

Programmation et Algorithmique II

Ch.4 – Héritage

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe
2. Polymorphisme et Transtypage
3. Classe `Object`
4. Redéfinition de méthode
5. Liaison dynamique
6. Masquage de variable

3. Encapsulation revisitée

Introduction

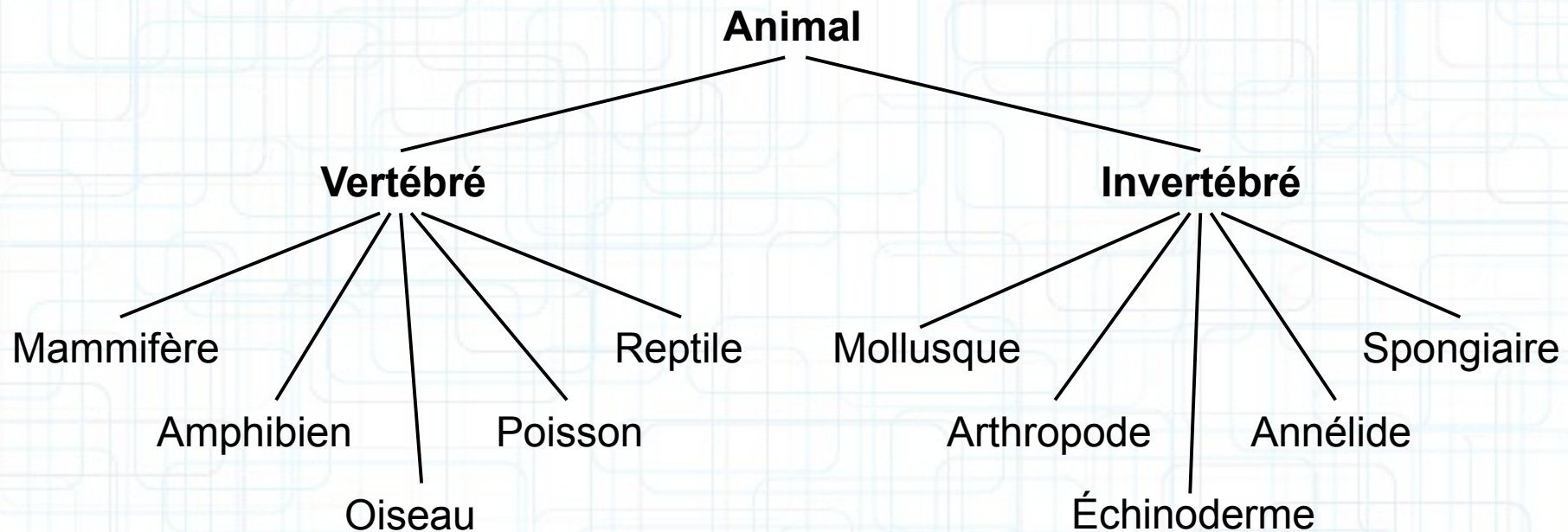
- **Notion d'héritage**

- La notion d'**héritage** est une des notions les plus importantes de la programmation orientée-objet. Elle consiste à créer de nouvelles classes d'objets à partir d'autres classes, par un *processus de spécialisation*.
 - Les classes nouvellement créées **héritent** des données et méthodes de leurs classes « parents ».
 - Extension : De nouvelles données et de nouveaux comportements peuvent être ajoutés.
 - Modifications : Les comportements hérités peuvent être modifiés (**redéfinition de méthodes**).
- Pour le programmeur, l'héritage permet de réduire la duplication de code. La définition rapide de nouvelles classes est possible sans « *copier-coller* ».

Introduction

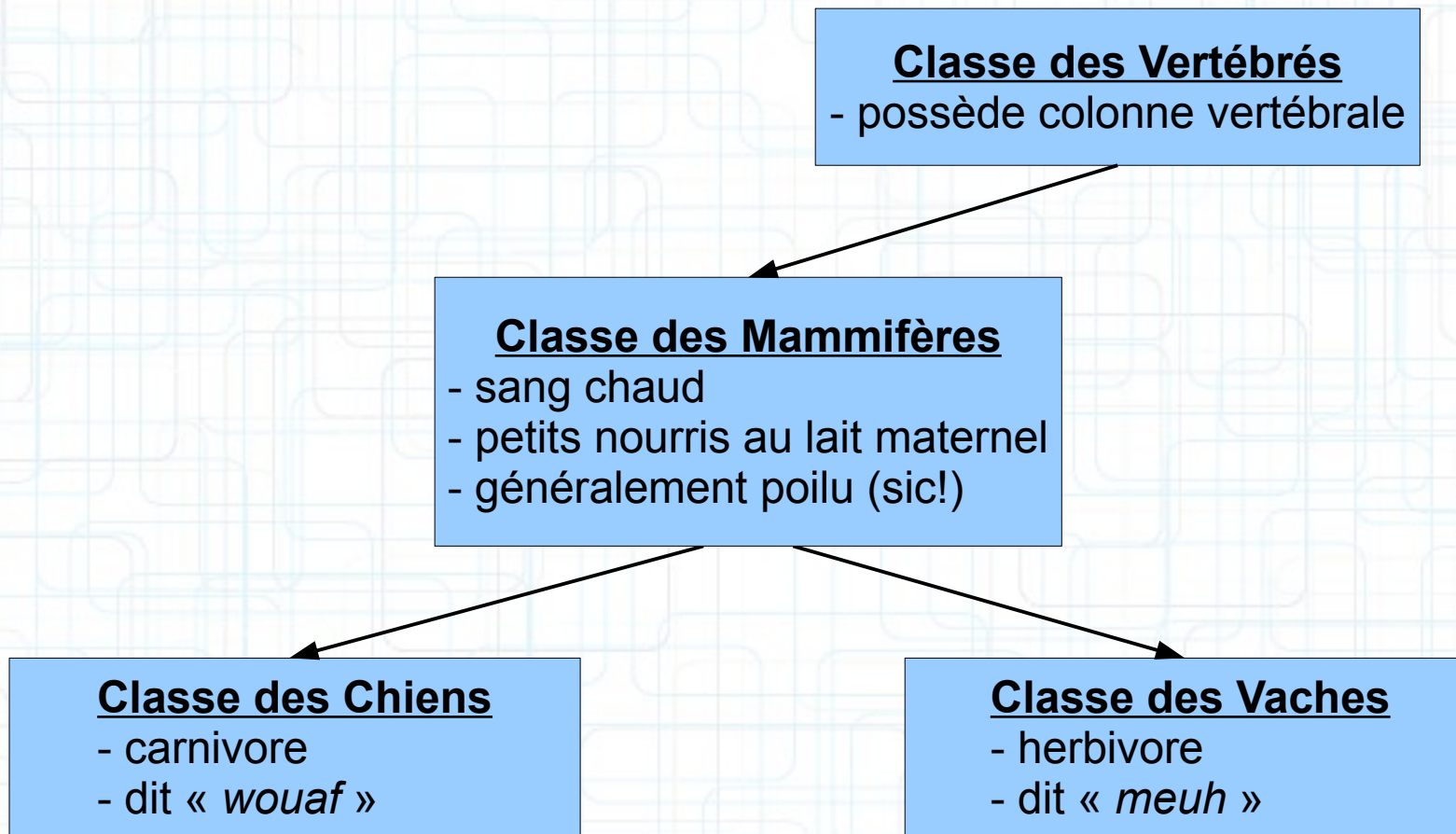
- **Analogie : classification des espèces**

- La notion d'héritage est souvent introduite en faisant l'analogie avec la classification des espèces.



Introduction

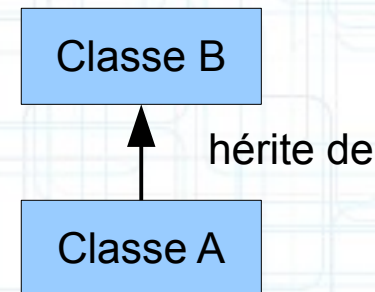
- **Analogie : classification des espèces**



Introduction

- **Héritage en P.O.O.**

- En programmation orienté-objet, l'héritage permet de créer une nouvelle classe **A** à partir d'une classe existante **B**. La nouvelle classe **A** va hériter des définitions de variables et de méthodes de la classe **B**.



- Terminologie

- **A descend de B** ou **A est dérivée de B**.
- **B** est appelée la **classe parent**, la **super-classe** ou la **classe de base** de **A**. On dit également que **B** est plus générique que **A**.
- **A** est appelée une **classe enfant**, une **sous-classe**, ou une **classe dérivée** de **B**. On dit également que **A** est plus spécialisée que **B**.

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

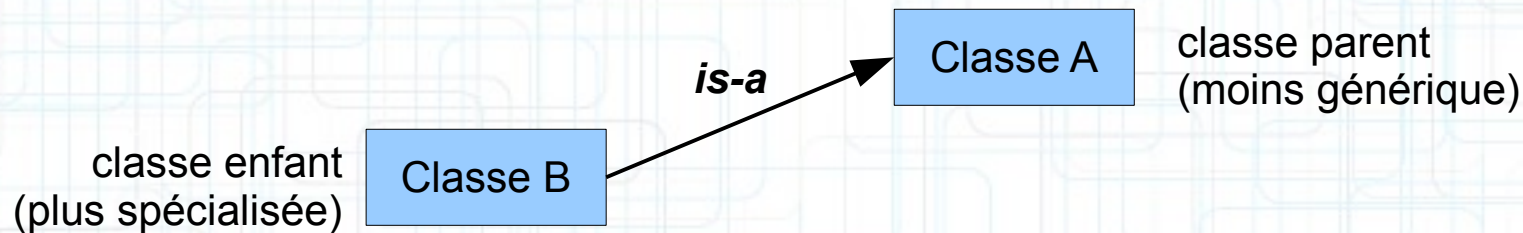
6. Masquage de variable

3. Encapsulation revisitée

Relation « *is-a* »

- **Principe**

- L'héritage définit une relation binaire entre classes. Il s'agit de la relation « ***is-a*** » (que nous lirons « *est-une-sous-classe-de* »).



- Exemple

- Tous les chiens sont des mammifères ; La classe des chiens hérite de certaines propriétés de la classe des mammifères comme par exemple le fait que ses instances ont un sang chaud.
- On dit que la classe des chiens *est-une-sous-classe-de* (est en relation ***is-a*** avec) la classe des mammifères.

Relation « *is-a* »

- Relation d'ordre

- La relation ***is-a*** forme un **ordre** sur l'ensemble des classes car elle satisfait les propriétés suivantes
- **Antisymétrique**
 - $\forall A, B : (A \text{ is-a } B) \text{ et } (B \text{ is-a } A) \Rightarrow A = B$
- **Transitive**
 - $\forall A, B, C : (B \text{ is-a } A) \text{ et } (C \text{ is-a } B) \Rightarrow C \text{ is-a } A$
- **Réflexive**
 - $\forall A : A \text{ is-a } A$

Relation « *is-a* »

- Relation d'ordre

- Il ne s'agit pas d'un ordre total. Il faudrait en effet que toutes les classes soient en relation *is-a*, c-à-d.

$\forall A, B : (A \text{ is-a } B) \text{ ou } (B \text{ is-a } A).$

- L'exemple ci-dessous donne un contre-exemple.

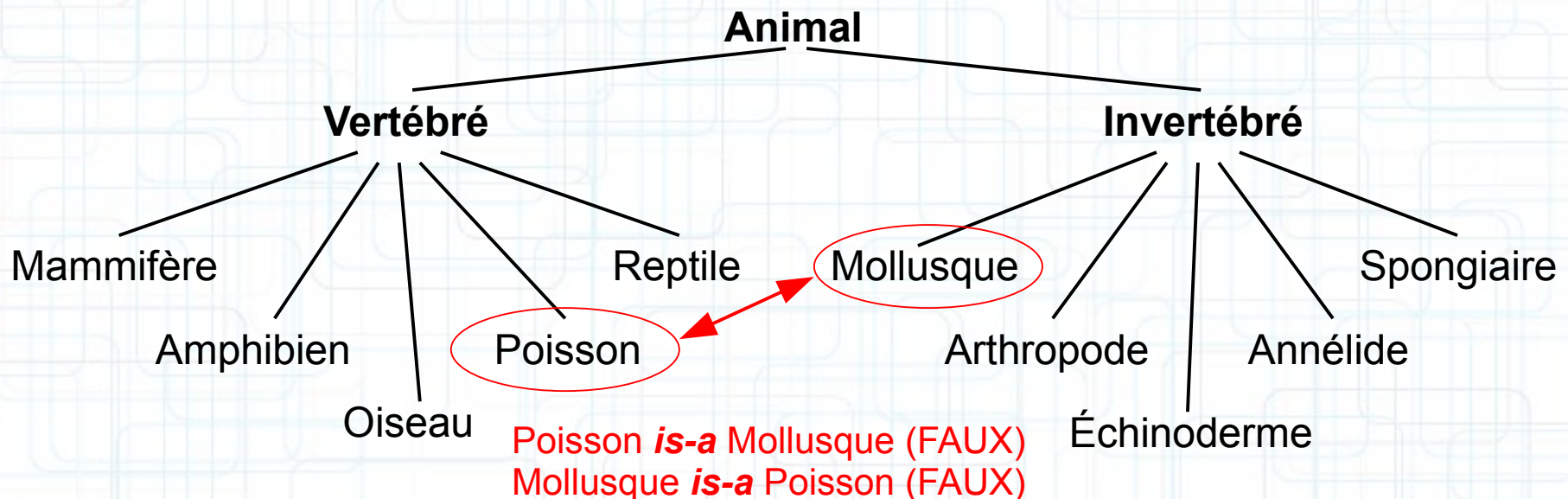


Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

Héritage

- **Mot-clé extends**

- Le mot-clé **extends** permet d'indiquer qu'une classe est définie par héritage d'une autre classe.

- Syntaxe

```
public class nomClasse extends nomClasseParent
{
    corps de la classe
}
```

- Exemple

```
public class Dictionnaire extends Livre
{
    ...
}
```

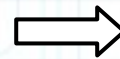

Héritage

- **Héritage en Java**

- L'héritage permet de ré-utiliser du code d'une autre classe.

```
public class Livre {  
    public int numPages;  
  
    public void setNumPages(int numPages) {  
        this.numPages= numPages;  
    }  
  
    public int getNumPages() {  
        return numPages;  
    }  
}
```

```
public class Dictionnaire extends Livre {  
    public int numDefs;  
  
    public void setNumDefs(int numDefs) {  
        this.numDefs= numDefs;  
    }  
  
    public int getNumDefs() {  
        return numDefs;  
    }  
}
```



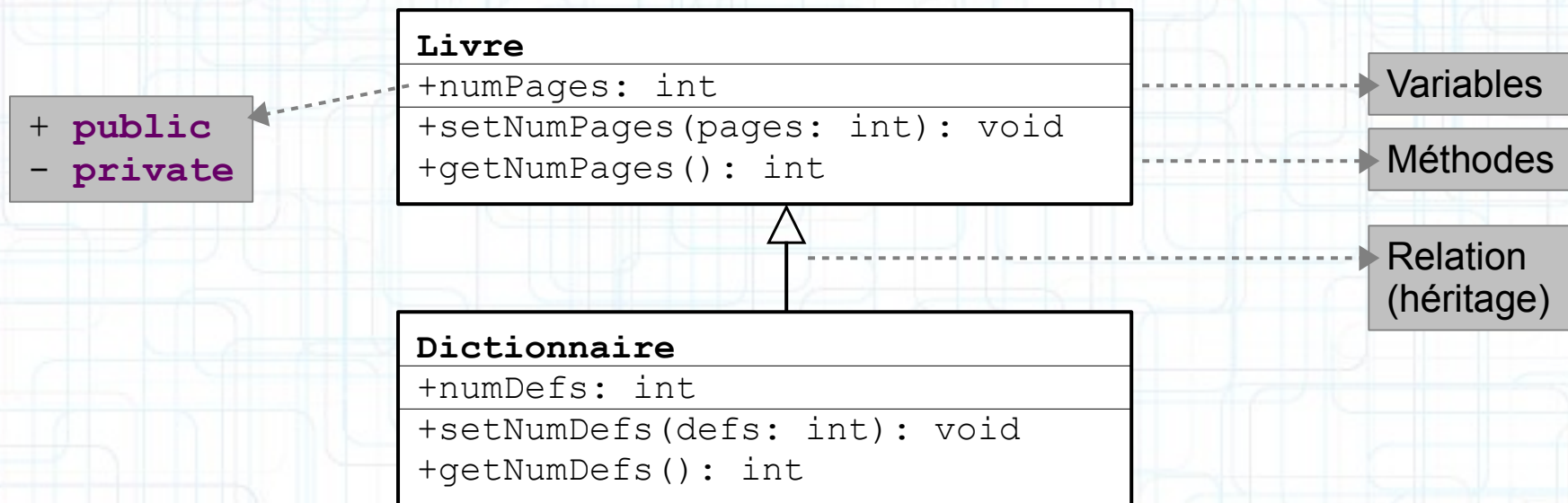
Le code ci-dessous illustre ce que nous aurions dû écrire pour définir la classe Dictionnaire en l'absence d'héritage.

```
public class Dictionnaire {  
  
    public int numPages;  
    public int numDefs;  
  
    public void setNumPages(int numPages) {  
        this.numPages= numPages;  
    }  
  
    public int getNumPages() {  
        return numPages;  
    }  
  
    public void setNumDefs(int numDefs) {  
        this.numDefs= numDefs;  
    }  
  
    public int getNumDs() {  
        return numDefs;  
    }  
}
```

Héritage

- **Diagramme de classes**

- Le **diagramme de classe** représente de façon synthétique les classes d'un programme, leurs membres (variables et méthodes) et les relations entre les classes.



- Ce diagramme permet de visualiser la **hiérarchies des classes** (relation d'héritage) mais également d'autres relations entre classes (composition, agrégation).

Héritage

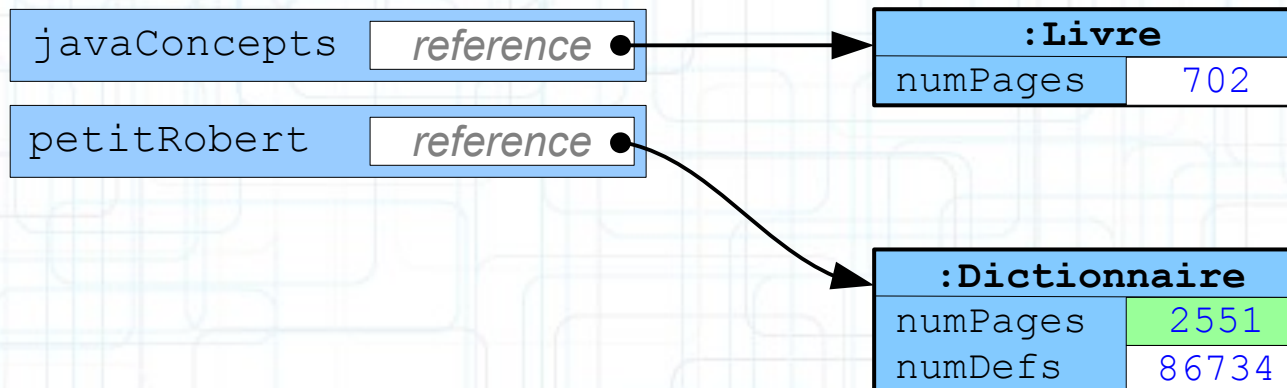
- **Héritage des membres**

- La classe enfant hérite des champs et méthodes définis dans la classe parent.

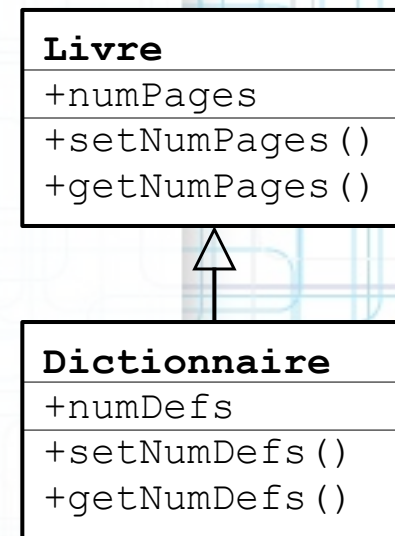
- Exemple

```
Livre javaConcepts= new Livre();  
Dictionnaire petitRobert= new Dictionnaire();  
System.out.println(petitRobert.getNumPages());
```

..... méthode définie dans la classe parent (Livre)



..... variable définie dans la classe parent (Livre)

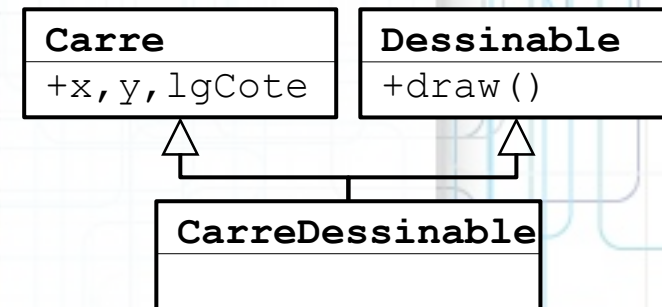


Héritage

- **Héritage simple *versus* multiple**

- **Héritage multiple** : une classe peut hériter de plusieurs parents. Supporté par des langages tels que C++ et python.
 - **Avantage** : permet la modélisation d'objets qui appartiennent à plusieurs catégories.

```
+ C
public class CarreDessinable :
    public Carre, public Dessinable
{
    // ...
}
```



- **Inconvénient** : situations ambiguës. Que se passe-t-il si le même membre est défini dans deux classes parent ?
- **Héritage simple** : une classe ne peut avoir qu'un parent. C'est le cas du langage Java⁽¹⁾.

⁽¹⁾ Nous verrons au [Chapitre V](#) que cette limitation peut être partiellement contournée avec la notion d'interface.

Héritage

- **Empêcher l'héritage**

- Il est possible d'empêcher qu'une classe soit utilisée comme classe parent. Il suffit d'ajouter le modificateur **final** dans la déclaration de la classe.

- Exemple

```
public final class Carre {  
    ...  
}
```

- **Note** : Plusieurs classes de la bibliothèque Java sont définies avec cette restriction. C'est le cas, par exemple, de la classe `String`.

Héritage

- **Exercice**

- Modélisation de comptes bancaires
 - comptes épargnes : taux d'intérêt, appliqué mensuellement sur le solde minimum du mois
 - comptes courants : opérations de retrait payantes (N gratuites par mois)
 - structure commune : numéro de compte, propriétaire, solde
 - opérations communes : obtenir le solde, effectuer un dépôt, effectuer un retrait, appliquer un traitement mensuel (p.ex. calcul des intérêts)

Héritage

- **Exercice**
 - Modélisation de comptes bancaires

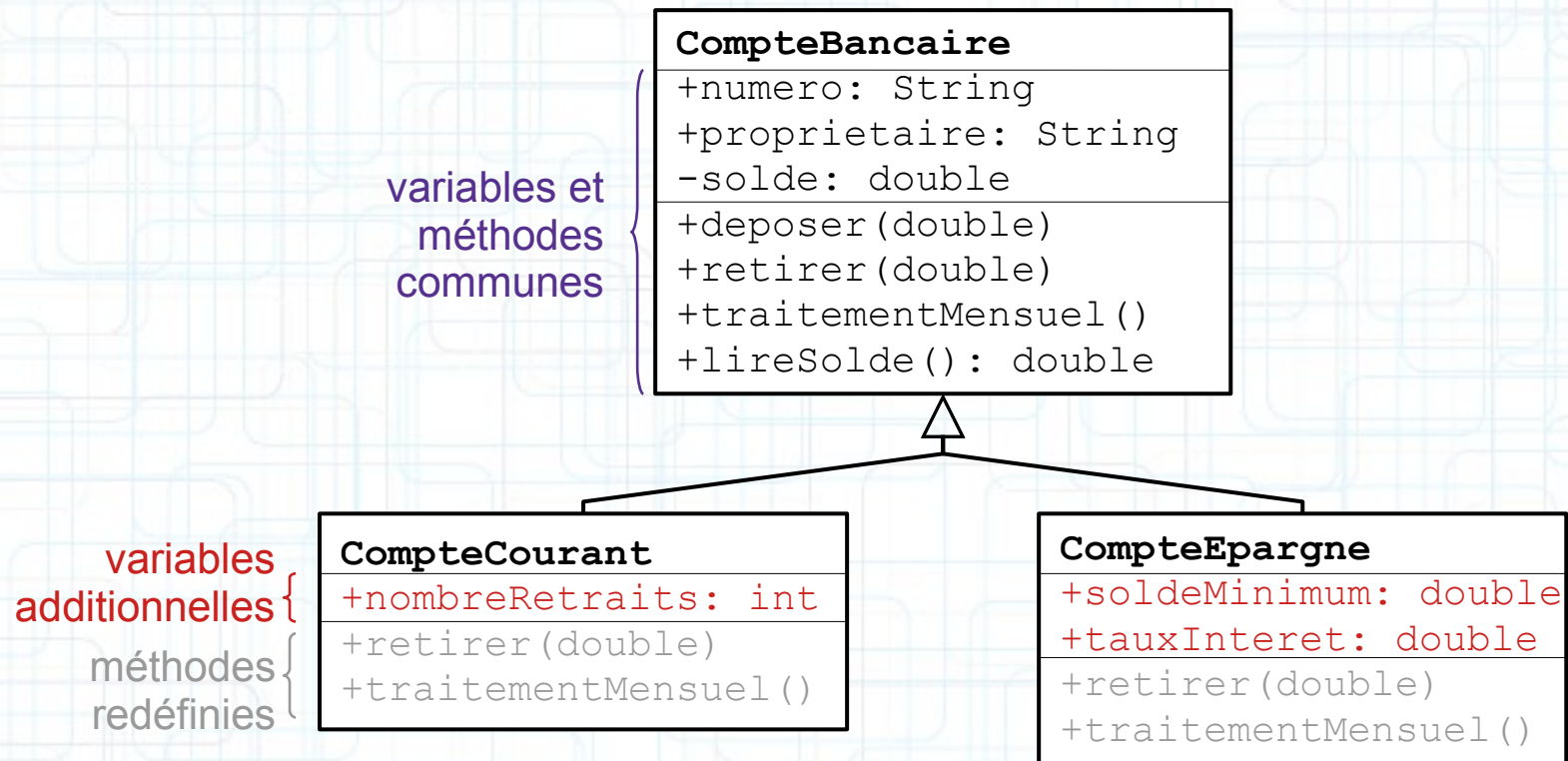


Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

Héritage

- **Compatibilité entre types**

- Au **Chapitre II**, nous avons considéré que le type d'une référence est toujours égal à celui de son instance.

```
Livre livre= new Livre();
```

```
Livre livre= new Carre();
```

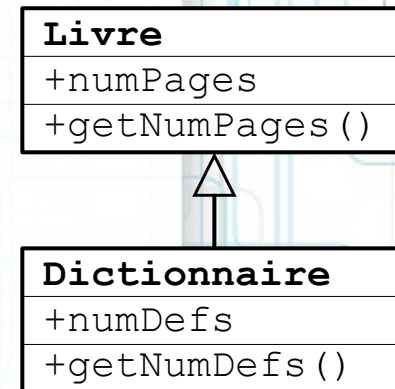
- **L'héritage change-t-il cette règle ?**

- Est-il légitime de manipuler un Dictionnaire comme n'importe quel Livre ? **OUI**

```
Livre livre= new Dictionnaire();  
System.out.println(livre.getNumPages());
```

- Est-il légitime de manipuler un Livre comme n'importe quel Dictionnaire ? **NON**

```
Dictionnaire dico= new Livre();  
System.out.println(dico.getNumDefs());
```



Héritage

- **Polymorphisme**

- **Corollaire** : Les références sont **polymorphiques** : une référence peut désigner des instances de différents types.
 - MAIS il ne doit être possible d'avoir une référence de type **A** vers une instance de type **C** que si « *l'instance de **C** peut être manipulée comme une instance de **A*** ».



- **Propriété des références** : toute référence de type **A** vers une instance de type **C** doit vérifier la propriété (**C is-a A**).
 - Le compilateur et la JVM se chargent de vérifier que cette propriété est toujours vérifiée.

Héritage

- **Polymorphisme**

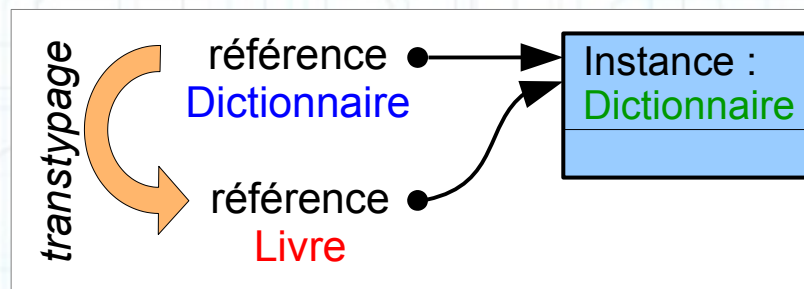
- Une **conversion d'un type** appelée aussi **transtypage** (*typecasting*) peut avoir lieu *implicitement* lors

- des affectations : variable destination vs valeur

```
Dictionnaire dico = new Dictionnaire();  
Livre livre = dico;
```

- des appels de méthodes : arguments vs valeurs

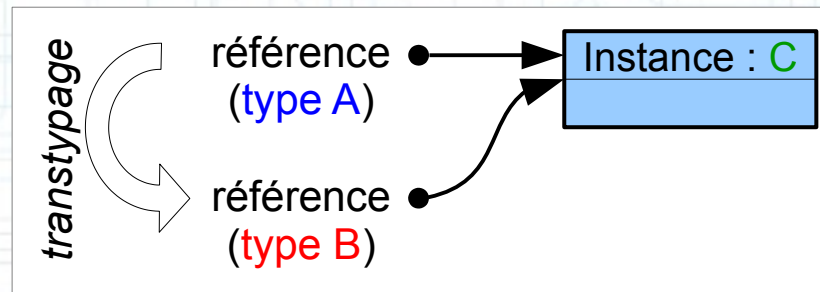
```
public static void printInfo(Livre livre) {  
    System.out.println(livre.getNumPages());  
}  
Dictionnaire dico = new Dictionnaire();  
printInfo(dico);
```



Héritage

- **Polymorphisme**

- Quand une conversion du type **A** vers le type **B** est-elle légitime ?

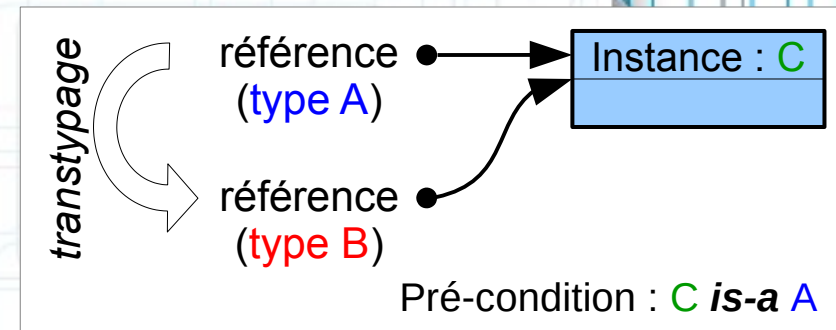


- La conversion ne peut être acceptée que si la référence résultant respecte la propriété (**C is-a B**).

Héritage

- **Conversion de types**

- Quand peut-on garantir que **C is-a B** ?
- Le compilateur connaît les types **A** et **B**. Trois situations se présentent selon leur relation.
- **non(A is-a B) et non (B is-a A)**
 - Transtypage invalide ⇒ **refusé** par le compilateur
- **A is-a B**
 - **Transtypage vers le haut** (*upcasting*)
 - Conversion *implicite* **acceptée** par le compilateur
- **B is-a A**
 - **Transtypage vers le bas** (*downcasting*)
 - Conversion *implicite* **refusée** par le compilateur
 - Conversion *explicite* possible (vérification par la JVM)



Héritage

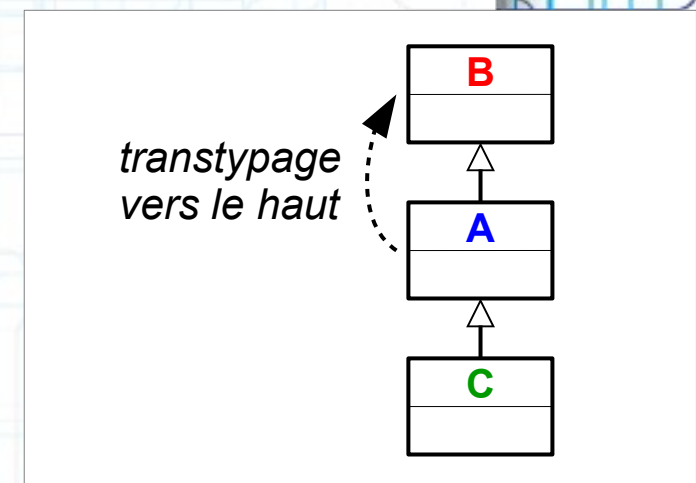
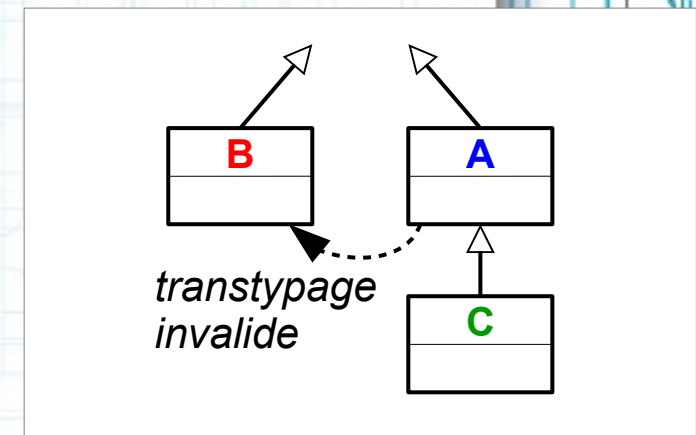
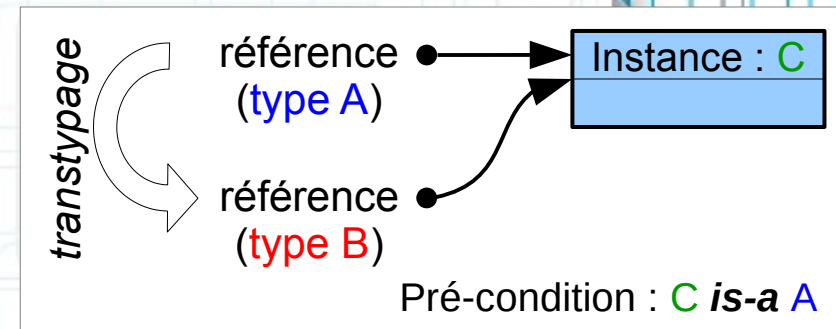
- Validité du transtypage

- Transtypage invalide

- Cas : non($A \text{ is-a } B$) et non ($B \text{ is-a } A$)
 - Supposons par l'absurde que ($C \text{ is-a } B$)
 - Comme ($C \text{ is-a } A$) et ($C \text{ is-a } B$),
il faut que soit ($A \text{ is-a } B$) soit ($B \text{ is-a } A$)
ce qui contredit notre hypothèse.

- Transtypage vers le haut

- Cas : $A \text{ is-a } B$
 - $C \text{ is-a } A$ (pré-condition)
 - Par transitivité, ($C \text{ is-a } B$)

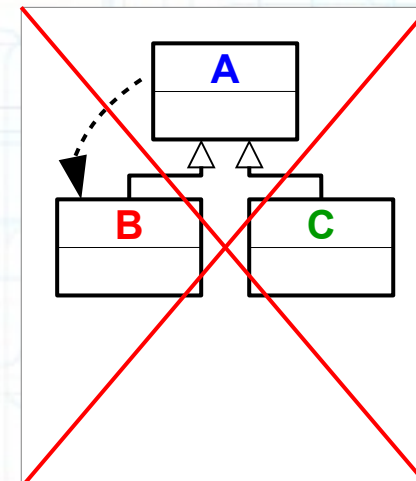
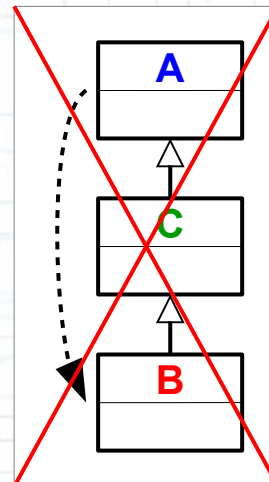
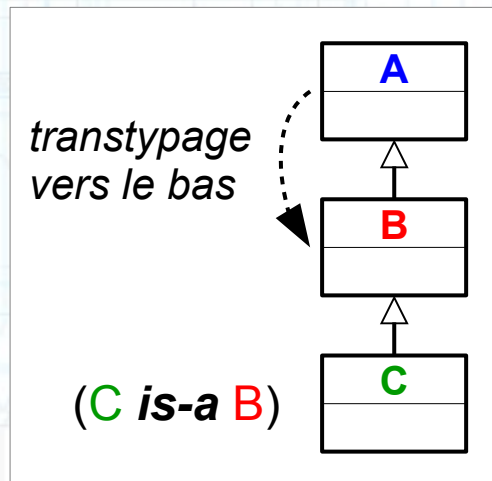


Héritage

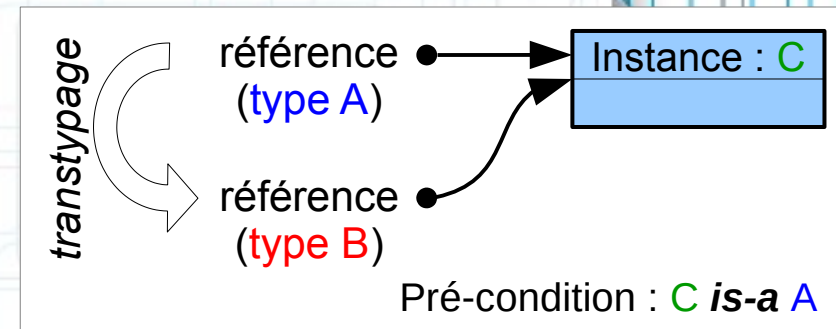
- Validité du transtypage

- Transtypage vers le bas

- Cas : **B** *is-a* **A**
 - C** *is-a* **A** (pré-condition)
 - Le compilateur ne peut rien conclure. Plusieurs situations sont possibles : certaines valides, d'autres pas.
 - Il faut connaître le type réel **C** de l'instance. Seule la JVM le connaît.



non(**C** *is-a* **B**)



Héritage

• Opérateur de transtypage

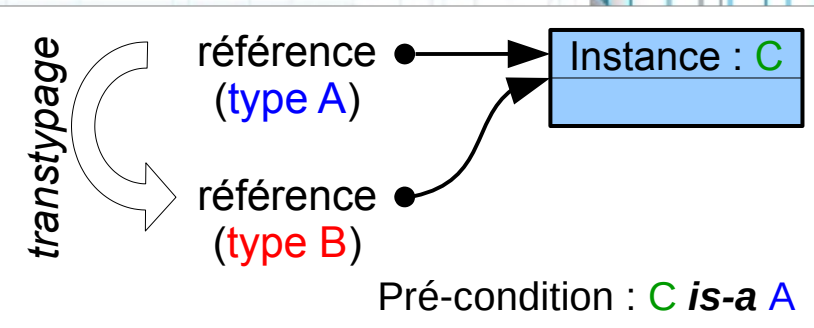
- L'**opérateur de transtypage** est utilisé pour changer *explicitement* le type d'une référence. Il permet d'indiquer au compilateur "qu'on sait ce qu'on fait".
- Seul moyen d'effectuer un transtypage vers le bas. Vérification déléguée à la JVM.
- L'opérateur de transtypage est un opérateur binaire.
 - Arguments : référence (source) et nom de type (cible).
 - Résultat : référence ayant le **type cible**.

– Syntaxe

(**nomType**) *reference*

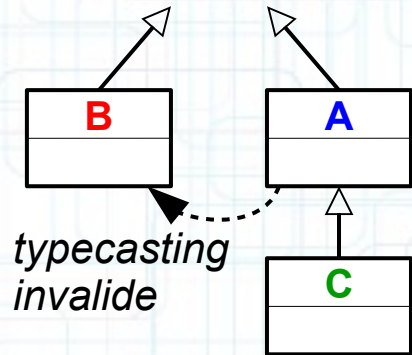
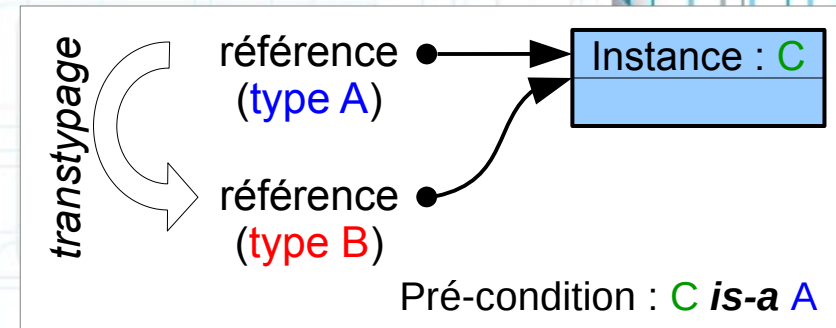
– Exemple

```
Livre livre= new Dictionnaire();  
Dictionnaire dico= (Dictionnaire) livre;
```

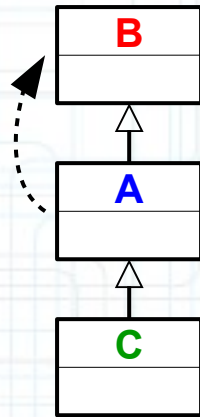


Héritage

• Transtypes valides



upcasting



vérif. compilateur

A is-a B

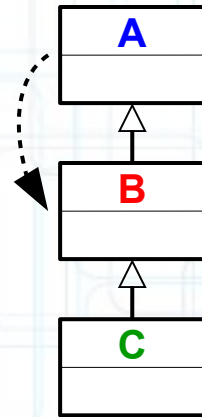
par transitivité

(*C is-a A*) et (*A is-a B*)

⇒ *C is-a B*

OK

downcasting



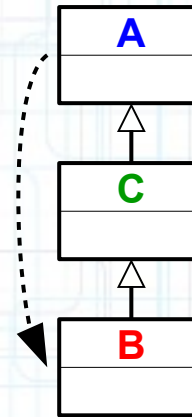
vérif. compilateur

B is-a A

vérif. JVM

C is-a B

OK



vérif. compilateur

B is-a A

vérif. JVM

non(*C is-a B*)

KO

typecasting
invalide

vérif. compilateur

non(*A is-a B*)

non(*B is-a A*)

supposons par l'absurde

(*C is-a B*)

⇒ comme (*C is-a A*), on a

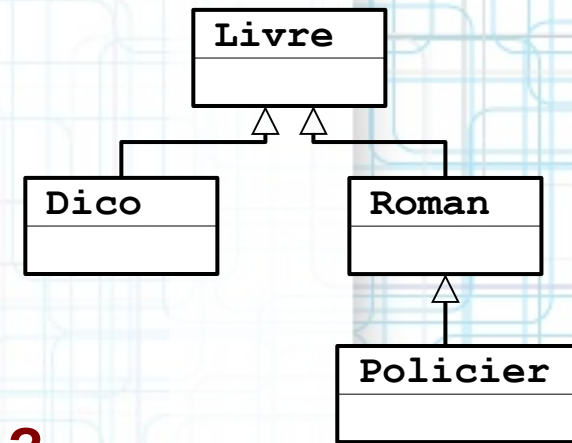
(*A is-a B*) ou (*B is-a A*)

⇒ contradiction

⇒ non(*C is-a B*)

KO

Héritage



- **Opérateur de transtypage**

- **Quels transtypes (explicites) sont valides ?**

- **Cas 1**

```
Roman roman= new Roman();
Dico dico= (Dico) roman;
```

comp. : non(Dico **is-a** Roman)
et non(Roman **is-a** Dico)

KO

- **Cas 2**

```
Roman roman= new Policier();
Livre livre= (Livre) roman;
```

comp. : Roman **is-a** Livre
(upcasting)

OK

- **Cas 3**

```
Livre livre= new Dico();
Dico dico= (Dico) livre;
```

comp. : Dico **is-a** Livre
(downcasting)

OK

JVM : Dico **is-a** Dico

- **Cas 4**

```
Livre livre= new Roman();
Dico dico= (Dico) livre;
```

comp. : Dico **is-a** Livre
(downcasting)

KO

JVM : non(Roman **is-a** Dico)

- **Cas 5**

```
Livre livre= new Roman();
Policier policier= (Policier) livre;
```

comp. : Policier **is-a** Livre
(downcasting)

JVM : non(Roman **is-a** Policier)

KO

Héritage

• Opérateur instanceof

- L'opérateur binaire **instanceof** permet de tester si une instance est compatible avec une classe particulière
 - L'opérateur prend en arguments une référence vers une instance de type A et le nom d'une classe B.
 - L'opérateur retourne **true** si **A is-a B**
- Syntaxe

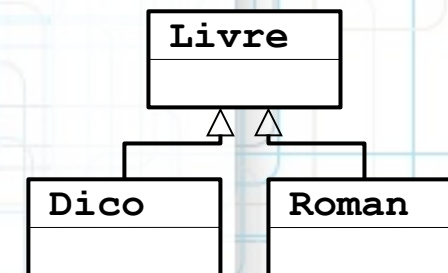
reference instanceof nomClasse

- Exemple

```
Livre livre= new Dico();  
System.out.println(livre instanceof Dico);  
System.out.println(livre instanceof Livre);  
System.out.println(livre instanceof Roman);
```

instanceof ?
réf. ● → Instance : A

Attention, c'est le **type de l'instance** qui est testé et non celui de la référence !



note : comme **instanceof** teste le type réel de l'instance, le résultat n'est connu qu'à l'exécution (JVM).

Héritage

- **Opérateur `instanceof`**

- **Attention !** L'opérateur `instanceof` est réservé à des cas bien particuliers.
 - Normalement, dans vos programmes il ne devrait pas être nécessaire de l'utiliser.
 - Si vous êtes amenés à utiliser `instanceof`, c'est peut-être parce que votre approche n'est pas orientée-objet et nécessite d'être retravaillée...

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe **Object**

4. Redéfinition de méthode

5. Liaison dynamique

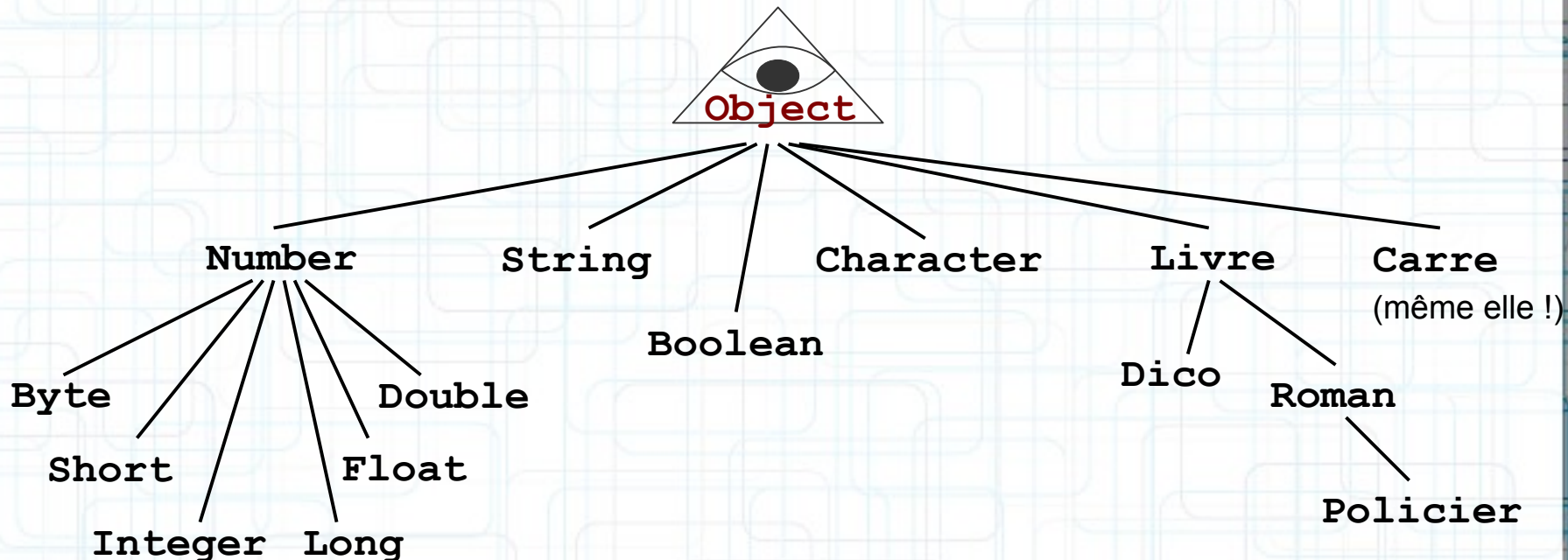
6. Masquage de variable

3. Encapsulation revisitée

Classe Object

- **La classe Object**

- Toutes les classes descendent d'une classe spéciale de la bibliothèque Java, la classe **Object**.
- Toute classe déclarée sans utilisation du mot-clé **extends** descend implicitement de la classe **Object**.



Classe Object

- **Class Object : services fournis**

- La classe `Object` fournit un certain nombre de services qui sont donc communs à toutes les classes !
- Notamment

```
public class Object {  
    public Object clone();  
    public boolean equals();  
    public Class getClass();  
    public int hashCode();  
    public String toString();  
    /* ... */  
}
```

permet de tester l'égalité (de contenu) entre deux objets.

retourne le type d'une instance, sous forme d'un objet `Class`.

retourne une représentation textuelle d'une instance.

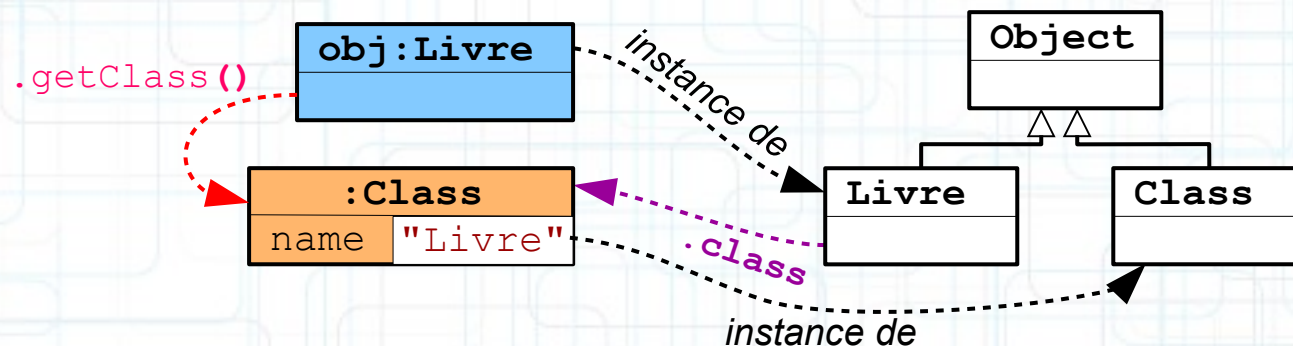
Classe Object

- **Tester l'égalité de deux instances**
 - La méthode **equals()** fournie par `Object` ne fait que tester l'égalité des références. Si deux objets ont la même référence, ils sont égaux.
 - Cependant, pour certaines sous-classes d'`Object`, l'égalité est définie sur base de leur état. Dans ce cas, la méthode `equals()` doit être redéfinie. Un exemple est la classe `String`.
 - Pour cette raison, de manière générale, il est nécessaire d'utiliser la méthode **equals()** pour tester l'égalité entre deux instances.

Classe Object

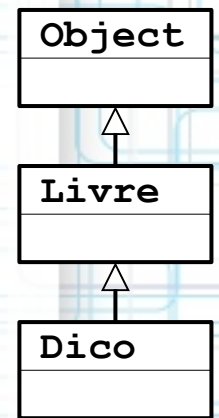
- **Classe Class**

- A chaque classe est associée une instance de **Class**
 - offre des méthodes telles que `getName()`, `getFields()`, `getConstructors()`, `getMethods()`, ...
 - la création d'instances de **Class** est interdite⁽¹⁾.
 - la classe d'une instance peut être récupérée avec la méthode **`getClass()`**
 - le littéral **`.class`** donne l'instance de **Class** associé à une classe (p.ex. `Livre.class`)



(1) La classe **Class** ne fournit pas de constructeur **public**.

Classe Object



- Applications de getClass()

- Exemple : obtenir le nom de la classe d'une instance.

```
Livre javaConcepts= new Livre();  
Object petitRobert= new Dico();  
System.out.println(javaConcepts.getClass().getName()); /* "Livre" */  
System.out.println(petitRobert.getClass().getName()); /* "Dico" */
```

- Exemple : tester le type d'une instance.

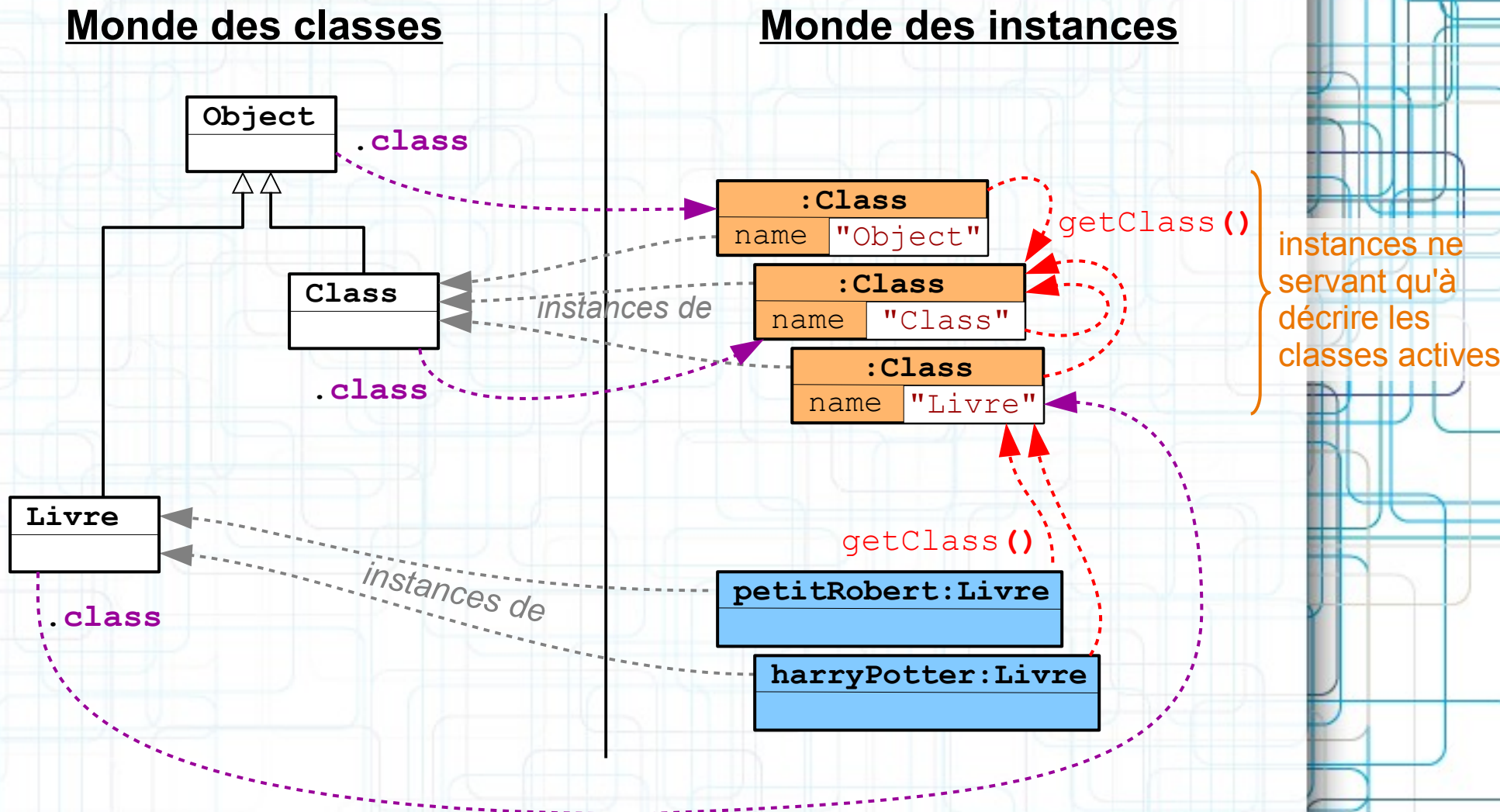
```
Object obj= new Dico();  
Class cls= obj.getClass();  
System.out.println(obj instanceof Livre); /* True */  
System.out.println(obj instanceof Dico); /* True */  
System.out.println(cls == Livre.class); /* False */  
System.out.println(cls == Dico.class); /* True */
```

On peut utiliser == car il n'existe qu'une instance de Class qui correspond à Livre OU Dico.

Contrairement à **instanceof**, ce test permet de déterminer la classe exacte de l'instance !

Classe Object

- Classes et instances de Class



Classe Object

- **Obtenir une représentation de la valeur**

- La méthode **toString()** retourne une représentation textuelle de la valeur d'une instance. Le résultat est une instance de `String`.
 - Par défaut, `toString()` retourne le nom de la classe de l'instance concaténée avec son *hashCode*⁽¹⁾.
 - Lorsqu'un objet est concaténé avec une `String` avec l'opérateur `+`, la méthode `toString()` est invoquée.

- Exemple

```
Livre javaConcepts= new Livre();  
Dictionnaire petitRobert= new Dictionnaire();  
System.out.println(javaConcepts.toString()); —→ Livre@@72e3b895  
System.out.println(petitRobert.toString()); —→ Dictionnaire@446b7920
```

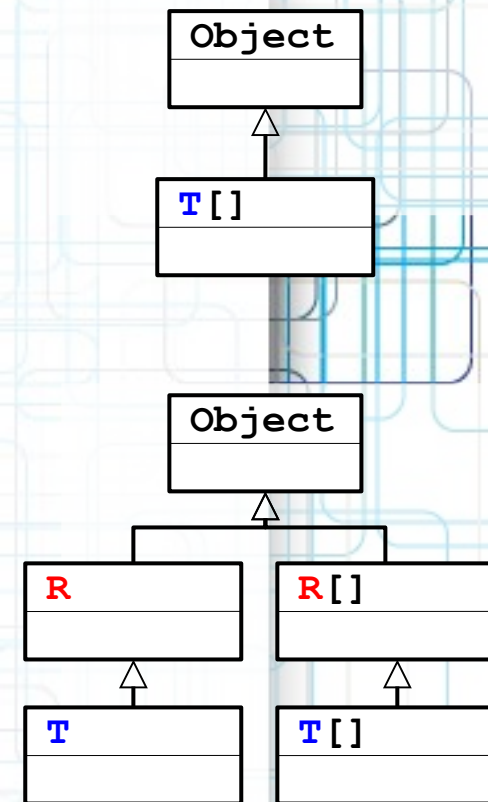
⁽¹⁾ Le *hashCode* est un entier servant à distinguer deux instances. Typiquement, deux instances n'ont un *hashCode* identique que si leur contenu est identique.

Héritage

- **Quid des tableaux ?**

- Les tableaux sont aussi des objets. Ils en héritent donc le comportement. De plus, un héritage entre tableaux est possible.

- Quelque soit T , ($T[]$ **is-a** Object).
Ceci est aussi valable si T est primitif.
- Si T est un type objet tel que (T **is-a** R),
alors ($T[]$ **is-a** $R[]$).
On dit que les tableaux d'objets sont **covariants**.
Cette propriété n'est pas valable si T est primitif.



Héritage

- Quid des tableaux ?

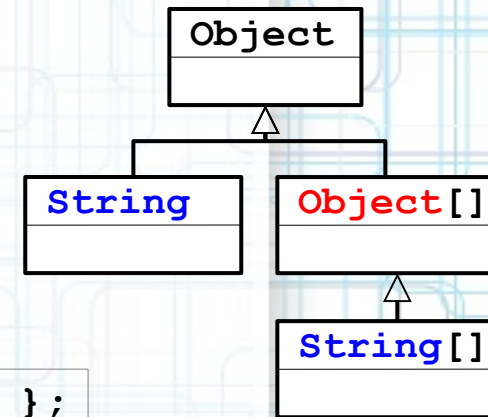
- Exemple

```
String[] auteurs= { "Orwell", "Bradbury", "Asimov" };  
System.out.println(auteurs.toString());
```

```
Object obj= auteurs;
```

```
Object[] objects= auteurs;  
System.out.println(objects.length);  
System.out.println(objects);
```

[Ljava.lang.String;@74a14482



- **Danger des alias !** Le fait que (T[] *is-a* Object[]) pourrait "casser" la vérification statique de types de Java.

```
String[] auteurs= { "Orwell", "Bradbury", "Asimov" };  
Object[] objects= auteurs;  
objects[0]= new Carre(5, 7);    /* accepté par le compilateur */
```

refusé par la JVM, ouf !

Exception in thread "main" java.lang.ArrayStoreException

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

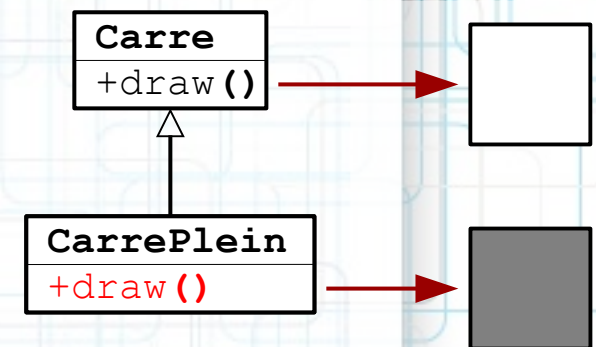
Redéfinition de méthode

- **Spécialisation**

- L'héritage permet de construire de nouvelles classes sur base de classes existantes. Par défaut, les méthodes héritées d'une classe parent sont identiques dans la classe enfant.
- Lors de la définition d'une nouvelle classe, il est souvent nécessaire de **spécialiser** son comportement.

- Exemple

- La méthode `draw()` de la classe `Carre` dessine le carré à l'écran.
- La classe `CarrePlein` dérivée de `Carre` hérite de `draw()`
- MAIS en **redéfinit** le comportement de façon à dessiner un carré plein.



Redéfinition de méthode

- Redéfinition de méthode

- La **redéfinition d'une méthode** (*overriding*) consiste à déclarer dans une sous-classe une méthode héritée d'une classe parent. Il s'agit d'un **mécanisme très important** de la Programmation Orienté-Objet (P.O.O.)
- En pratique,
 - La signature et le nom de la méthode redéfinie doivent être identiques à celle de la classe parent.
 - La nouvelle méthode remplace ou redéfinit le comportement hérité de la classe parent.

Redéfinition de méthode

- Redéfinition de méthode

- Exemple

```
public class Object {  
    public String toString() {  
        return getClass().getName() + hashCode();  
    }  
}
```

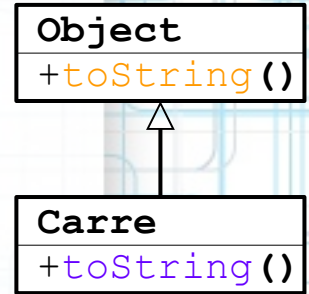
Méthode d'origine.

```
public class Carre extends Object {  
    public String toString() {  
        return "Carre[x=" + x  
            + ",y=" + y  
            + "]";  
    }  
}
```

Méthode redéfinie.

```
Carre carre= new Carre(5, 7);  
System.out.println(carre);          /* appel à toString() */
```

```
Carre[x=5,y=7]
```



Redéfinition de méthode

- **Méthode masquée et mot-clé super**

- Lorsqu'une sous-classe redéfinit une méthode, la méthode du parent est masquée.
- Le mot-clé **super** permet d'invoquer la méthode définie dans le parent.
- Exemple

```
public class A {  
    public void m() {  
        /* ... */  
    }  
}
```

```
public class B extends A {  
    public void m() {  
        m();  
        /* ... */  
    }  
}
```

La méthode du parent est masquée → l'appel de **m()** résulte en un appel récursif infini !!!

```
public class B extends A {  
    public void m() {  
        super.m();  
        /* ... */  
    }  
}
```

Référence la méthode **m()** définie dans le parent.

Redéfinition de méthode

- Redéfinition de méthode

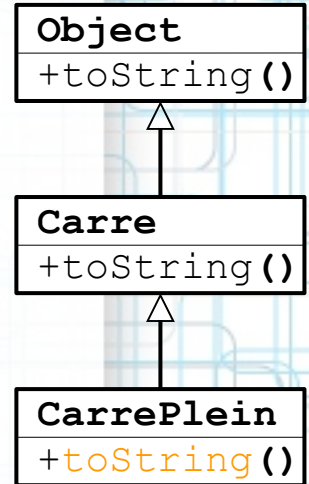
- Exemple

```
public class Carre extends Object {  
    public String toString() {  
        return getClass().getName()  
            + "[x=" + x  
            + ",y=" + y + "];"  
    }  
}
```

```
public class CarrePlein extends Carre {  
    private Color fillColor;  
    public String toString() {  
        return super.toString()  
            + "[fillColor=" + fillColor + "];"  
    }  
}
```

```
CarrePlein carre= new CarrePlein(5, 7);  
System.out.println(carre);
```

```
CarrePlein[x=5,y=7][fillColor=Color[LightGrey]]
```



Redéfinition de méthode

- **Constructeur**

- Le mot réservé **super** permet également d'appeler le constructeur de la classe parent à partir du constructeur d'une sous-classe.

- Exemple

```
public class CarrePlein extends Carre {  
    ...  
    public Color fillColor;  
    ...  
    public CarrePlein(double x, double y,  
                      double longueurCote,  
                      Color fillColor) {  
        super(x, y, longueurCote);  
        this.fillColor= fillColor;  
    }  
}
```

-----> Nouvel attribut.

-----> Nouveau constructeur.

1) Ré-utilise le **constructeur du parent**

2) Initialise le nouvel attribut.

Redéfinition de méthode

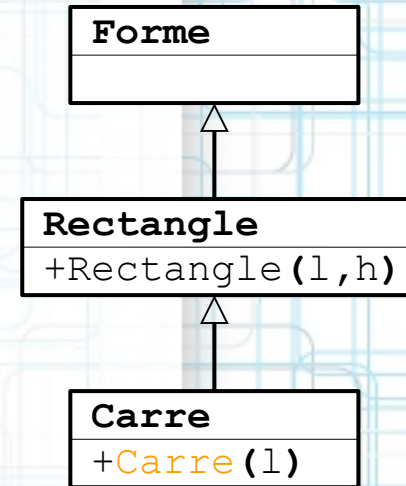
- Constructeur

- Autre exemple

```
public class Rectangle extends Forme {  
    public double largeur, hauteur;  
    public Rectangle(double largeur, double hauteur) {  
        this.largeur= largeur;  
        this.hauteur= hauteur;  
    }  
}
```

```
public class Carre extends Rectangle {  
    public Carre(double largeur) {  
        super(largeur, largeur);  
    }  
}
```

Un Carre est un Rectangle dont les largeur et hauteur sont égales.



Redéfinition de méthode

- **Annotation `@Override`**

- Lors d'une redéfinition de méthode, les erreurs suivantes peuvent se produire
 - la méthode à redéfinir n'existe pas dans un parent
 - la méthode est mal-orthographiée
- Afin d'aider le compilateur à détecter ces erreurs, il est possible (et recommandé) d'utiliser l'annotation **`@Override`**
- Exemple

```
public class Student {  
    @Override  
    public String toString() {  
        return "Student[name=" + name + "];"  
    }  
}
```

Le compilateur peut détecter qu'il n'existe pas de méthode `toString()` dans le parent (Object).

Redéfinition de méthode

- **Empêcher la redéfinition**

- Il est possible d'empêcher qu'une méthode soit redéfinie par une sous-classe. Il suffit d'ajouter le modificateur **final** dans la déclaration de la méthode.

- Exemple

```
public class Carre {  
    ...  
    public final void dessiner() {  
        ...  
    }  
}
```

- Note : une classe B peut surcharger une méthode d'une classe parent A et la redéfinir avec le mot-clé **final** afin qu'elle ne soit pas surchargée par ses propres sous-classes.

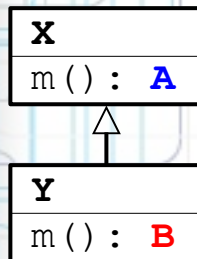
Redéfinition de méthode

- **Empêcher la redéfinition**
 - Il y a deux intérêts principaux pour empêcher la redéfinition d'une méthode
 - Premièrement, le développeur de la classe souhaite que les utilisateurs de sa classe ne puissent pas redéfinir une méthode pour des **raisons propres à son design**.
 - Deuxièmement, le développeur de la classe souhaite **forcer la liaison statique** (voir plus loin). Si la surcharge de méthode n'est pas permise, la liaison dynamique n'est pas nécessaire et cela peut amener un gain de performance (pas de décision à prendre à l'exécution).

Redéfinition de méthodes

- **Type de retour : covariant**

- Lors de la redéfinition d'une méthode, la signature de la redéfinition doit être identique à celle de la méthode de la classe parent.
- **Le type de retour PEUT être différent⁽¹⁾**. Il doit cependant être un sous-type du type d'origine
 - type de retour dans la classe parent : **A**
 - type de retour de la redéfinition : **B**
 - contrainte : **B is-a A**
- Les types de retour des deux méthodes sont dits "**covariants**".



(1) Depuis la version 5.0 de Java.

Redéfinition de méthodes

- **Type de retour : covariant**

- Exemple

```
public class Livrotron {  
    public Livre generate(String titre) {  
        return new Livre(titre);  
    }  
}
```

```
public class Dicotron extends Livrotron {  
    public Dictionnaire generate(String titre) {  
        return new Dictionnaire(titre);  
    }  
}
```

```
public class Test {  
    public static void main(String [] args) {  
        Livre [] livres= {  
            (new Livrotron()).generate("From 0 to 1"),  
            (new Dicotron()).generate("Robert & Collins"),  
        };  
    }  
}
```

Livrotron
generate() : **Livre**

Dicotron
generate() : **Dictionnaire**

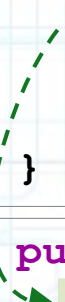
Redéfinition de méthodes

- **Spécificateur d'accès**

- Une méthode redéfinie PEUT avoir un spécificateur d'accès différent de celui de la méthode d'origine. Il ne peut cependant pas être plus restrictif !

```
public class A {  
    protected void m() {  
        System.out.println("A");  
    }  
}
```

```
public class B extends A {  
    public void m() {  
        System.out.println("B");  
    }  
}
```



```
public class A {  
    public void m() {  
        System.out.println("A");  
    }  
}
```

```
public class B extends A {  
    private void m() {  
        System.out.println("B");  
    }  
}
```

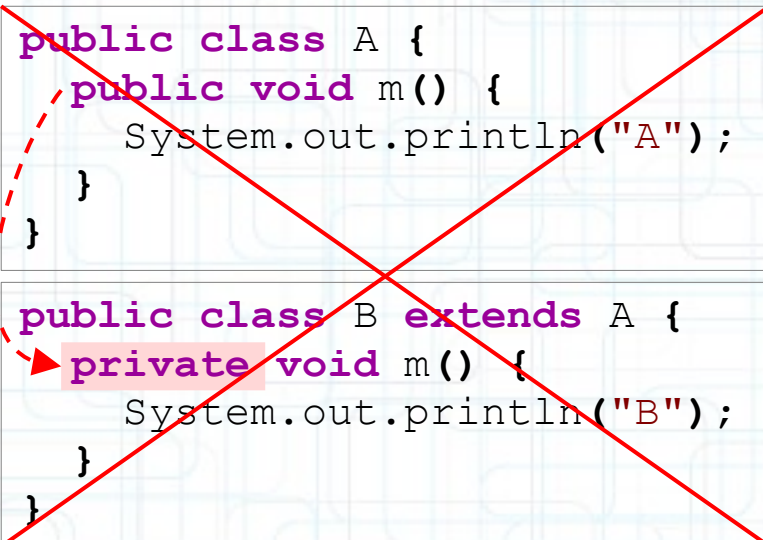


Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

Liaison dynamique

- **Dé-référencement**

- Au **Chapitre II**, nous avons introduit l'*opérateur de dé-référencement* qui permet d'accéder aux membres (attributs ou méthodes) d'une instance ou d'une classe.
 - **Membres d'instance** : (référence, symbole) → membre
 - **Membres de classe** : (nom classe, symbole) → membre
- En présence d'*héritage*
 - **Références polymorphiques** : une référence de type **A** peut référencer des instances de type **B** où (**B is-a A**).
 - **Méthodes redéfinies** : la classe **B** peut avoir surchargé des méthodes de **A**, sa classe parent.
- **Comment le dé-référencement effectue-t-il cette correspondance en présence d'héritage ?**

Liaison dynamique

- **Types de liaison**

- La « **liaison** » (*binding*) est le processus qui établit la correspondance entre une paire (référence/classe, symbole) et un membre.
- En Java la liaison est réalisée de deux façons différentes :
- **Liaison statique** (*early-/static-binding*)
 - Réalisé lors de la compilation, par le compilateur.
 - Utilisé pour : **variables d'instance**, **variables de classe** et **méthodes de classe**.
- **Liaison dynamique** (*late-/dynamic-binding*)
 - Réalisé lors de l'exécution, par la machine virtuelle (JVM).
 - Utilisé pour les **méthodes d'instance**, lorsque la méthode à appeler ne peut être déterminée lors de la compilation.

Liaison dynamique

- **Principe**

- Lors du dérèférencement d'une méthode d'instance, le compilateur ne sait pas déterminer avec certitude quelle méthode est désignée (polymorphisme + redéfinitions).
 - Le compilateur ne connaît que le type de la référence.
 - La machine virtuelle effectue la liaison car elle connaît le type réel de l'instance. On parle alors de **liaison dynamique** (ou de liaison tardive).
- Note : en Java, la liaison dynamique est utilisée **uniquement** lors du dé-rèférencement d'une méthode d'instance.
- Lorsqu'une méthode est privée (**private**) ou ne peut être redéfinie (**final**), le compilateur peut effectuer la liaison.

Liaison dynamique

- Liaison dynamique

- Exemple

```
public class A {  
    public void showMessage() {  
        System.out.println("Methode de A");  
    }  
}
```

```
public class B extends A {  
}
```

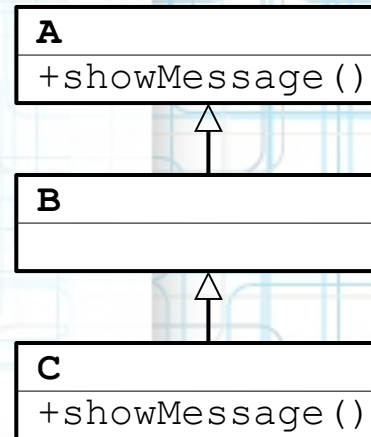
```
public class C extends B {  
    public void showMessage() {  
        System.out.println("Methode de C");  
    }  
}
```

```
A ref= new A();  
ref.showMessage();  
ref= new B();  
ref.showMessage();  
ref= new C();  
ref.showMessage();
```

Methode de A

Methode de A

Methode de C



Le type de la référence est identique
dans les 3 cas (A) → le compilateur ne fait pas la différence.

L'instance référencée est différente
(A, B ou C) → la JVM détermine quelle méthode appeler, lors de l'exécution

Liaison dynamique

```
A ref= new B() ;  
ref.m() ;
```

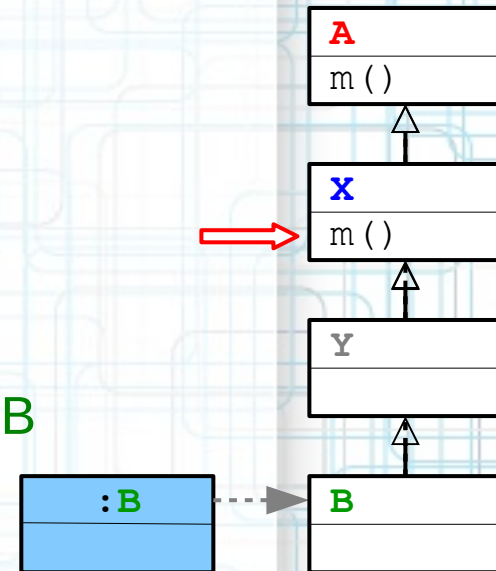
- **Fonctionnement**

- Soit

- une classe **A** qui définit une méthode **m()**
 - une classe **B** telle que (**B is-a A**)
 - une référence de type **A** vers une instance de type **B**

- Quelle méthode doit être appelée ?

- Ne dépend pas du type **A** de la référence !
 - Dépend du type de l'instance **B**
 - La méthode à appeler est celle (re-)définie par la classe **X** telle que
 - (**B is-a X**) et (**X is-a A**)
 - pour tout $Y \neq X$ telle que (**B is-a Y**) et (**Y is-a X**), **Y** ne redéfinit pas **m()**



Liaison dynamique

- **Algorithme**

- La machine virtuelle utilise l'algorithme suivant pour trouver (« résoudre ») une méthode d'instance à partir d'une référence et d'un nom de méthode (symbole).

entrée : *v* est la référence vers l'instance
 m est le nom de la méthode
sortie : méthode

```
I = instance(V)
C = class(I)
faire
    si la classe C contient une définition de la méthode m
        retourner C.m
    sinon
        C = superclass(C)
tant que (C != null)
générer erreur(« méthode non trouvée »)
```

Note : l'algorithme exact (plus efficace) peut être trouvé dans la spécification de la machine virtuelle, dans la description de l'instruction bytecode `invokevirtual`

(<https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokevirtual>)

Liaison dynamique

- **Algorithme**

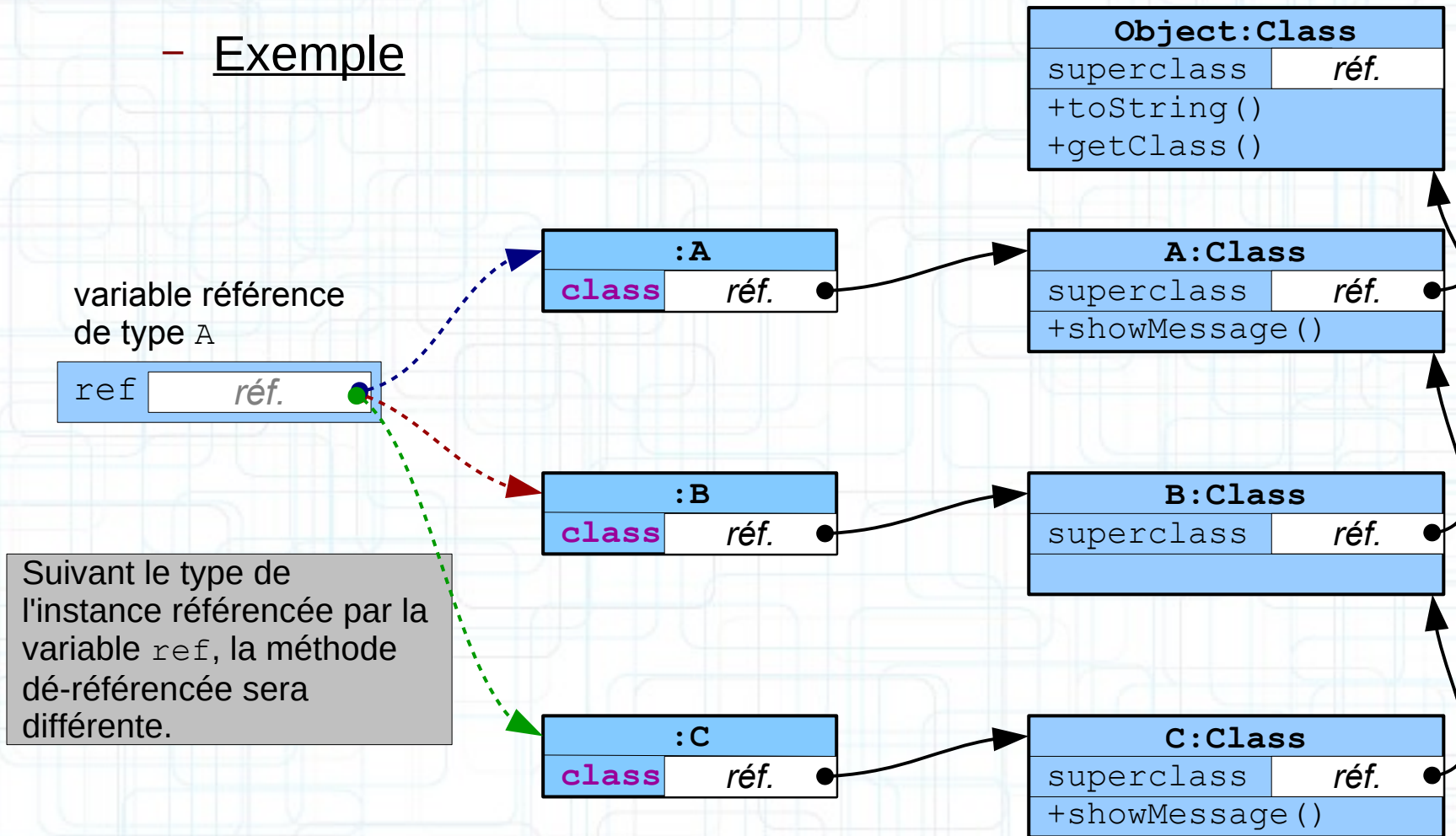
- Pour rendre les liaisons dynamiques possibles, il faut que la machine virtuelle puisse déterminer à l'exécution quelle méthode appeler.
- En particulier, en Java,
 - chaque instance en mémoire contient une référence vers une représentation de sa classe. Souvenez vous de la méthode `getClass()` définie par la classe `Object`.
 - chaque représentation de classe contient une référence vers sa classe parent. Nous notons cette référence `superclass`⁽¹⁾.

⁽¹⁾ Il ne s'agit pas d'un mot-clé utilisable en java. En revanche, la classe `Class` possède une méthode `getSuperClass()` permettant de déterminer la classe parent d'une classe.

Liaison dynamique

- **Algorithme**

- Exemple



Liaison dynamique

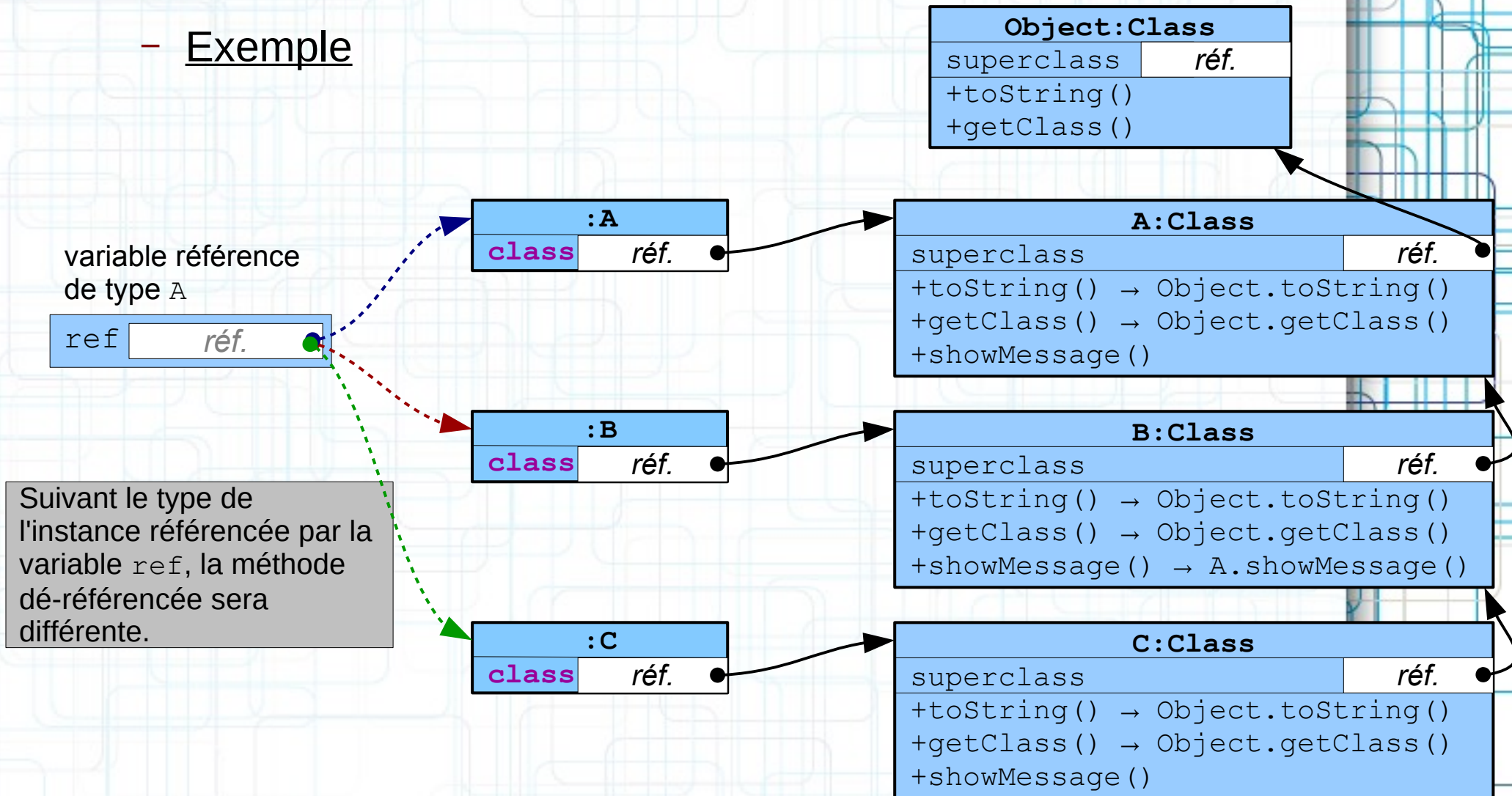
- **Table de méthodes**

- Il serait trop lent d'exécuter l'algorithme précédent à chaque appel de méthode.
- En pratique, la machine virtuelle pré-calcule une **table de méthodes** pour chaque classe. Lorsqu'une méthode est appelée, un simple accès à la table permet d'obtenir la bonne méthode.

Liaison dynamique

- Table de méthodes

- Exemple



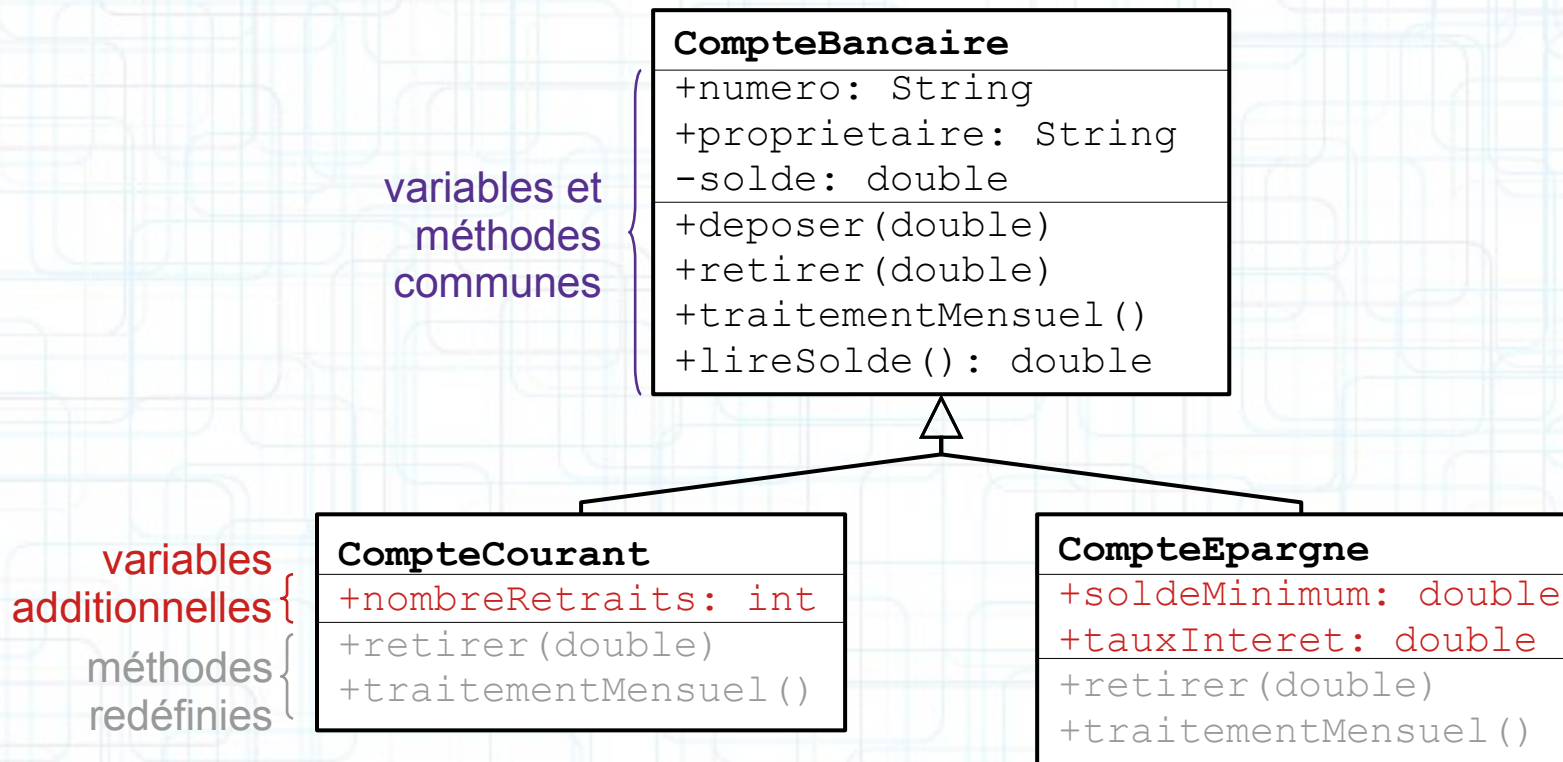
Héritage

- **Exercice (suite)**

- Modélisation de comptes bancaires
 - comptes épargnes : taux d'intérêt, appliqué mensuellement sur le solde minimum du mois
 - comptes courants : opérations de retrait payantes (N gratuites par mois)
 - structure commune : numéro de compte, propriétaire, solde
 - opérations communes : obtenir le solde, effectuer un dépôt, effectuer un retrait, appliquer un traitement mensuel (p.ex. calcul des intérêts)
- Fournir une implémentation en Java tirant parti de l'héritage, de la **redéfinition de méthode** et du **polymorphisme**...

Héritage

- **Exercice (suite)**
 - Modélisation de comptes bancaires



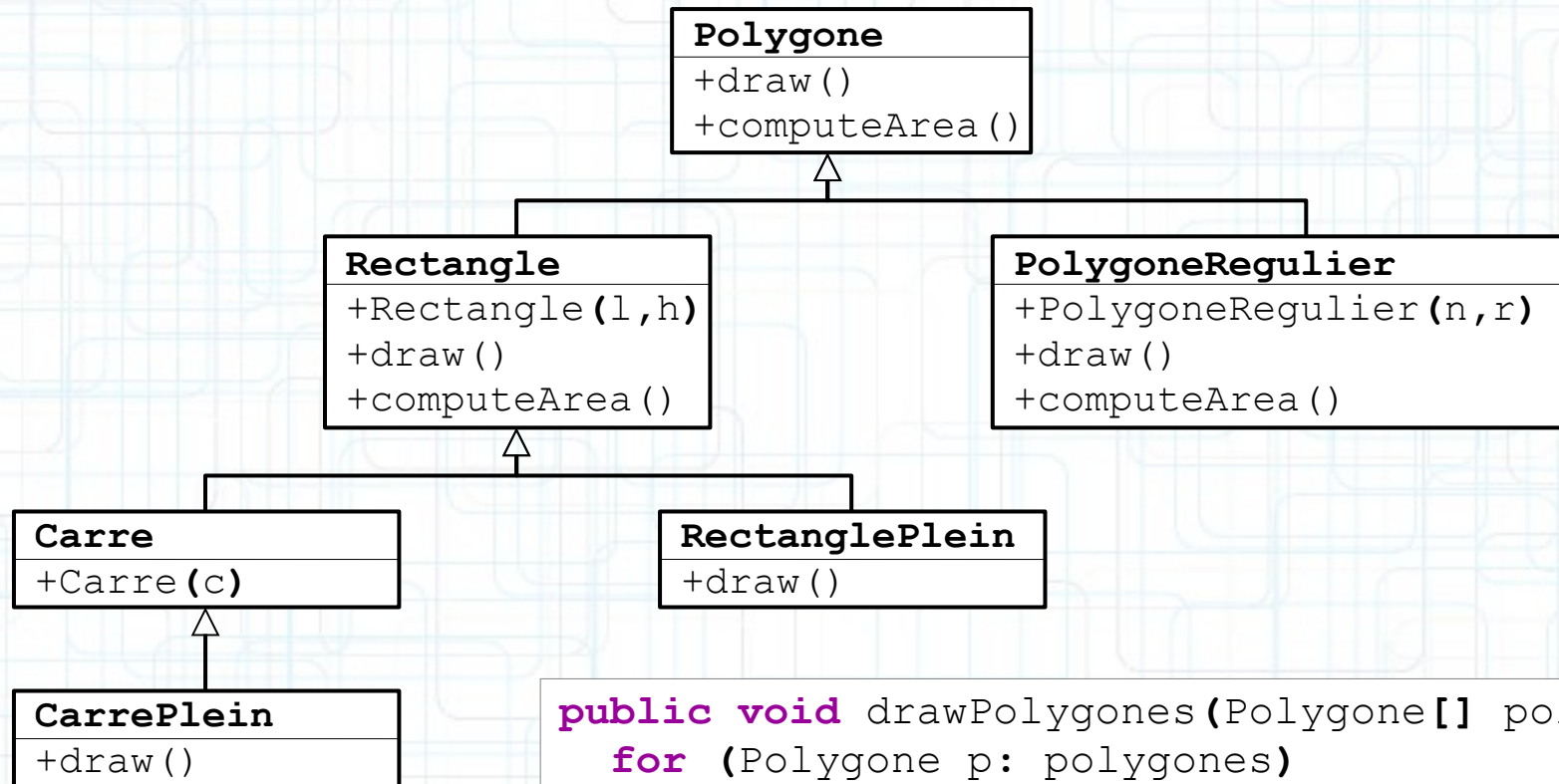
Liaison dynamique

- **Polymorphisme**

- Le polymorphisme est un outil particulièrement puissant lorsqu'utilisé avec des tableaux ou des collections d'objets.
- Grâce à la liaison dynamique, le comportement du programme dépend du type réel de chaque instance conservée par le tableau / la collection.
- Exemple
 - Supposons un programme qui doit manipuler une collection de polygones (carrés, rectangles, polygones réguliers, ...) ?
 - Pour chacun d'entre eux, il est nécessaire de pouvoir calculer le périmètre, la surface, ... éventuellement les dessiner à l'écran.

Liaison dynamique

- Polymorphisme



```
public void drawPolygones(Polygone[] polygones) {
    for (Polygone p: polygones)
        p.draw();
}
```

```
Polygone[] polygones= { new PolygoneRegulier(5, 3),
    new Carre(5), new Rectangle(6, 18) };
drawPolygones();
```

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

Masquage de variable

- Redéfinition de variable

- Peut-on redéfinir des variables ?

```
public class Maman {  
    public int x= 10;  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
}
```

- Lorsqu'une variable d'une sous-classe masque une variable de la classe parent, en fait, **les deux variables vont co-exister** dans les instances de la sous-classe, mais seule une des deux pourra être accédée.



Masquage de variable

- **Masquage de variable**

- En cas de masquage, la variable accédée dépend uniquement du type de la référence utilisée. Il s'agit de liaison statique !!!
- Exemple

```
public class Maman {  
    public int x= 10;  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
}
```

```
Enfant ref1= new Enfant;  
Maman ref2= ref1;  
System.out.println(ref1.x); /* affiche 5 */  
System.out.println(ref2.x); /* affiche 10 */
```



Masquage de variable

- **Liaisons statique et dynamique**
 - Exercice : donnez la suite de valeurs imprimées

```
public class Maman {  
    public int x= 10;  
    public void setX(int x) {  
        this.x= x; }  
    public int getX() {  
        return x; }  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
    public void setX(int x) {  
        this.x= x+1; }  
}
```

```
Enfant enfant= new Enfant();  
Maman maman= enfant;  
System.out.println(enfant.x);           /* ??? */  
System.out.println(enfant.getX());      /* ??? */  
System.out.println(maman.x);            /* ??? */  
System.out.println(maman.getX());       /* ??? */  
enfant.setX(0);  
System.out.println(enfant.x);           /* ??? */  
System.out.println(maman.x);            /* ??? */  
maman.setX(100);  
System.out.println(enfant.x);           /* ??? */  
System.out.println(maman.x);            /* ??? */
```

Masquage de variable

- **Liaisons statique et dynamique**

- Exercice : donnez la suite de valeurs imprimées

```
public class Maman {  
    public int x= 10;  
    public void setX(int x) {  
        this.x= x; }  
    public int getX() {  
        return x; }  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
    public void setX(int x) {  
        this.x= x+1; }  
}
```

```
Enfant enfant= new Enfant();  
Maman maman= enfant;  
System.out.println(enfant.x);  
System.out.println(enfant.getX());  
System.out.println(maman.x);  
System.out.println(maman.getX());  
enfant.setX(0);  
System.out.println(enfant.x);  
System.out.println(maman.x);  
maman.setX(100);  
System.out.println(enfant.x);  
System.out.println(maman.x);
```

```
/* 5 */  
/* 10 */  
/* 10 */  
/* 10 */  
/* ??? */  
/* ??? */  
/* ??? */  
/* ??? */
```

Early-binding → Enfant.x

Late-binding → Maman.getX(),
puis early-binding → Maman.x

Early-binding → Maman.x

Late-binding → Maman.getX(),
puis early-binding → Maman.x

Masquage de variable

- Liaisons statique et dynamique

- Exercice : donnez la suite de valeurs imprimées

```
public class Maman {  
    public int x= 10;  
    public void setX(int x) {  
        this.x= x; }  
    public int getX() {  
        return x; }  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
    public void setX(int x) {  
        this.x= x+1; }  
}
```

```
Enfant enfant= new Enfant();  
Maman maman= enfant;  
System.out.println(enfant.x);           /* 5 */  
System.out.println(enfant.getX());      /* 10 */  
System.out.println(maman.x);            /* 10 */  
System.out.println(maman.getX());       /* 10 */  
enfant.setX(0); -----  
System.out.println(enfant.x);           /* 1 */  
System.out.println(maman.x);            /* inchangé */  
maman.setX(100); -----  
System.out.println(enfant.x);           /* 101 */  
System.out.println(maman.x);            /* inchangé */
```

↖ Late-binding → Enfant.setX(),
puis early-binding → Enfant.x

Early-binding → Enfant.x

↖ Late-binding → Enfant.setX(),
puis early-binding pour Enfant.x

Early-binding → Enfant.x

Masquage de variable

- **Liaisons statique et dynamique**

- Exercice : donnez la suite de valeurs imprimées

```
public class Maman {  
    public int x= 10;  
    public void setX(int x) {  
        this.x= x; }  
    public int getX() {  
        return x; }  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
    public void setX(int x) {  
        super.setX(x+1); }  
}
```

Seul changement par rapport à l'exemple précédent.

```
Enfant enfant= new Enfant();  
Maman maman= enfant;  
System.out.println(enfant.x);           /* 5 */  
System.out.println(enfant.getX());      /* 10 */  
System.out.println(maman.x);            /* 10 */  
System.out.println(maman.getX());       /* 10 */  
enfant.setX(0);  
System.out.println(enfant.x);           /* ??? */  
System.out.println(maman.x);            /* ??? */  
maman.setX(100);  
System.out.println(enfant.x);           /* ??? */  
System.out.println(maman.x);            /* ??? */
```

Masquage de variable

- Liaisons statique et dynamique

- Exercice : donnez la suite de valeurs imprimées

```
public class Maman {  
    public int x= 10;  
    public void setX(int x) {  
        this.x= x; }  
    public int getX() {  
        return x; }  
}
```

```
public class Enfant extends Maman {  
    public int x= 5;  
    public void setX(int x) {  
        super.setX(x+1); }  
}
```

```
Enfant enfant= new Enfant();  
Maman maman= enfant;  
System.out.println(enfant.x);           /* 5 */  
System.out.println(enfant.getX());      /* 10 */  
System.out.println(maman.x);            /* 10 */  
System.out.println(maman.getX());       /* 10 */  
enfant.setX(0); -----  
System.out.println(enfant.x);           /* inchangé */  
System.out.println(maman.x);           /* 1 */  
maman.setX(100); -----  
System.out.println(enfant.x);           /* inchangé */  
System.out.println(maman.x);           /* 101 */
```

→ *Late-binding* → Enfant.setX(),
puis *late-binding* → Maman.setX(),
puis *early-binding* → Enfant.x

→ *Late-binding* → Enfant.setX(),
puis *late-binding* → Maman.setX(),
puis *early-binding* → Maman.x

Liaisons

- **Résumé**

- **Liaison dynamique**

- Méthode d'instance
 - Dépend uniquement du type réel de l'instance (connu par la JVM)

- **Liaison statique**

- Méthode de classe, Variable d'instance, Variable de classe
 - Dépend uniquement du type de la référence (connu par le compilateur)

Table des Matières

1. Introduction

1. Relation « *is-a* »

2. Héritage en Java

1. Déclaration de classe

2. Polymorphisme et Transtypage

3. Classe `Object`

4. Redéfinition de méthode

5. Liaison dynamique

6. Masquage de variable

3. Encapsulation revisitée

Encapsulation revisitée

- **En présence d'héritage**
 - Comment réaliser une bonne encapsulation en présence d'héritage ?
 - Deux objectifs s'opposent pour le contrôle d'accès à des variables et méthodes
 - empêcher l'accès à des **classes externes** (avec **private**)
 - autoriser l'accès à des **sous-classes**
 - Un nouveau spécificateur d'accès est nécessaire : **protected** permet aux sous-classes et à leurs instances d'accéder à une variable ou une méthode .
 - Nous verrons plus tard que **protected** permet également l'accès aux classes d'un même *package*.

Encapsulation revisitée

- **Spécificateur d'accès `protected`**

- Exemple

- Dans le cas des classes `Livre` et `Dictionnaire`, le champ `numPages` peut-être défini avec **`protected`** afin que les instances de la classe `Dictionnaire` puissent y accéder directement.
 - Si le champ `numPages` est défini comme **`private`**, les instances de `Dictionnaire` ne peuvent y accéder qu'au travers des accesseurs définis dans la classe `Livre`.

```
public class Livre {  
    private int numPages;  
    public void setNumPages(int numPages)  
    {  
        this.numPages= numPages;  
    }  
    public int getNumPages() {  
        return numPages;  
    }  
}
```

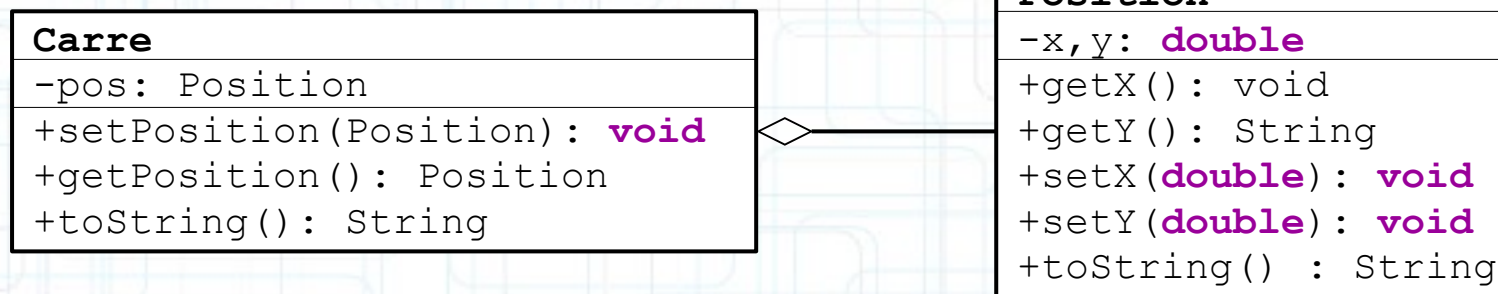
```
public class Livre {  
    protected int numPages;  
    public void setNumPages(int numPages)  
    {  
        this.numPages= numPages;  
    }  
    public int getNumPages() {  
        return numPages;  
    }  
}
```


Encapsulation revisitée

- **Variable d'instance référence**

- Les règles vues au cours précédent pour assurer une bonne encapsulation ne sont pas suffisantes dans le cas où une instance possède une **variable d'instance qui est une référence vers un autre objet**.

- Exemple



La flèche avec un
diamant non rempli
représente la relation
d'aggrégation.

Encapsulation revisitée

- **Variable d'instance référence**

- Example

```
public class Position {
    protected double x, y;
    public Position(double x, double y) {
        this.x= x;
        this.y= y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x) { this.x= x; }
    public void setY(double y) { this.y= y; }
    public String toString() { return "Position("+x+", "+y+")"; }
}
```

```
public class Carre {
    protected Position pos;
    public Carre(Position pos) {
        this.pos= pos;
    }
    public void setPosition(Position pos) { this.pos= pos; }
    public Position getPosition() { return pos; }
    public String toString() { return "Carre("+pos+")"; }
}
```

La variable d'instance `pos` est une référence vers un objet `Position`.

La variable d'instance `pos` est une référence vers un autre objet.

Encapsulation revisitée

- **Variable d'instance référence**
 - Les classes `Position` et `Carre` de l'exemple précédent ont été définies avec les règles d'encapsulation vues préalablement, à savoir
 - les variables d'instance sont définies avec **private** ou **protected**
 - des accesseurs sont fournis pour lire et écrire les variables d'instance.
 - Pourtant, du point de vue de l'encapsulation, **cet exemple a une faiblesse. Laquelle ?**

Encapsulation revisitée

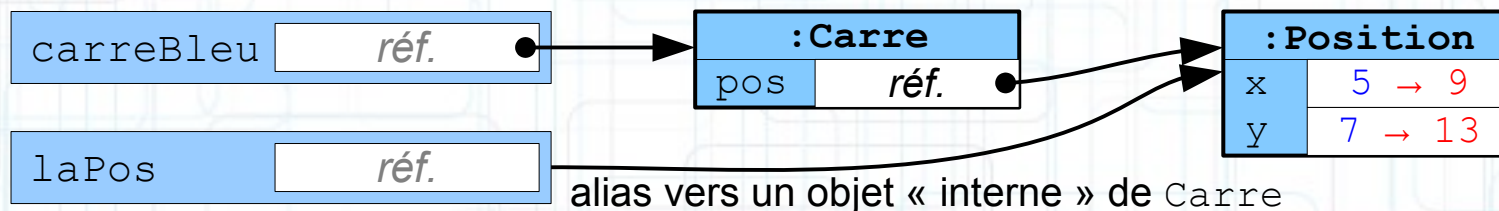
- Variable d'instance référence

- Exemple

```
Carre carreBleu= new Carre(new Position(5, 7));  
System.out.println(carreBleu);  
Position laPos= carreBleu.getPosition();  
laPos.setX(9);  
laPos.setY(13);  
System.out.println(carreBleu);  
System.out.println(laPos);
```

```
Carre(Position(5,7))  
Carre(Position(9,13))  
Position(9,13)
```

En manipulant l'objet retourné par `getPosition()`, il est possible de changer l'état interne de l'objet `Carre` ! Ouille, pas bon ça !



Encapsulation revisitée

- **Deux solutions possibles**

- **Objet référencé immuable** → rendre immuable l'objet référencé retourné par l'accessesseur de sorte que le client ne puisse le modifier.
- **Objet référencé copié** → retourner une copie de l'objet référencé de sorte que le client puisse modifier cette copie sans impact sur l'instance initiale.
- Ces deux solutions doivent être ajoutées à notre liste de règles de « bonne encapsulation » !!!

Encapsulation revisitée

- **Solution 1: Objet immuable**

- Exemple

```
public class Position {  
    protected double x, y;  
    public Position(double x, double y) {  
        this.x= x;  
        this.y= y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public void setX(double x) { this.x= x; }  
    public void setY(double y) { this.y= y; }  
    public String toString() { return "Position("+x+", "+y+")"; }  
}
```

Les mutateurs sont supprimés et la variable référence est **protected**. L'état de l'objet ne peut être changé.

- Note : on pourrait également ajouter le mot réservé **final** lors de la déclaration des variables d'instance `x` et `y` et supprimer les accesseurs `getX` et `getY`.

Encapsulation revisitée

- **Solution 2: Copie de l'objet référencé**

- Exemple

```
public class Carre {  
    protected Position pos;  
    public Carre(Position pos) {  
        this.pos= pos;  
    }  
    public void setPosition(Position pos) { this.pos= pos; }  
  
    public Position getPosition() {  
        return new Position(pos.getX(), pos.getY());  
    }  
  
    public String toString() { return "Carre("+pos+")"; }  
}
```

Ici, la méthode `getPosition()` crée une nouvelle instance de `Position` qui a les mêmes valeurs de champs `x` et `y`.

Il s'agit donc d'une copie de l'instance référencée par la variable d'instance `pos`.

Clonage

- **Problème**

- La copie complète d'un objet existant est appelée **clonage**. Il s'agit d'une opération fréquente.
- La responsabilité de la copie incombe à l'objet lui-même. Il est en effet nécessaire de connaître la structure interne de l'objet pour en effectuer une copie correcte. Or l'encapsulation vise à cacher cette structure !
- Cloner un objet correctement implique notamment de pouvoir répondre aux questions suivantes
 - L'objet contient-il des références vers d'autres objets ?
 - Faut-il les copier également ? Sont-ils immuables ? ...