

A decorative vertical bar on the left side of the slide is composed of several parallel lines in various shades of blue. To the right of this bar are several spheres of different sizes, also in various shades of blue, some with horizontal stripes. A thick red horizontal line spans across the top of the slide, and a thinner red horizontal line is positioned near the bottom.

# HÉRITAGE ET INTERFACES

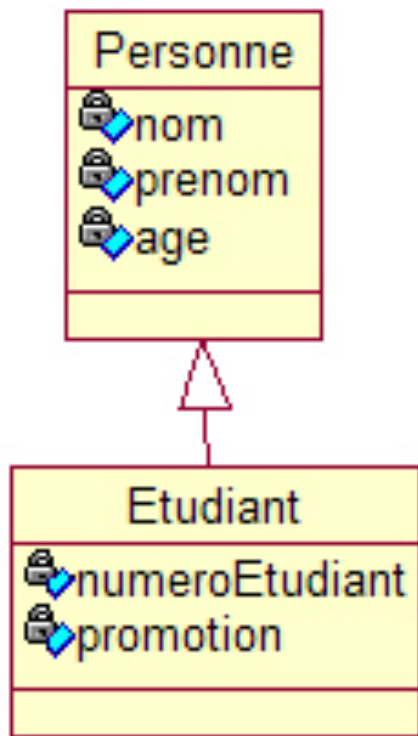
## Cours 5

# Héritage en Java

---

□ Mot clé : extends

□ Exemple :



```
public class Etudiant extends Personne {
```

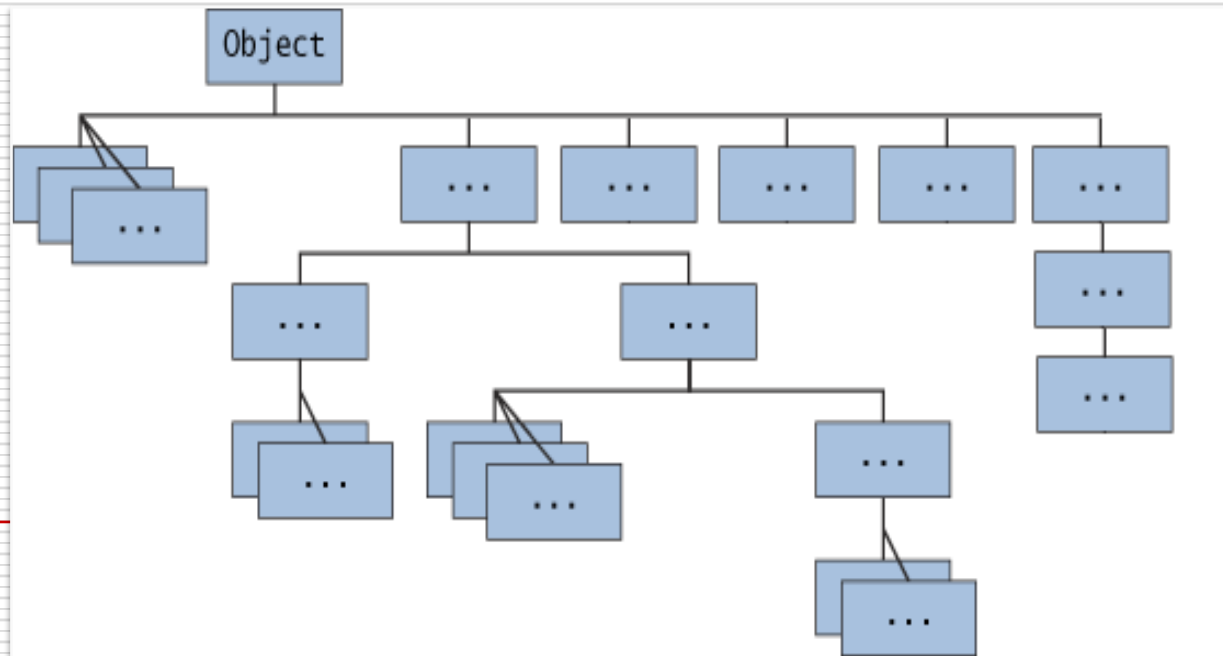
```
// corps de la classe Etudiant
```

```
}
```

# Héritage « multiple » en Java

---

- ❑ 1 classe ne peut hériter que d'une seule classe au maximum ...
- ❑ ... en plus de l'héritage implicite de la classe `java.lang.Object`



# Accès classe parente

---

- La classe fille a accès à tout membre non privé de la classe mère :
  - champs
  - méthodes
  
- La classe fille n'a pas accès aux
  - membres privés
  - constructeurs } de la classe mère

# Corps classe fille

---

On écrit uniquement :

- les constructeurs
- les nouveaux membres
- les membres redéfinis

# Constructeur classe fille (1)

---

❑ Réutilisation du constructeur parent avec le mot clé **super**

❑ Exemple :

```
// constructeur de la classe Etudiant
public Etudiant(String nom, String prenom, int age,
                String numEtudiant, String promotion)
{
    super(nom, prenom, age);
    this.numEtudiant = numEtudiant;
    this.promotion = promotion;
}
```

---

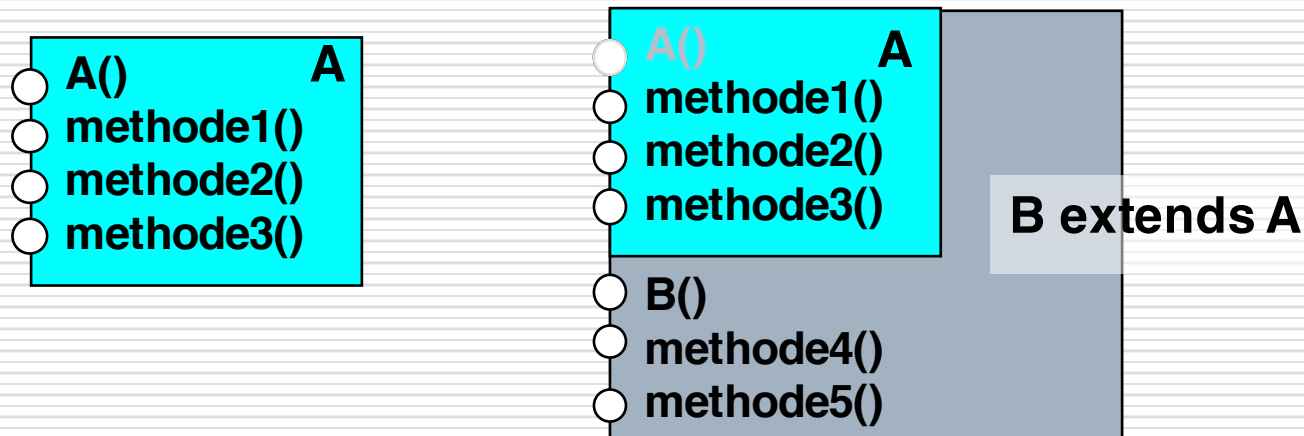
## Constructeur classe fille (2)

---

- ❑ **super(...)** toujours en 1<sup>ère</sup> instruction
- ❑ **attention** : si pas d'appel explicite
  - appel implicite au constructeur par défaut de la classe parente : `super();`
  - erreur si aucun constructeur par défaut dans la classe mère
- ❑ **conclusion** : toujours faire un appel explicite en réutilisant les initialisations de la classe parente

# Manipulation des références (1)

## □ vue graphique de l'héritage



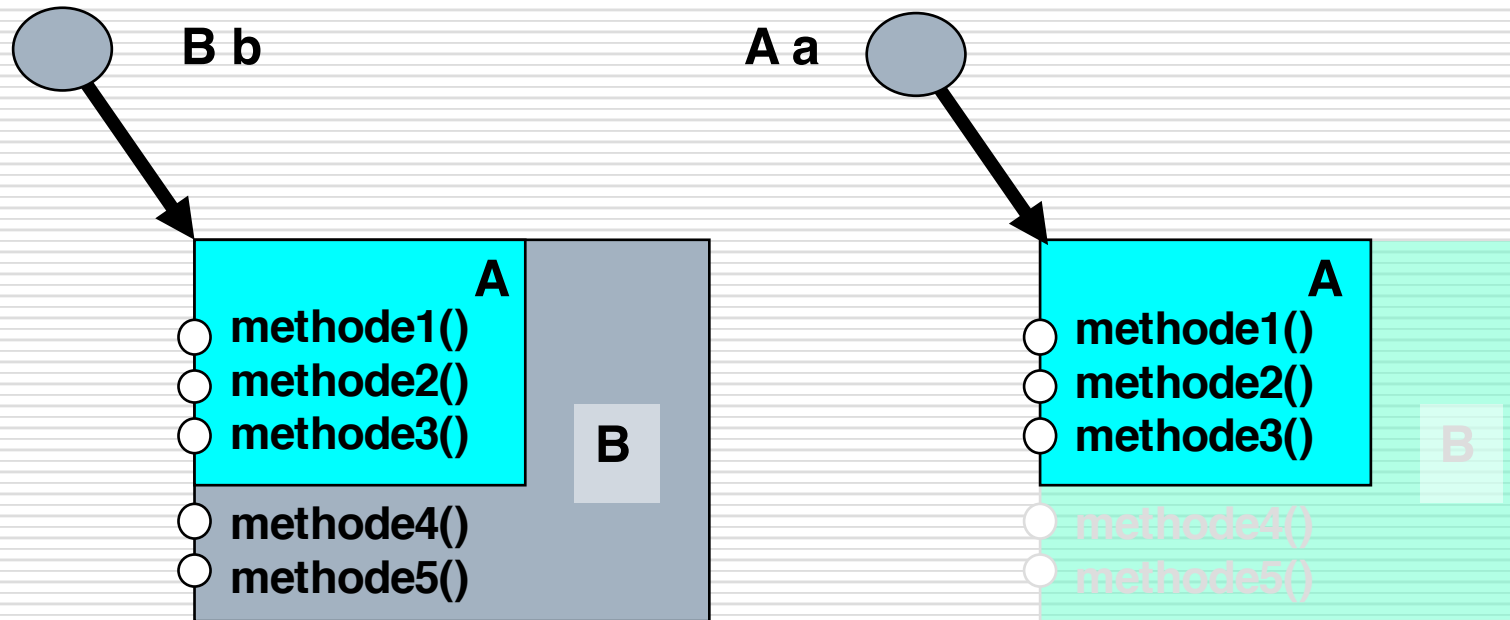
## □ créations classiques

```
A a = new A(); // référence de type A vers un objet A
```

```
B b = new B(); // référence de type B vers un objet B
```

# Manipulation des références (2)

---



`B b = new B(); // est légal (1)`

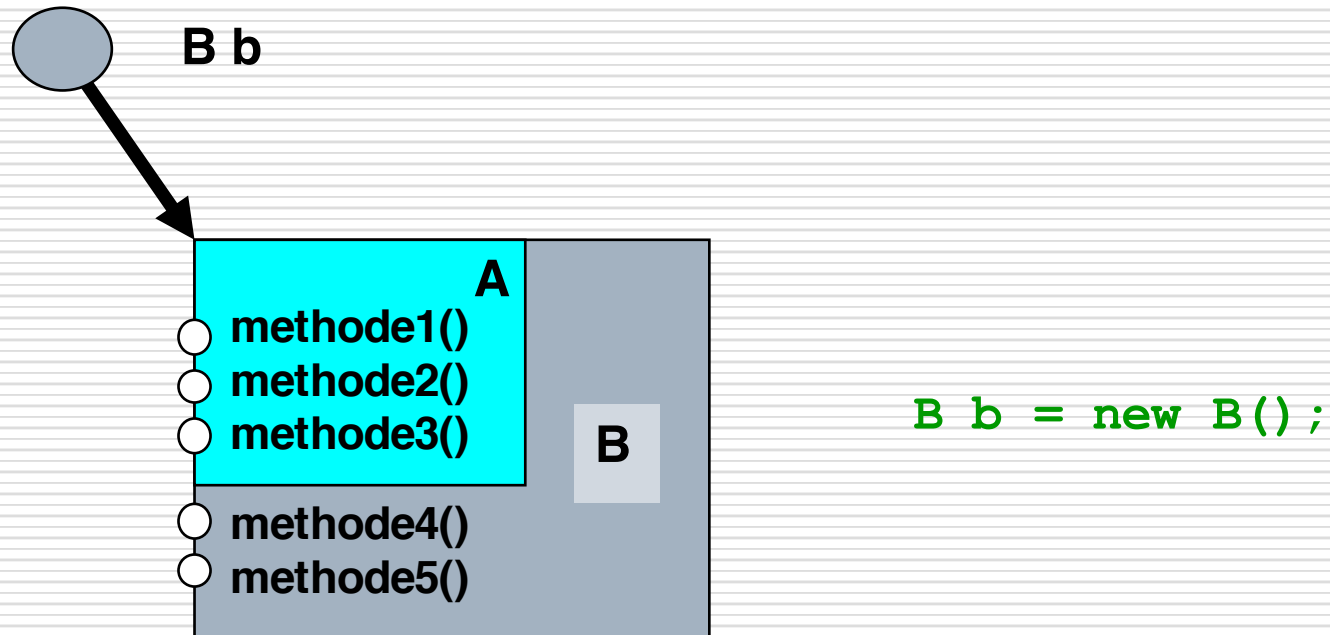
`A a = new B(); // est légal (2)`

`B b = new A(); // est illégal`

---

# Manipulation des références (3)

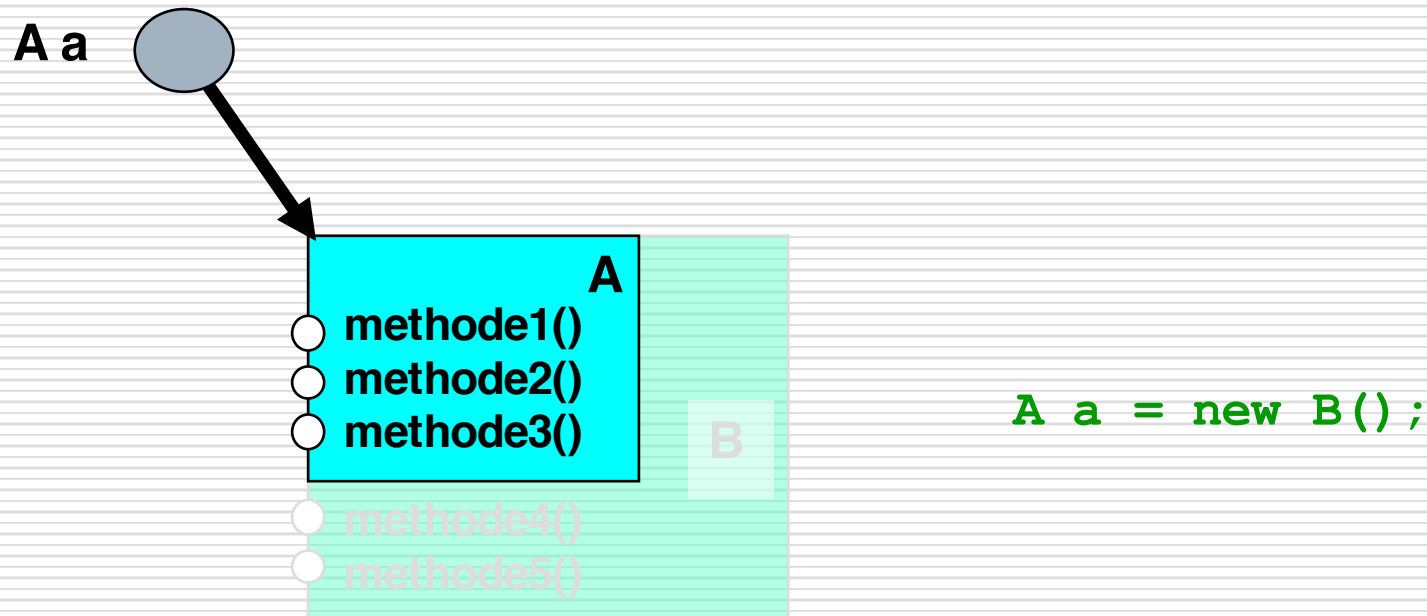
---



(1) `b` est un `B` vu comme un `B` : accès à tous les membres

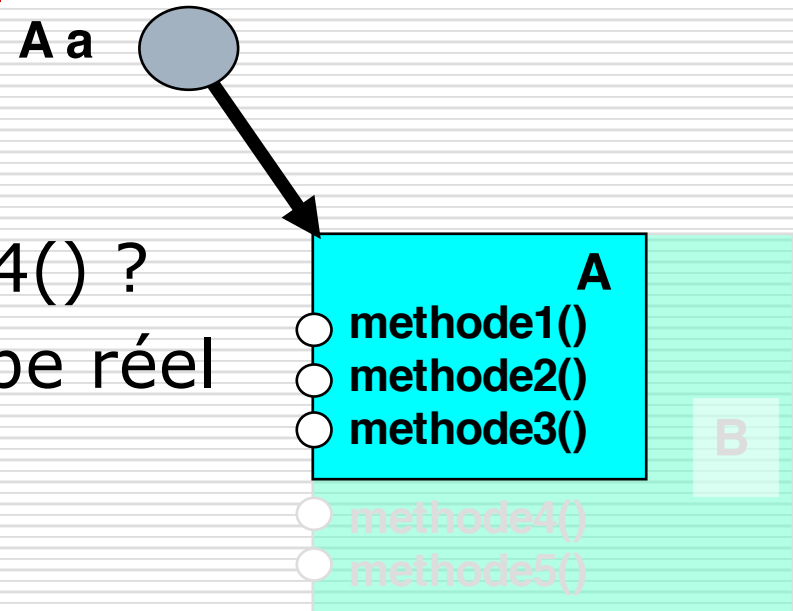
# Manipulation des références (4)

---



(2) `a` est un `B` vu comme un `A` : accès aux seuls membres de `A`

# Manipulation des références (5)



- ❑ Comment accéder à methode4() ?
- ❑ Solution : voir 'a' avec son type réel B par transtypage :

```
((B) a).methode4();
```

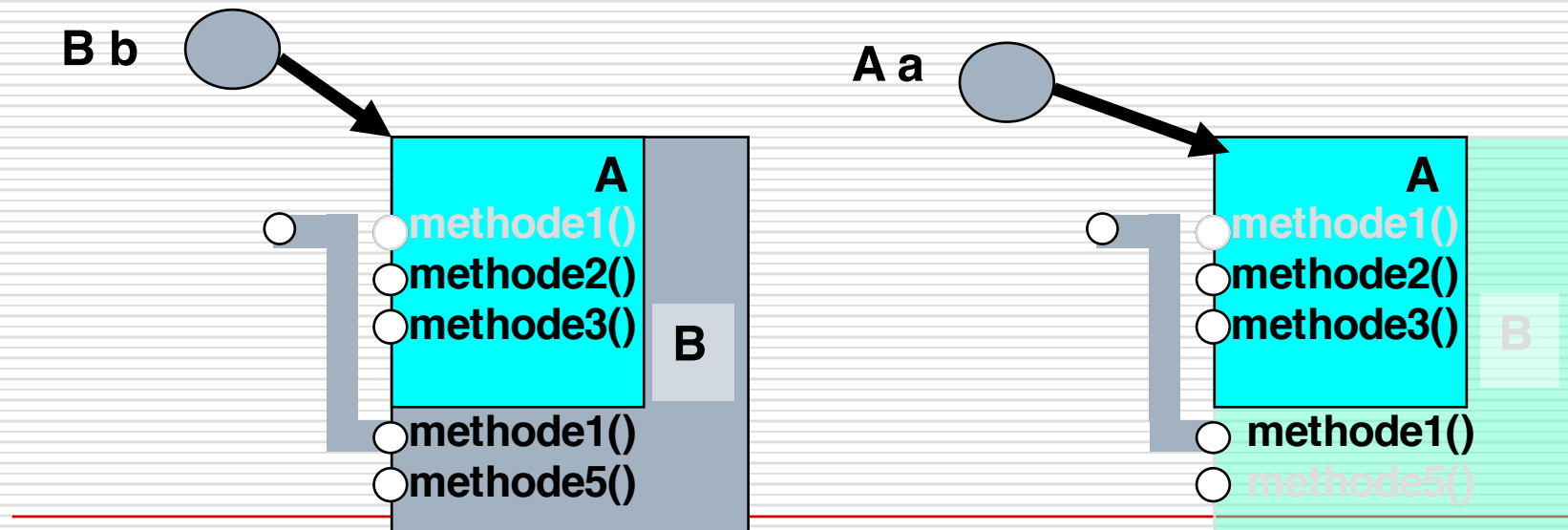
- ❑ **Attention** : il faut être sûr que 'a' soit un B. Sinon tester avec la méthode **instanceof** :

```
if (a instanceof B) { // a est-il un B ?  
    ((B) a).methode4();
```

```
}
```

# Redéfinition de méthodes (1)

- Redéclarer une méthode dans une classe fille est une « redéfinition ». Conséquences :
  - Pour les méthodes d'instance : la méthode **remplace** l'ancienne méthode, quelle que soit la façon dont on voit l'objet :



# Redéfinition de méthodes (2)

---

a.méthode1();

- Le compilateur vérifie les appels de méthodes par rapport au type déclaré d'une référence
  - méthode1() appartient à A a : OK
- La machine virtuelle exécute la méthode de la classe réelle de l'objet en cas de redéfinition: mécanisme de **liaison tardive**
  - méthode1() redéfinie dans B : c'est elle qui est exécutée et non celle de A

# Redéfinition vs surcharge

---

- ❑ Une méthode est redéfinie ssi elle a la même signature que l'originale
- ❑ Si la signature est différente, il s'agit d'une simple surcharge : pas de mécanisme de liaison tardive
- ❑ Pour éviter des erreurs , ajouter annotation `@Override`.
  - le compilateur vérifie qu'il s'agit bien d'une redéfinition
  - documente le code

# Annotations

---

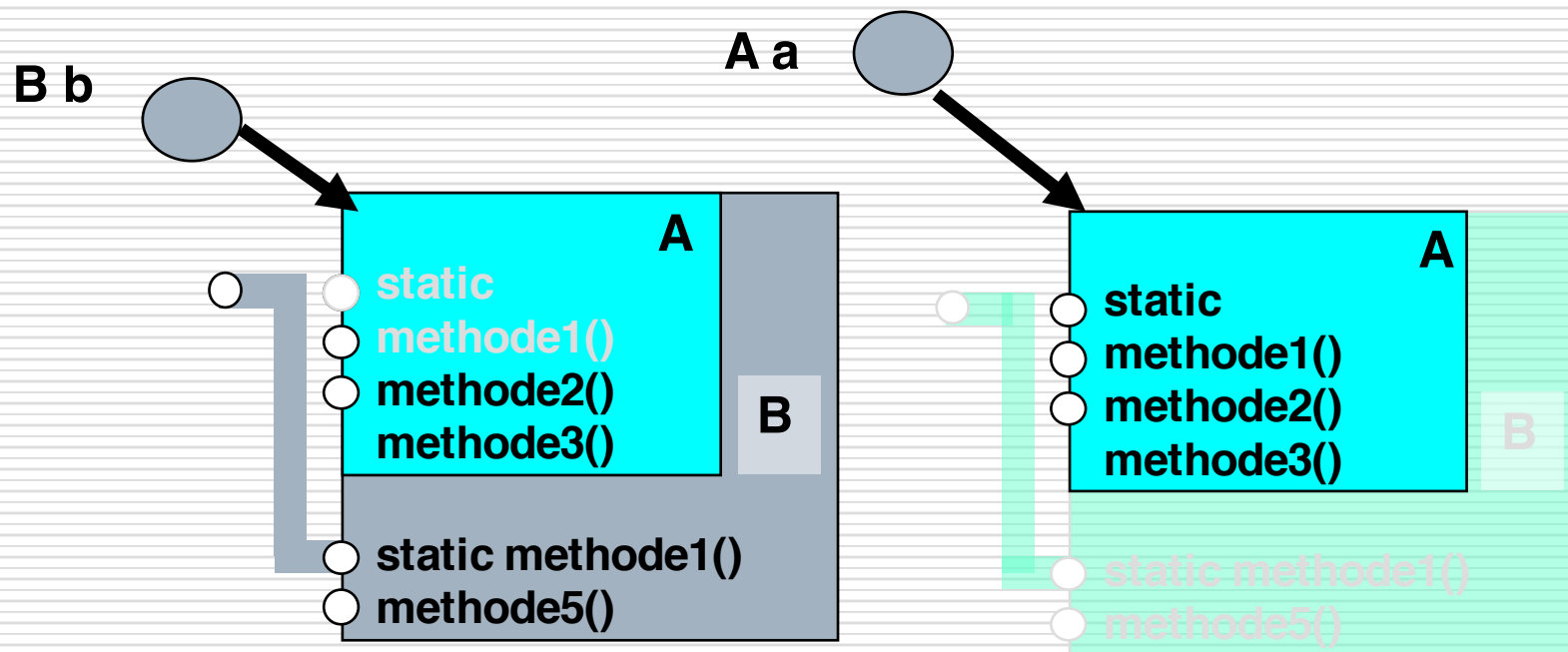
- Annotation = métacode
  - interaction avec le compilateur
  - utilisation par la JVM

- Exemple de redéfinition :

```
public class B extends A {  
    ...  
    @Override  
    public void méthode1() { ... }  
}
```

# Redéfinition de méthodes (3)

- Redéfinition pour une méthode de classe :
  - Pour les méthodes de classe : la méthode **masque** l'ancienne méthode, lorsqu'elle est accessible :

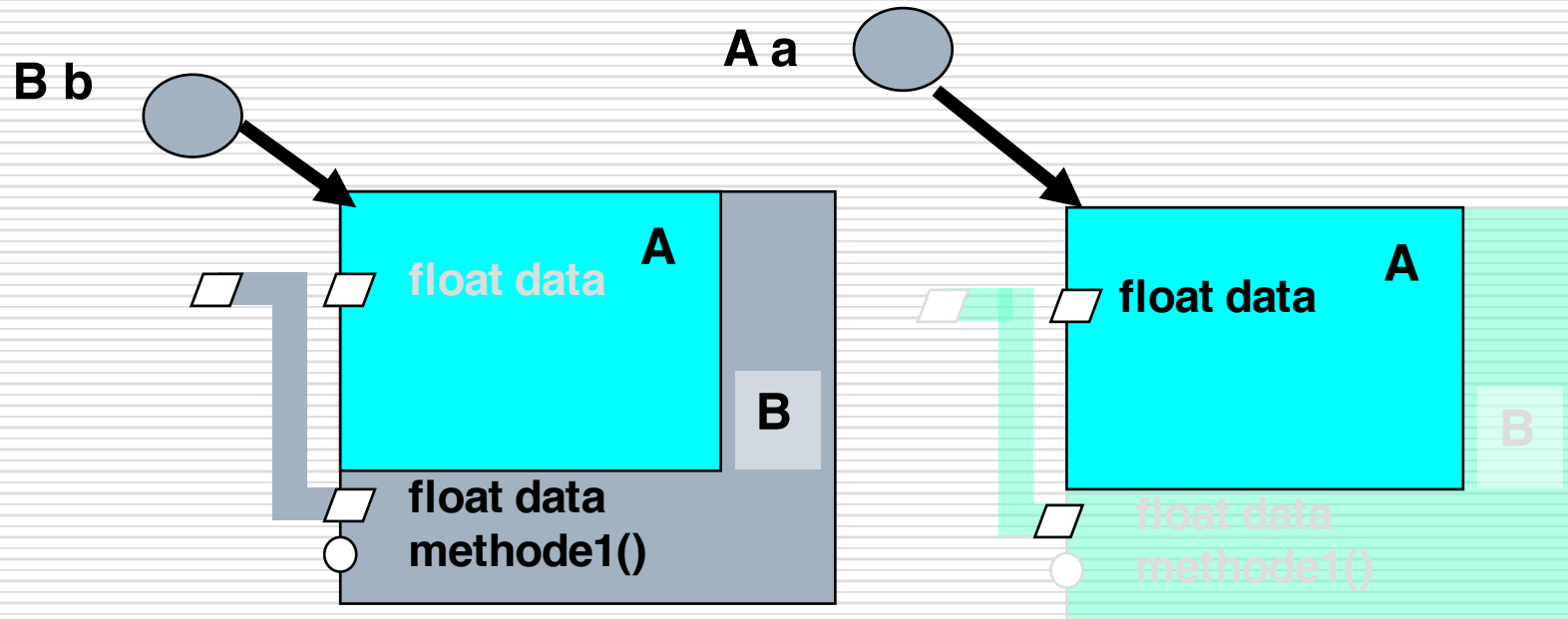


- NB : normal car `a.methode1() ≡ A.méthode1()`

# Masquage de champ

---

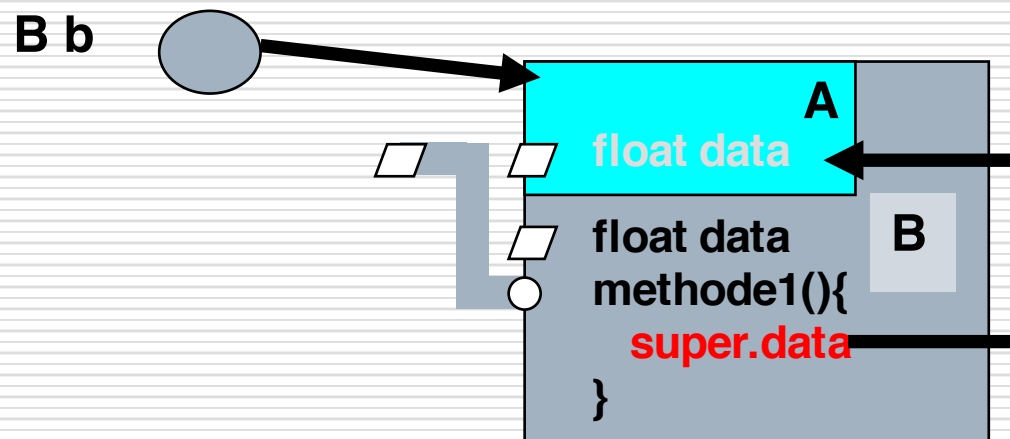
- Un champ redéfini dans une sous-classe est masqué



# Usage de `super`

---

- Permet d'accéder à une définition cachée dans la superclasse
  - s'utilise lorsqu'il y a masquage d'un membre parent
  - s'utilise pour appeler un super constructeur spécifique



# Superclasse absolue Object

---

- ❑ Est racine absolue des classes
- ❑ Est superclasse implicite des classes qui ne déclarent rien
- ❑ Contient quelques données et méthodes transcendantes à la nature "objet"

# Superclasse Object (2)

---

- Numérote les objets de façon unique :
  - hashCode()
- Installe les infrastructures élémentaires :
  - clone() pour la copie
  - equals() pour la comparaison
  - getClass() pour l'identité de classe
  - toString() pour l'observabilité
  - finalize() pour le nettoyage

# Superclasse Object : equals

---

- equals() : comparaison champs à champs de 2 objets
  - redéfinie de cette façon dans les classes du JDK
  - à faire par le programmeur pour ses classes
    - par défaut équivalent à == (égalité de références pour les objets)

# Superclasse Object : equals (2)

---

```
public class Personne {  
    ...  
    @Override  
    public boolean equals(Object autreObjet) {  
        if (autreObjet==null || !autreObjet instanceof Personne) {  
            return false;  
        }  
        Personne autrePersonne = (Personne) autreObjet;  
        return (this.nom.equals(autrePersonne.getNom())  
            && this.prenom.equals(autrePersonne.getPrenom())  
            && this.age == autrePersonne.getAge());  
    }  
}
```

---

# Superclasse Object : toString

---

- toString() : représentation sous la forme d'une chaîne de caractère
  - à redéfinir par le programmeur pour ses classes
  - par défaut : nom de la classe + '@' + hashCode
  - utilisations :
    - public void println(Object o)
      - => imprime o.toString()
    - public String "+"(String s, Object o)
      - => s + o.toString()

# Gestion de "l'héritabilité"

---

- On peut contrôler la "dérivabilité" de certaines classes ou méthodes
  - méthodes finales : bloque la redéfinition
  - classes finales : bloque la dérivabilité. } altérateur **final**
- Exemples :
  - `public final class Etudiant { ... }`  
// on ne pourra pas spécialiser cette classe
  - `public final double PI = 3.1415926535; // 1 constante`
  - `public class Professeur {`  
    `public final void noter() { ... }`  
    // 1 sous-classe de Professeur ne pourra pas redéfinir  
    // cette méthode  
}

## Gestion de "l'héritabilité" (2)

---

- ❑ Les droits d'accès ont une influence sur l'héritage
- ❑ Un attribut privé n'est pas hérité
- ❑ Privé (mère) -> inaccessible / non redéfinissable
- ❑ Protected (mère) -> accessible / redéfinissable
- ❑ Package (mère) -> ssi dans le même package
- ❑ Public (mère) -> accessible en tout temps

# Classe abstraite

---

- ❑ Mot clé : abstract
- ❑ Exemple :

```
public abstract class C {  
    private String unChamps;  
    public void méthode1concrète(...) {  
        // code  
    }  
    public abstract void méthode2abstraite(...);  
}
```

# Classe abstraite (2)

---

- ❑ pas d'instanciation possible
  - `new C(...); // illégal`
- ❑ constructeur possible :
  - mutualisation de code
  - initialisation des champs de la classe abstraite
  - appel avec `super(...)` dans la classe fille
- ❑ manipulation de référence possible  
`public class D extends C { ... } // classe concrète`

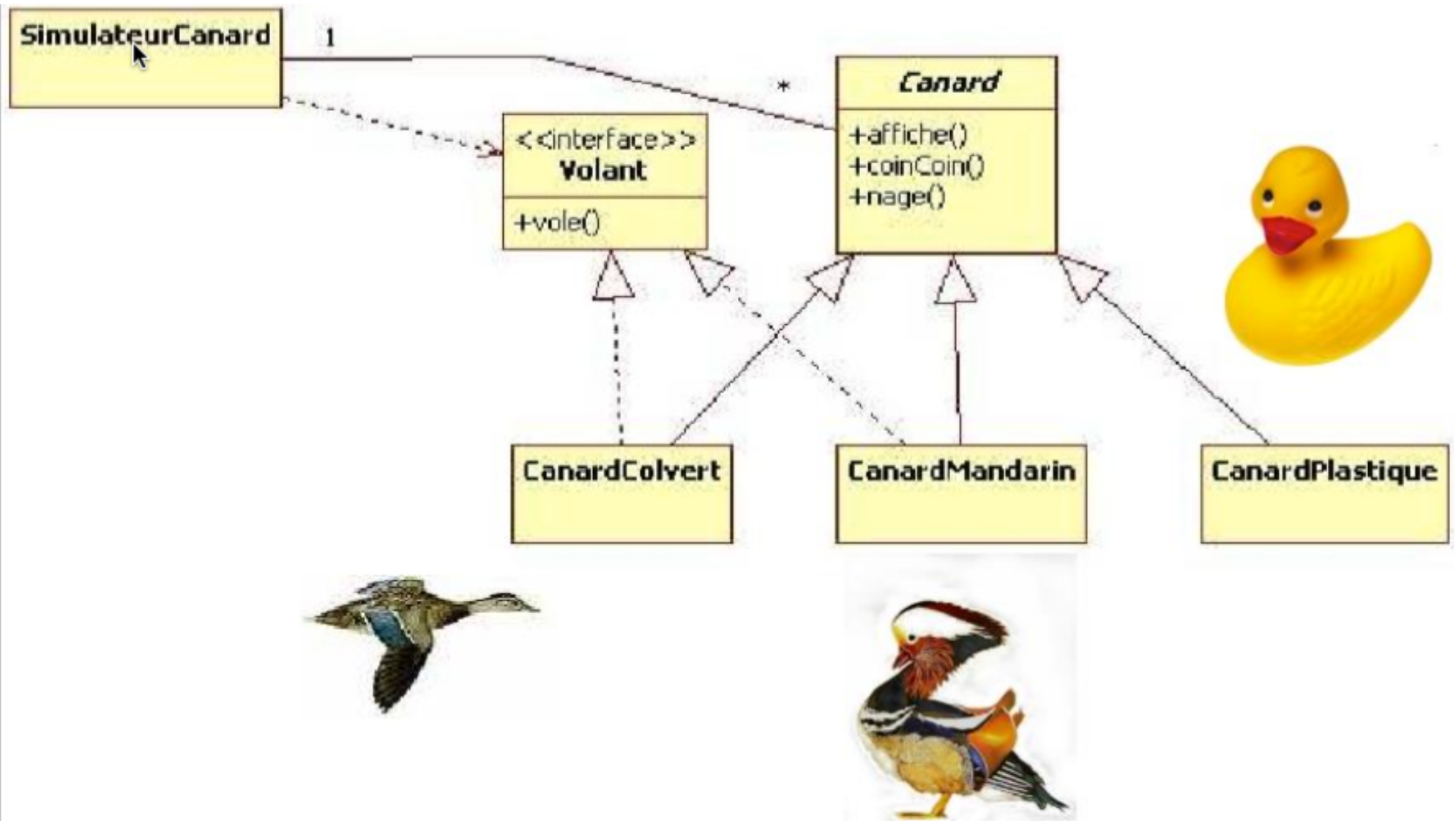
`C refC = new D(); // légal`

---

# Interfaces

---

- ❑ Une interface ne contient pas de code
- ❑ Elle indique les méthodes que les classes doivent implémenter



# Interfaces

---

- 1 interface
  - mot-clé : interface
  - Liste des méthodes publiques

- Exemple

```
public interface Volant {  
    void vole();  
}
```

# Interfaces

---

- ❑ Réaliser / implémenter une interface
  - Mot-clé : implements
- ❑ Les méthodes sont publiques
- ❑ Exemple

```
public class CanardColvert
    extends Canard
    implements Volant {

    @Override
    public void vole() { //code }

}
```

```
public class SimulateurCanard {
    // les canards gérés par le simulateur
    private Set<Canard> canards;

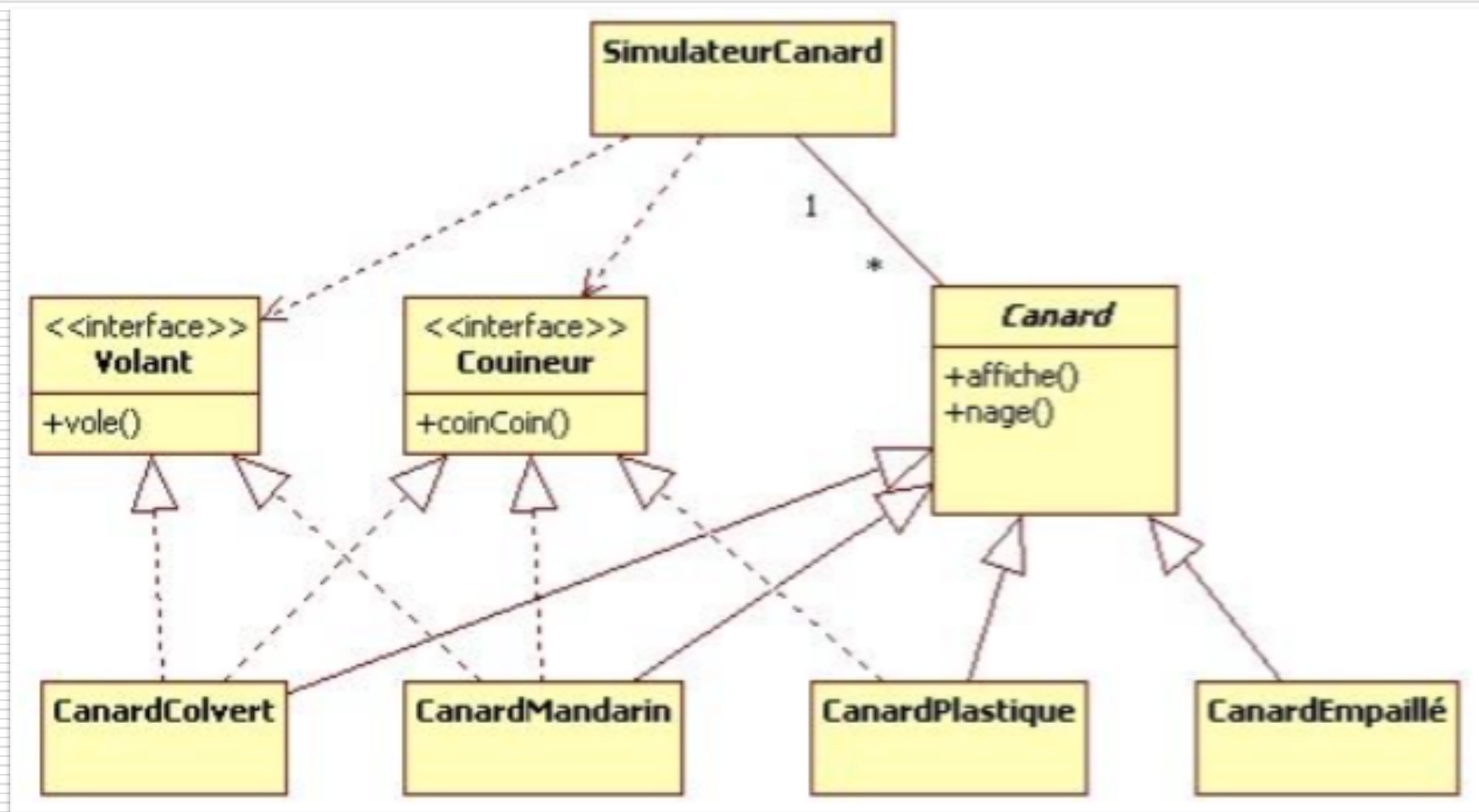
    private static void deplacerCanardVolant(Volant volant) {
        volant.vole();
    }

    public animerCanards() {
        for (Canard canard : canards) {
            canard.coinCoin();
            if (canard instanceof Volant) {
                // on se déplace plus vite en volant
                deplacerCanarVolant((Volant) canard);
            } else {
                // tant pis, on rame un peu à la nage
                canard.nage();
            }
        }
    }
}
```

# Implémentations multiples

---

- ❑ 1 classe peut implémenter plusieurs interfaces
- ❑ Les interfaces peuvent avoir plusieurs méthodes communes
- ❑ La classe devra apporter le code de l'union des méthodes de chaque interface



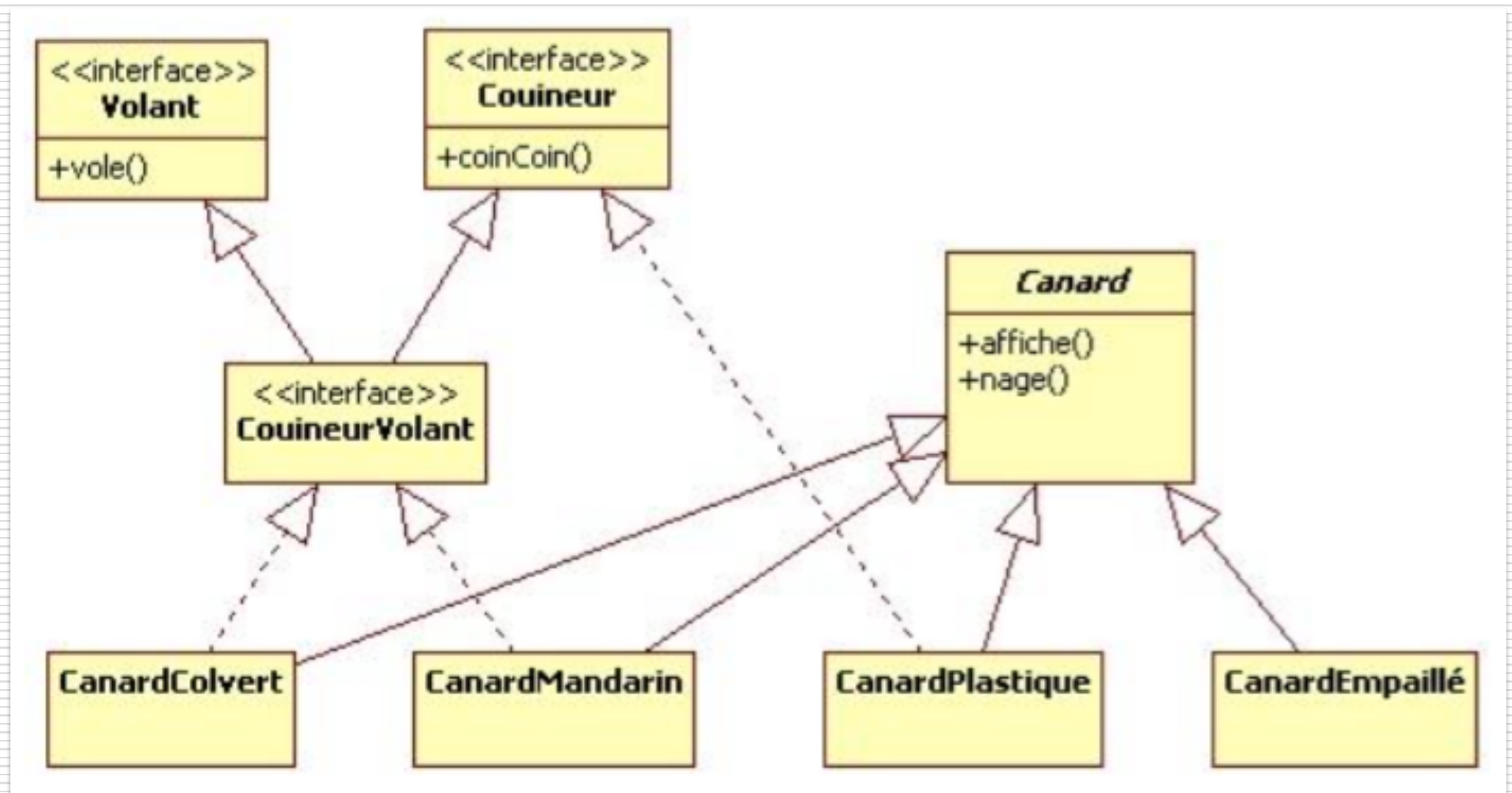
public class CanardColvert  
 extends Canard

implements Volant, Couineur { ... }

# Héritage entre interfaces

---

- ❑ 1 interface peut hériter d'une ou plusieurs interfaces
- ❑ Les interfaces mères peuvent avoir des méthodes communes
- ❑ L'interface fille récupère l'union des méthodes (signatures) de chaque interface



**public interface CouineurVolant  
extends Volant, Couineur**