Plan du chapitre

1
Le paradigme objet

2
Les objets

3
Les classes

4
Associations entre classes

5
Héritage et polymorphisme

Héritage

- Classes
- Relations
  - Association
  - Dépendance
  - Héritage

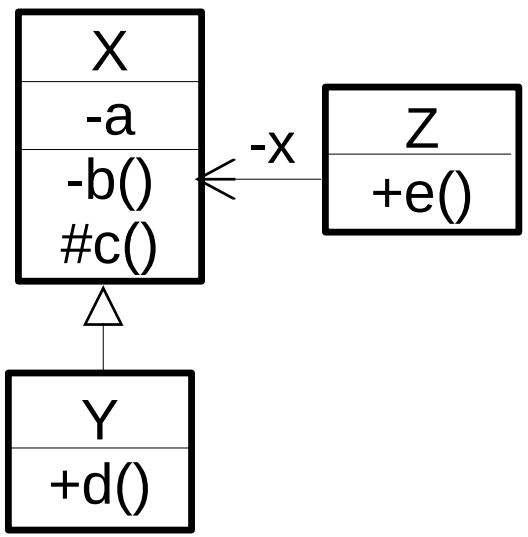
Visibilité

- Limiter l'accès aux membres des classes (attributs, méthodes, associations)
- Intention : sécuriser la représentation interne des classes (encapsulation)

	Notation UML	Classe	Classe (même pkg)	Sous-classe (même pkg)	Sous-classe (diff pkg)	Classes (diff pkg)
public	+	+	+	+	+	+
protected	#	+	+	+	+	-
package	~	+	+	+	-	-
private	-	+	-	-	-	-

Visibilité : règle

- Afin de garantir l'encapsulation
  - Restreindre le plus possible la visibilité
  - **Les attributs et associations sont TOUJOURS privés (règle incontournable)**
  - Finalement, la visibilité est une préoccupation qui ne concerne que les services (on pourrait omettre de marquer la visibilité sur les attributs et les associations).



Soit trois variantes de la méthode Y::d(), les codes suivants sont-ils compilables ?

```
1. void d() { a=5; }
2. void d() { b(); }
3. void d() { c(); }
```

Soit quatre variantes de la méthode Z::e(), les codes suivants sont-ils compilables ?

```
4. void e() { x.a=5; }
5. void e() { x.b(); }
6. void e() { x.c(); }
7. void e() { x.d(); }
```

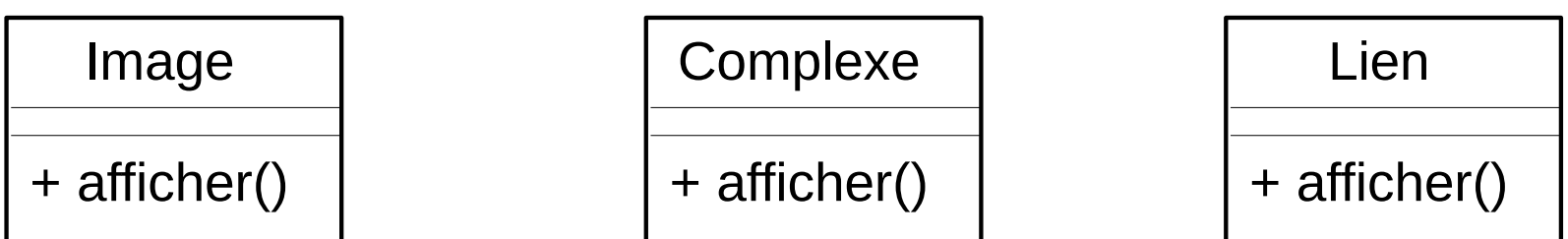
Ce code de la méthode X::c() est-il compilable ?

```
8. void c() {
       X x = new X();
       x.b();
   }
```

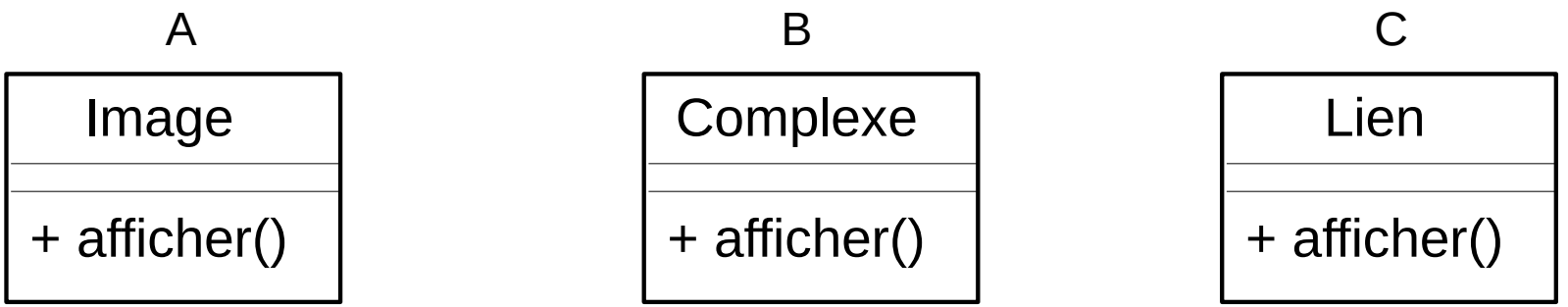
9. Et ce même code dans le main() ?

Portée des noms

- Il est possible de donner un même nom à des méthodes de classes différentes.
- La portée des noms est locale.
- Le choix est fait à la compilation : liaison statique.



Portée des noms



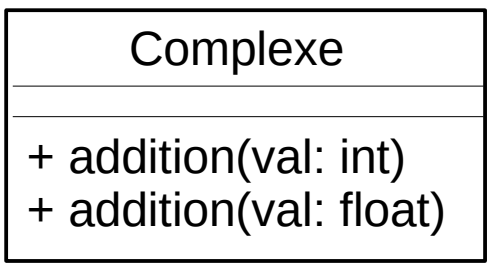
Quelle méthode est exécutée (A, B ou C) ?

```
Image i = new Image();
i.afficher();
```

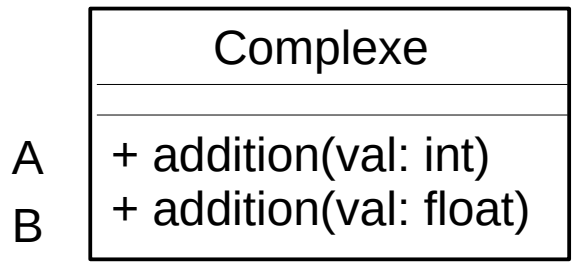
Surcharge

- Dans une même portée des méthodes de même nom mais avec des signatures différentes.
- Attention : Signature d'une méthode :

  nom + paramètres (pas le type de retour ni les exceptions.)
- Le choix peut être fait à la compilation : liaison statique.



Surcharge

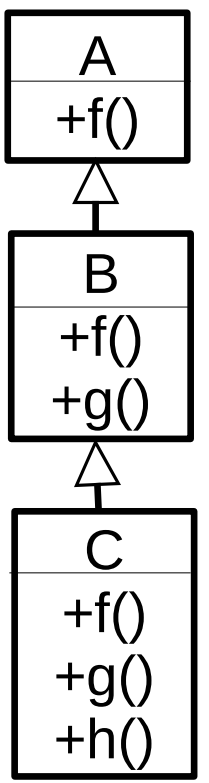


Quelle méthode est exécutée (A ou B) ?

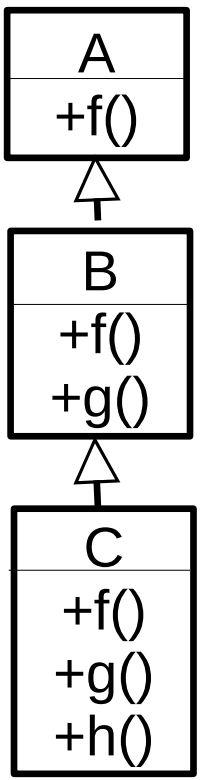
```
Complexe c = new complexe();
c.addition(5);
c.addition(5F);
c.addition(5D);
```

Polymorphisme

- Donner la même signature à des méthodes d'une même hiérarchie.
  - La méthode exécutée est celle qui est la plus proche de la classe réelle de l'objet appelant en remontant dans la hiérarchie.



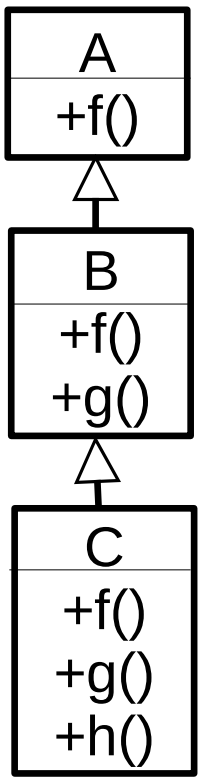
Polymorphisme



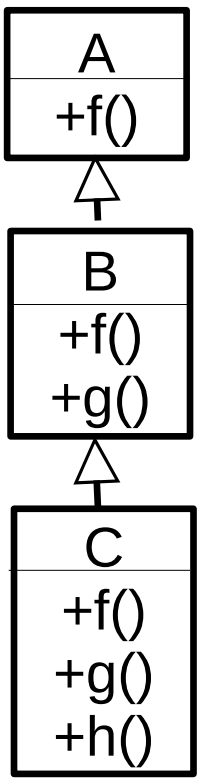

```
A a = new C();
a.f();
a.g();
a.h();
B b = (B)a;
b.f();
b.g();
b.h();
C c = (C)b;
c.g();
```

Polymorphisme

- Le choix de la méthode ne peut pas être décidé à la compilation mais seulement à l'exécution.



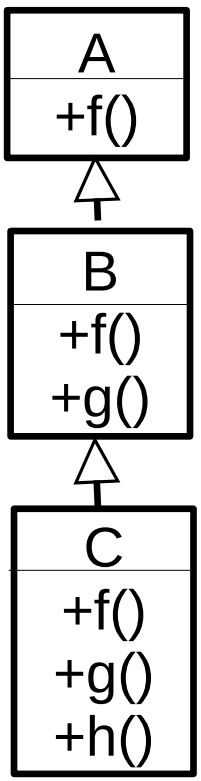
Polymorphisme




```
A getObject(int id) {
   switch(id) {
       case 1: return new B();
       case 2: return new C();
   }
}

A a = getObject(1);
a.f();
```

Polymorphisme



Les classes dérivées ne peuvent pas diminuer la visibilité d'une méthode redéfinie

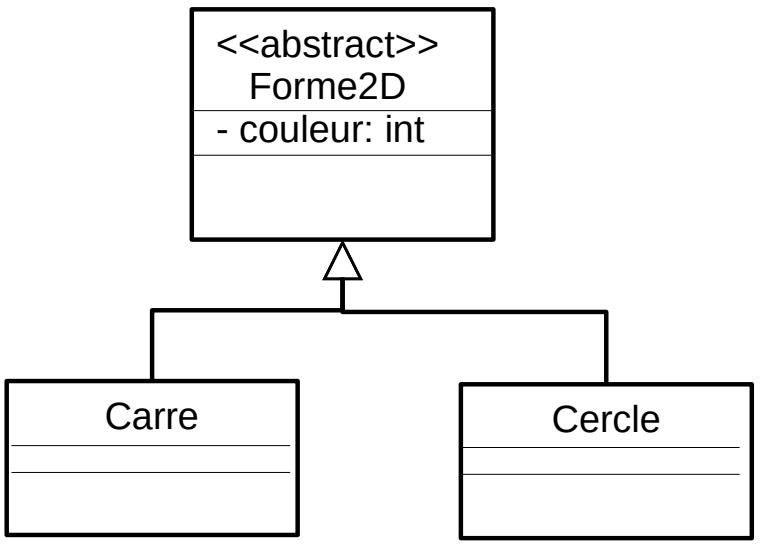
Pourquoi ?

```
A getObject(int id) {
    switch(id) {
       case 1: return new B();
       case 2: return new C();
    }
}

A a = getObject(1);
a.f();
```

Classe abstraite

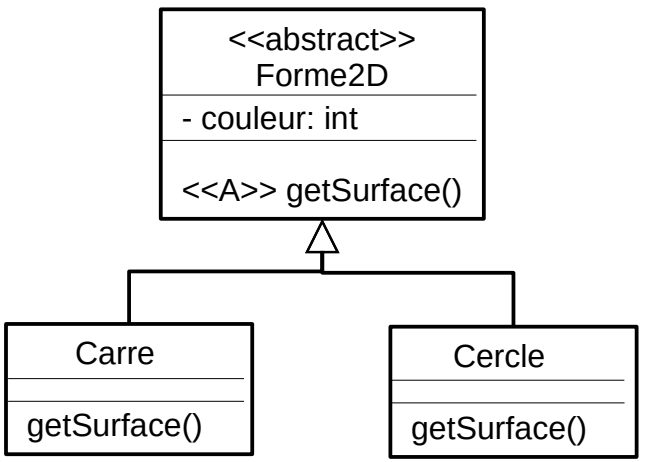
- Classe sans instance
  - Interdire la création d'instances puisqu'elles n'ont pas de sens. La classe abstraite n'est pas assez complète pour être instanciée.



- En Java
  - Mot clé abstract devant la classe.

Méthode abstraite

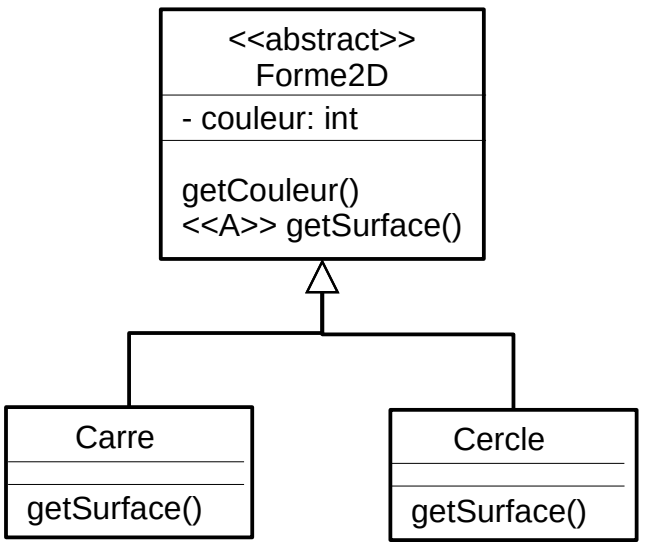
- Méthode sans code
  - Obliger les sous-classes à définir le code de la méthode
  - Profiter du polymorphisme pour exécuter la bonne méthode.
  - Exemple :
    - Forme2D f = new Carre();
    - double s = f.getSurface();



- En Java
  - Le mot clé abstract devant la méthode.
  - Pas de code pour la méthode

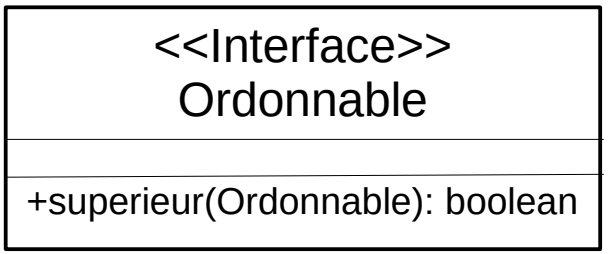
Classe et méthode abstraites

- Une classe avec une méthode abstraite est forcément abstraite
- Une classe abstraite peut ne contenir que des méthodes concrètes



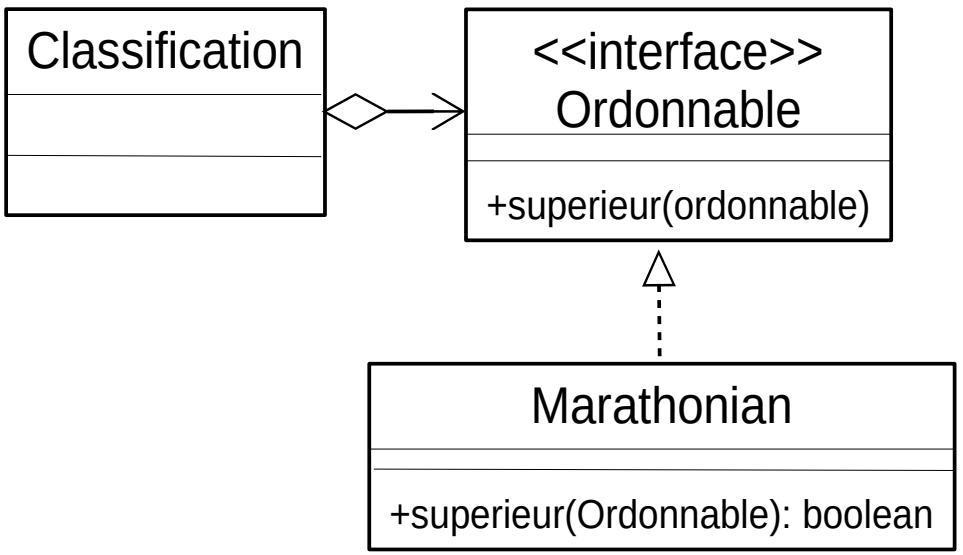
Interface

- Une classe avec que des méthodes **abstraites pures** et sans attribut ni association
- Une interface définit un type
  - Elle impose les méthodes que doivent posséder toutes les classes qui implémentent l'interface.



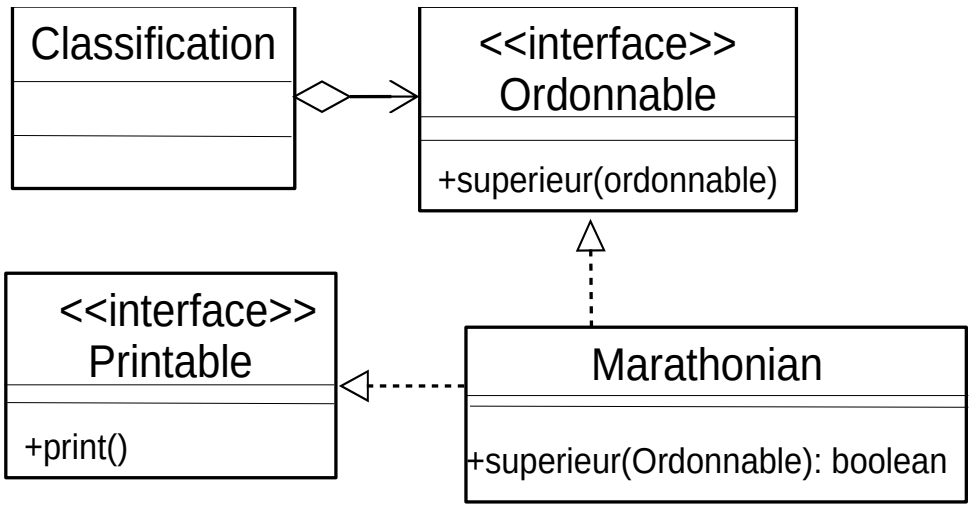
Interface

- Contrat en deux classes :
  - Une classe fournit un service
  - Une classe utilise le service



Interface

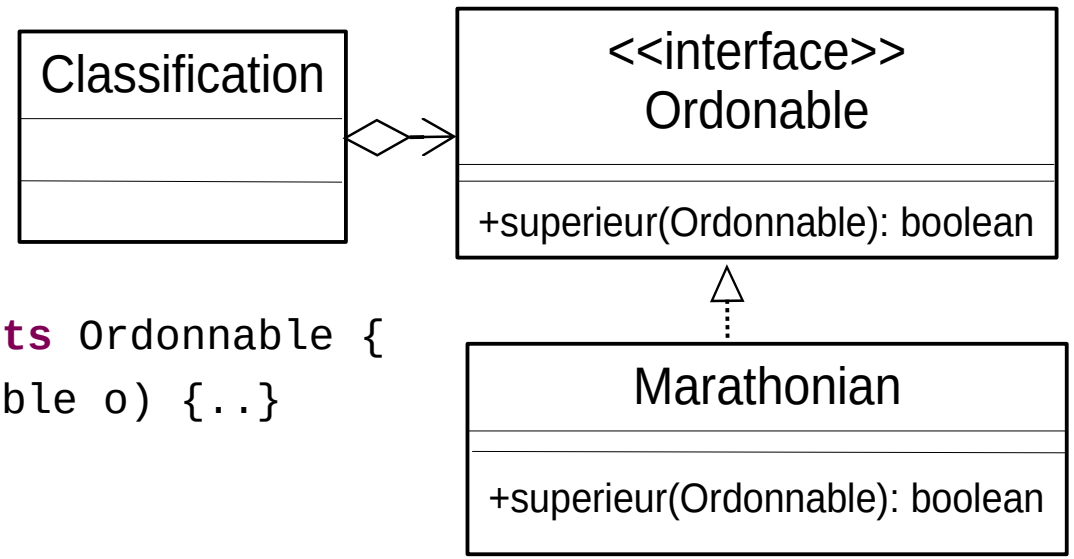
- Une classe peut implémenter plusieurs interfaces
- On peut définir des références vers une interface
  - Ordonnable o = new Marathonian() ;
  - ~~Ordonnable o = new Ordonnable() ;~~



Interface en Java

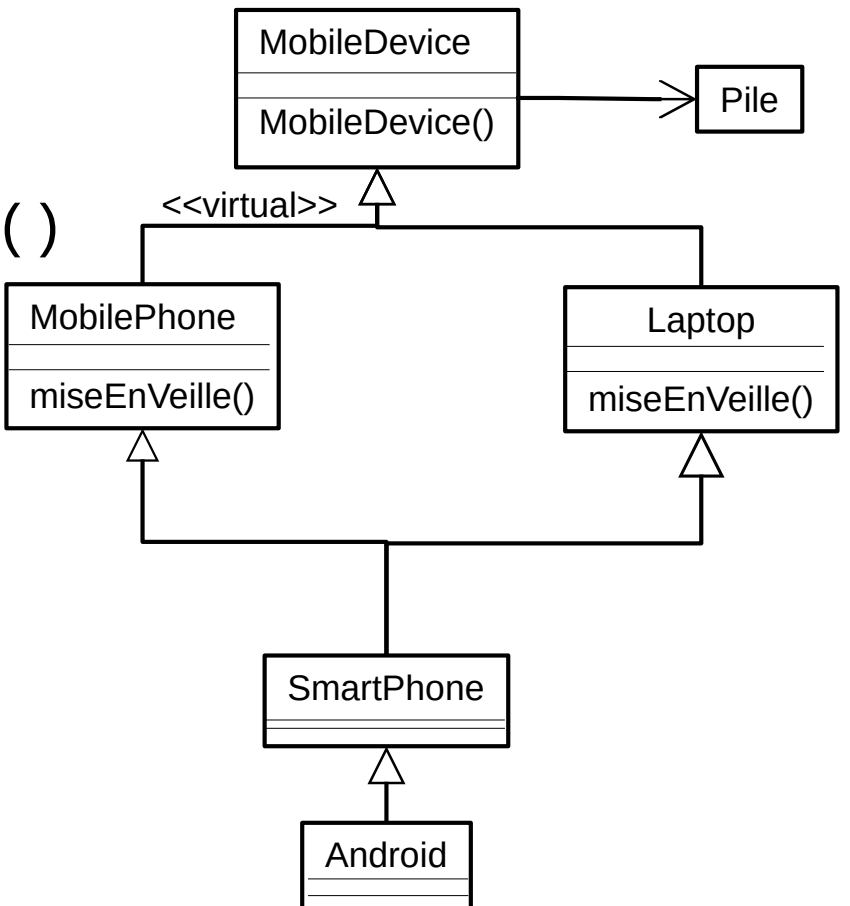
- Notation Java :

```
public interface Ordonnable {
  public boolean superieur();
}
public class Marathonian implements Ordonnable {
  public boolean superieur(Ordonable o) {..}
  // méthodes spécifiques
}
public class Classification {
    ArrayList<Ordonnable> coureurs;
    …
    coureurs.get(1).superieur(coureurs.get(2));
}
```

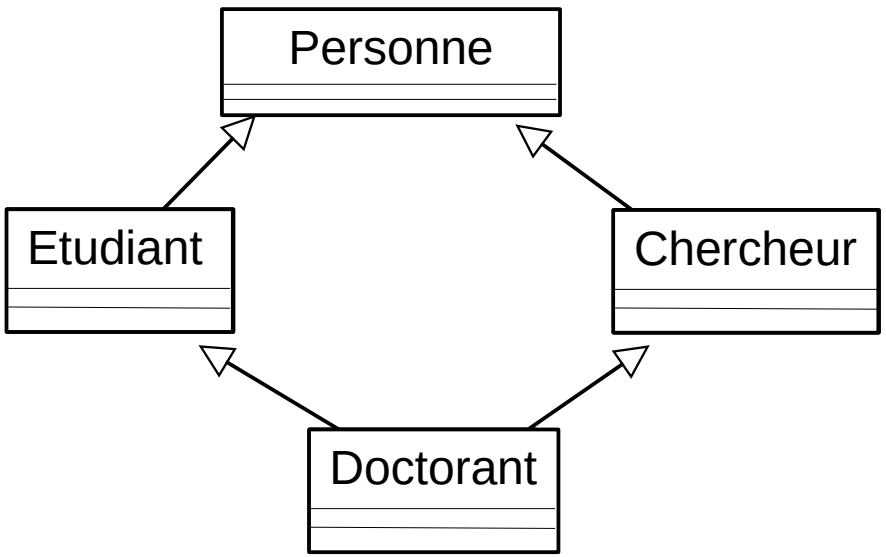


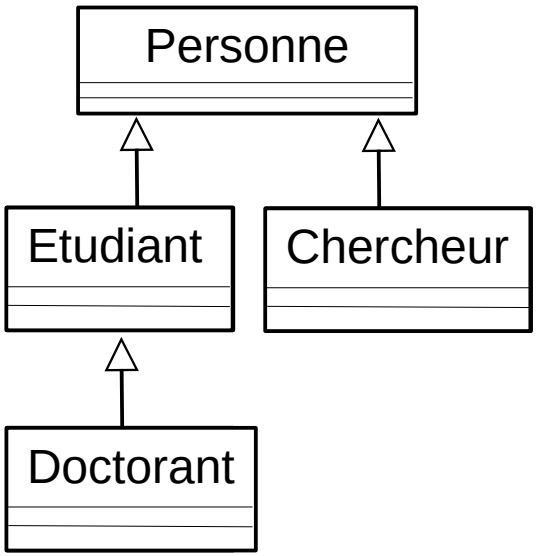
Cas de l'héritage multiple

- Questions
  - Combien de téléphone ?
  - Quelle méthode est appelée ?
    - `t = new TelephoneAndroid()`
    - `t.miseEnVeille();`

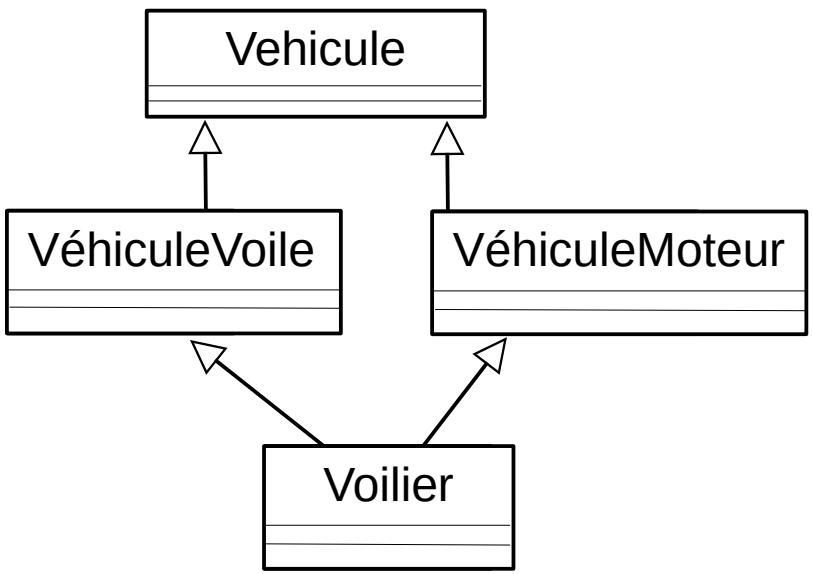


Avons nous besoin de l'héritage multiple ?





Avons nous besoin de l'héritage multiple ?



Que retenir de ce chapitre ?

- Le paradigme objet définit plusieurs concepts importants :
  - Classe et objet.
  - Encapsulation.
  - Relations.
    - Association.
      - Standard, Agrégation, Composition, Dépendance.
    - Héritage et polymorphisme.
    - Interface et type.

Que retenir de ce chapitre ?

- Dans l'utilisation de ces concepts, le développeur doit respecter deux principes fondamentaux :
  - Restreindre le plus possible la visibilité des membres pour respecter le principe d'encapsulation et la sécurité.
    - Ainsi les attributs sont TOUJOURS privés.
  - Exprimer le plus de choses statiquement pour être vérifiables par le compilateur, ie, ne pas repousser les choix au moment de la programmation.