et constructeurs

- Par principe, **chaque classe se charge de l'initialisation de ses attributs dans son (ou ses) constructeur(s)**, ceci afin d'éviter de dupliquer le code d'initialisation des attributs hérités
- Donc, lors de la définition d'une classe héritant d'une super-classe, le (ou les) constructeur(s) doivent préciser **le constructeur de la super-classe qui doit être appelé** lors de la création d'un objet

Héritage et constructeurs (suite)

- L'initialisation d'une instance :
 1. Débute toujours par l'initialisation des attributs hérités de la classe de base (Object)
 2. Puis se poursuit par l'initialisation des attributs des sous-classes en « descendant » la hiérarchie d'héritage
 3. Pour se terminer par l'initialisation des attributs de la classe à laquelle appartient l'instance

Héritage et constructeurs (suite)



Héritage et constructeurs (suite)

- En reprenant l'exemple précédent, la classe `ElementMultimedia` possède un constructeur :

```
public class ElementMultimedia {  
    private String titre;  
    private int durée;  
    private boolean emprunt;  
    private String commentaires;  
    public ElementMultimedia(String unTitre,int uneDurée) {  
        this.titre = unTitre;  
        this.durée = uneDurée;  
        this.emprunt = false;  
        this.commentaires = "";  
    }  
    //méthodes...  
}
```

Valeurs par défaut

Héritage et constructeurs (suite)


- Le(s) constructeur(s) de la classe DVD doit préciser le constructeur d'ElementMultimedia utilisé via le mot-clé super :

```
public class DVD extends ElementMultimedia {  
    private String metteurEnScene;  
    public DVD(String unMetteurEnScene,  
                String unTitre,  
                int uneDurée)  
  
    {  
        super(unTitre, uneDurée);  
        this.metteurEnScene = unMetteurEnScene;  
    }  
    // méthodes...  
}
```



Cet appel (s'il est présent) doit toujours apparaître en **première ligne** du constructeur!...

Héritage et constructeurs (suite)

- Si aucun appel au constructeur de la super-classe n'est précisé alors un appel vers le *constructeur par défaut* (constructeur sans paramètre) est introduit **automatiquement** :
 - S'il existe OK...
 - S'il n'existe pas → **erreur à la compilation**
- Rappel : 
 - Le compilateur fournit un constructeur par défaut uniquement si la classe ne contient **aucun constructeur**.

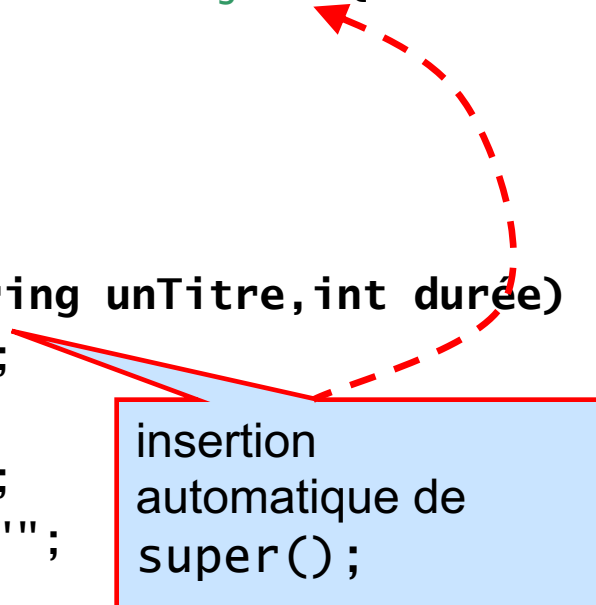
Héritage et constructeurs (suite)

- Ceci est le cas du constructeur de la classe `ElementMultimedia` :

```
public class ElementMultimedia extends Object {  
    private String titre;  
    private int durée;  
    private boolean emprunt;  
    private String commentaires;  
    public ElementMultimedia(String unTitre, int durée) {  
        this.titre = unTitre;  
        this.durée = durée;  
        this.emprunt = false;  
        this.commentaires = "";  
        //méthodes...  
    }  
}
```

Pas de constructeur par défaut dans cette classe

insertion automatique de `super();`



Héritage et constructeurs (suite)

```
public class DVD extends ElementMultimedia {  
    private String metteurEnScene;  
    public DVD(String unMetteurEnScene,  
                String unTitre,  
                int uneDurée)  
  
    {  
  
        this.metteurEnScene = unMetteurEnScene;  
    }  
    // méthodes...  
}
```



Si oubli de `super` alors erreur à la compilation car l'indication `super()` qui sera insérée fera référence au cst de `ElementMultimedia` qui n'existe pas !...

Héritage et constructeurs (suite)

- Ne pas confondre `new` et `super(...)` :
 - `new` :
 - Permet de **construire** un objet
 - Réserve la mémoire et exécute les constructeurs
 - `super(...)` :
 - Représente un appel permettant d'initialiser (grâce à des constructeurs existants) les différents attributs hérités d'un objet
 - L'allocation mémoire est déjà réalisée lors de son appel.

Héritage et constructeurs (suite)

- Test : dans le code ci-dessous, quels sont les problèmes rencontrés à la compilation ?...

```
public class Parente {  
    private int attribut;  
    public Parente(int valeur) {  
        this.attribut = valeur;  
    }  
}
```

```
public class Fille extends Parente {  
    private int att2;  
    public Fille() {  
        this.att2 = 0;  
    }  
    public Fille(int valeur1, int valeur2) {  
        this.att2 = valeur1;  
        this.attribut = valeur2;  
    }  
}
```

Héritage et constructeurs (suite)

- Pour résumer :
 - Si le constructeur d'une classe doit préciser le constructeur de sa super-classe alors il doit le faire explicitement avec `super(paramètres)`
 - Cette indication `super(...)` doit toujours apparaître comme **première instruction** du constructeur.
 - Si elle n'est pas indiquée, le compilateur considère que le constructeur utilise le **constructeur par défaut** de la super-classe (s'il existe !...)
 - L'instruction `super(...)` ne peut apparaître que dans un constructeur.

En conclusion...

- Principe d'encapsulation = toujours qualifier les attributs « `private` » pour masquer l'implémentation et être libre de la modifier
- Lorsqu'une information doit être partagée par l'ensemble des instances d'une classe, utiliser la notion de variable de classe (`static`)
- L'héritage permet une organisation et une factorisation du code (réutilisation)
- Une sous-classe hérite des attributs et des méthodes de sa super-classe
- Une instance d'une classe C peut être utilisée à la place d'une instance d'une super-classe de C (mais pas l'inverse !)
- La construction d'une instance s'effectue en initialisant d'abord les attributs issus de la classe de base puis ceux des sous-classes (chaînage des constructeurs) et enfin ceux de la classe de l'instance
- L'accès à une méthode de la super-classe s'effectue avec le mot-clé `super`