

Chapitre 4 : Héritage et ses implications

- Principes de l'héritage
- redéfinition et surcharge
- contrôle de l'héritage
- polymorphisme
- classes abstraites
- Interfaces

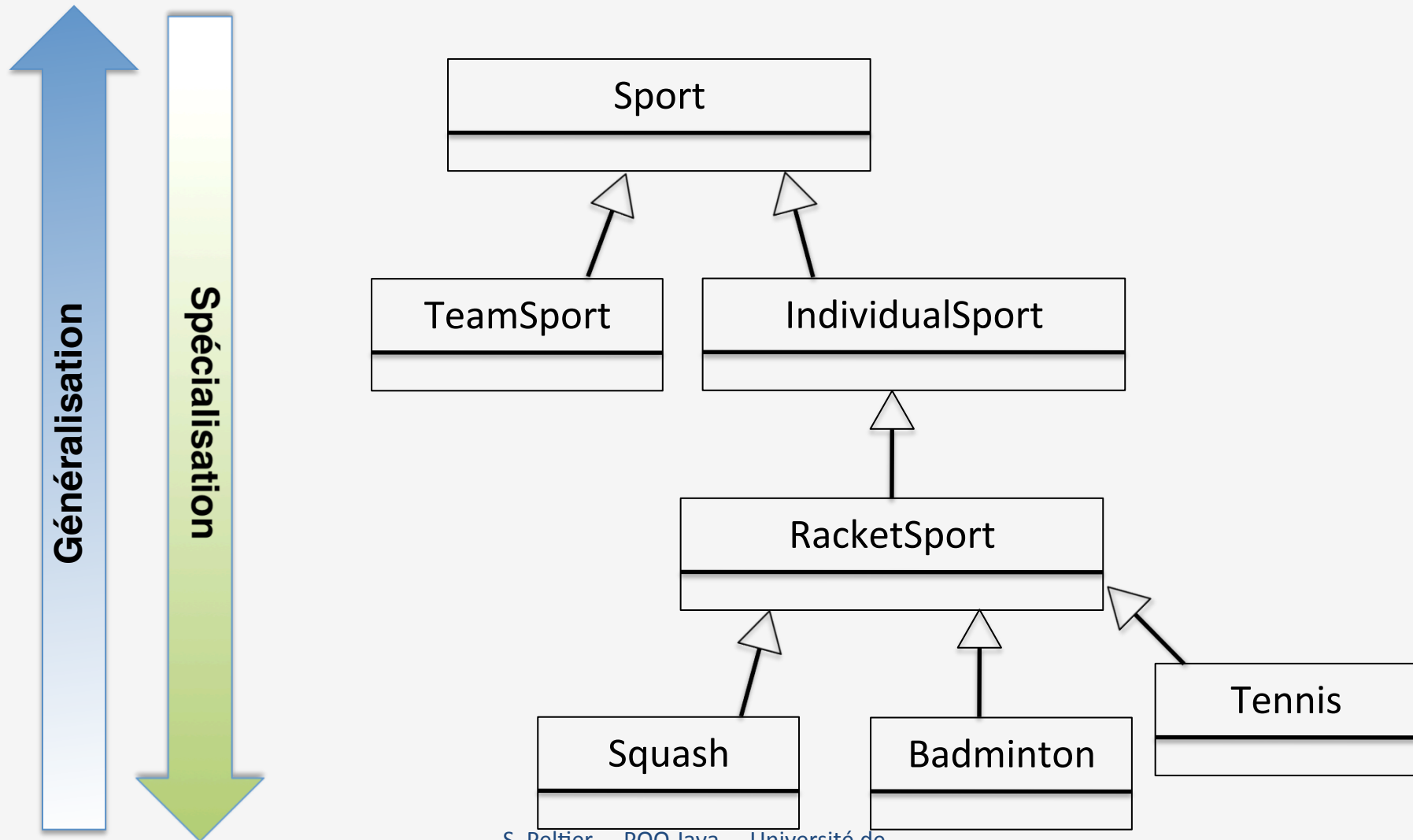
Chapitre 4 : Héritage et ses implications

- Jusqu'à présent, nous avons vu comment construire des classes et définir des associations.
- Mais un des principaux intérêts de la POO est de pouvoir étendre aisément un programme en l'enrichissant sans tout reprogrammer.

Intérêts de l'héritage

- Au moment de la conception :
 - s'appuyer sur des classes existantes et testées
 - utilisation du polymorphisme
- Au moment de la maintenance :
 - corriger des erreurs dans une classe mère corrige du même coup les mêmes erreurs dans toutes les classes dérivées

Un exemple d'héritage



Chapitre 3 : Héritage et ses implications

- Le mécanisme de l'**héritage** permet de définir une classe (fille) à partir d'une autre classe (mère).
- La classe fille
 - contient tous les attributs/méthodes de la classe mère,
 - peut être dotée d'attributs/méthodes supplémentaires,
 - peut redéfinir des méthodes de la classe mère,
 - peut à son tour être dérivée.

Héritage

Spécialisation des objets :

- ajout d'attributs/méthodes : extension
- ajout de méthodes de même nom : surcharge
- modification de méthodes existantes : redéfinition

Héritage

Un objet de type ClasseFille est aussi du type ClasseMère. Autrement dit, tout ce que sait faire un objet de type ClasseMère, un objet de type ClasseFille sait le faire également.

- *Tester le type d'un objet avec **instanceof** :*

```
if (myInstance instanceof AClass) {  
    ...  
}
```

Héritage

- Vocabulaire :
 - Lien direct : classe mère, classe fille
 - Lien hiérarchique : sous-classe, sur-classe
 - La classe fille hérite de la classe Mère
- La syntaxe Java :

```
public class Boat extends Vehicle {  
...  
}
```


Héritage et constructeurs

- Un constructeur de la classe fille fait toujours appel à un constructeur de la classe mère.
- Soit de manière **explicite** :
`super(. . .) ;`

Cet appel explicite doit être effectué en première instruction.

Héritage et constructeurs

- Soit de manière **implicite** :
si l'instruction `super(...)` n'est pas explicitée dans le constructeur, alors java insère l'instruction `super ()`.

Appel implicite au constructeur par défaut de la classe mère

- **Précondition : la classe mère doit avoir un constructeur par défaut**

La visibilité protected

- Un élément **protected** (# en UML) est accessible depuis les sous-classes et les classes du package.

Note : Pour les attributs, il est fortement recommandé d'utiliser une visibilité private et d'accéder aux attributs avec une méthode (éventuellement protected).

La redéfinition de méthode

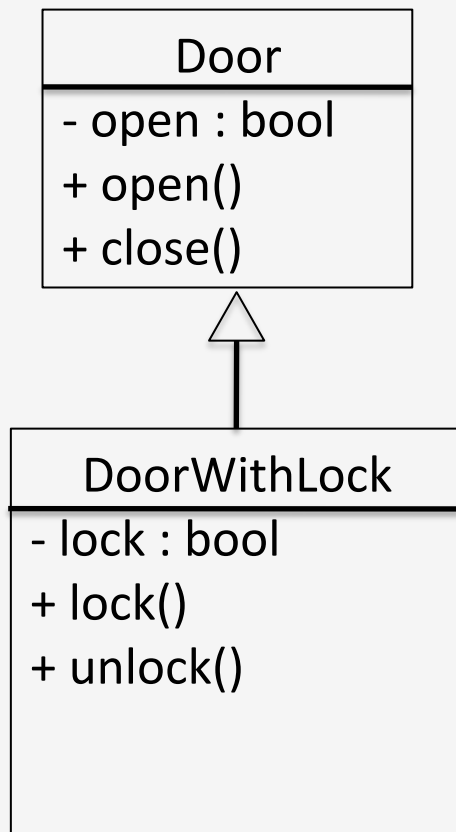
- **Principe** : donner une nouvelle implantation d'une méthode de la classe mère.

Pour cela il faut :

- même signature
- ne pas restreindre la visibilité de la méthode

La redéfinition de méthode

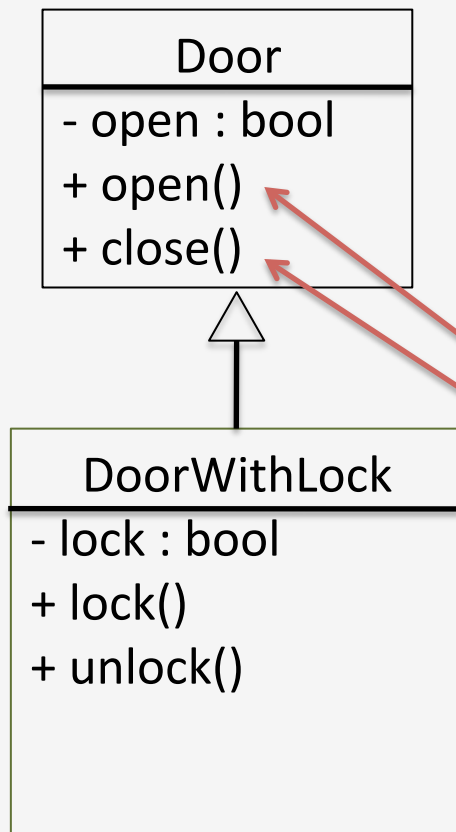
exemple :



Une porte s'ouvre et se ferme « directement », alors qu'une porte avec verrou ne s'ouvre que si le verrou n'est pas enclenché.

La redéfinition de méthode

exemple :

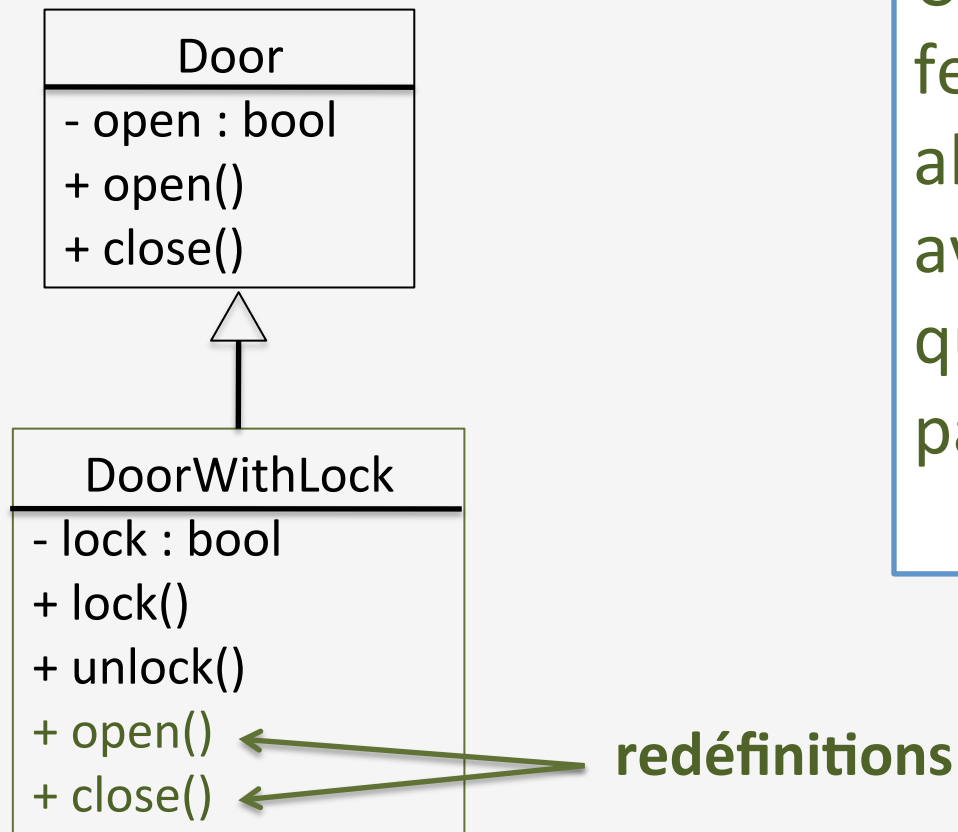


Une porte s'ouvre et se ferme « directement », alors qu'une porte avec verrou ne s'ouvre que si le verrou n'est pas enclenché.

PB : On peut ouvrir et fermer une porte verrouillée

La redéfinition de méthode

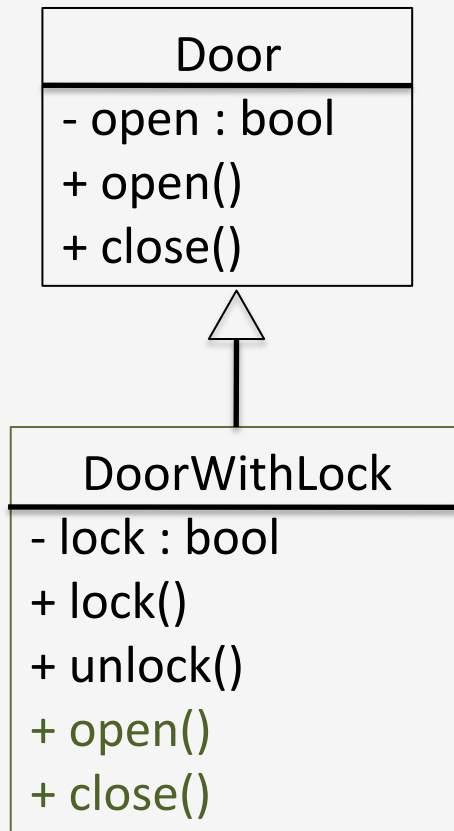
exemple :



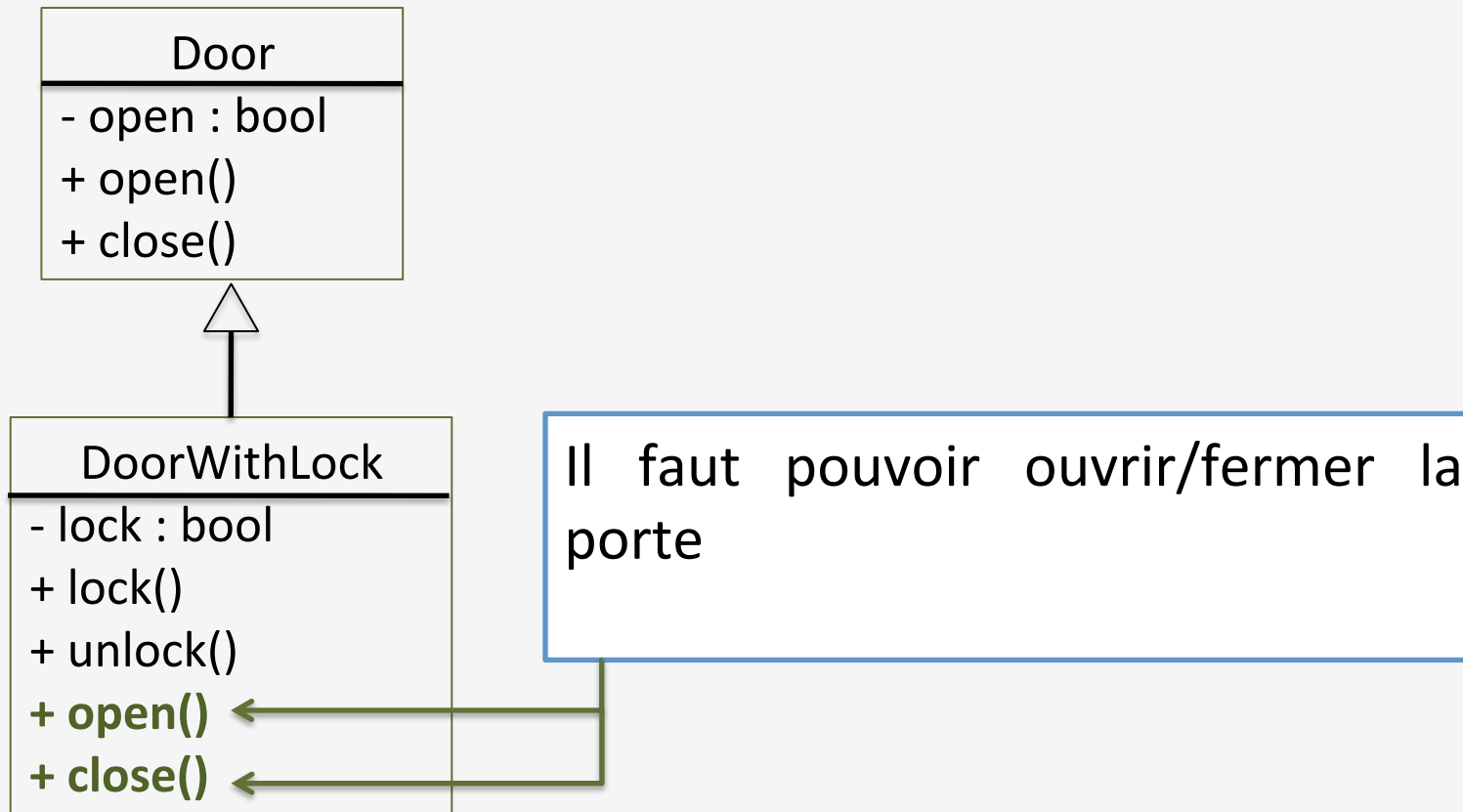
Une porte s'ouvre et se ferme « directement », alors qu'une porte avec verrou ne s'ouvre que si le verrou n'est pas enclenché.

PB : On peut ouvrir et fermer une porte verrouillée

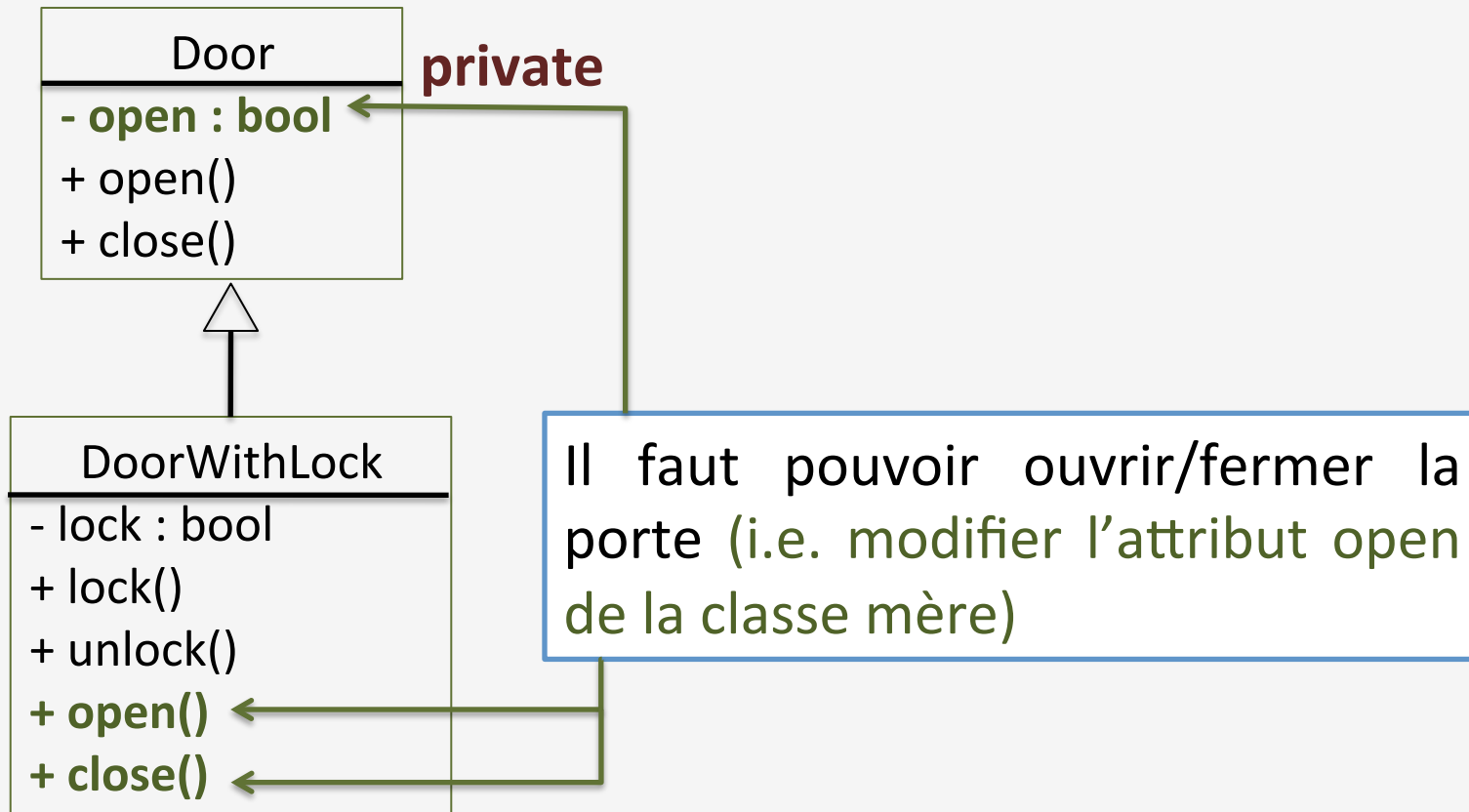
Utiliser une méthode de la classe mère c'est **super**



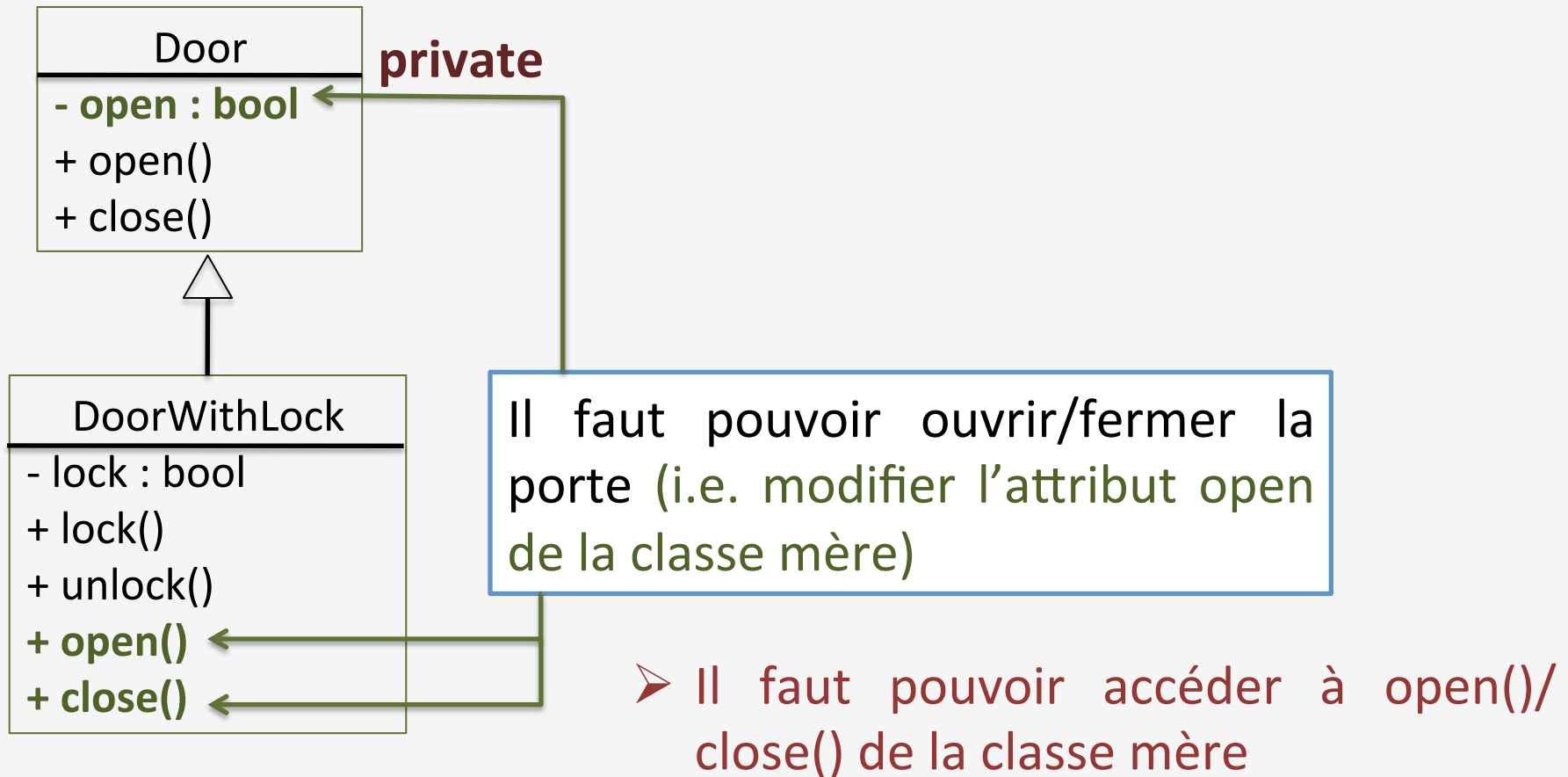
Utiliser une méthode de la classe mère c'est **super**



Utiliser une méthode de la classe mère c'est **super**



Utiliser une méthode de la classe mère c'est **super**



La redéfinition de méthode

exemple : redéfinition de la méthode open

```
public class DoorWithLock{  
...  
    @Override  
    public void open() {  
        if (!this.lock) {  
            super.open();  
        }  
    }  
}
```

La redéfinition de méthode

exemple : redéfinition de la méthode open

```
public class DoorWithLock{  
...  
    @Override  
    public void open() {  
        if (!this.lock) {  
            super.open();  
        }  
    }  
}
```



annotation

La redéfinition de méthode

exemple : redéfinition de la méthode open

```
public class DoorWithLock{
```

```
...
```

```
@Override
```

annotation

```
public void open() {
```

```
    if (!this.lock) {
```

```
        super.open();
```

Appel à la méthode
de la classe mère

```
    }
```

```
}
```

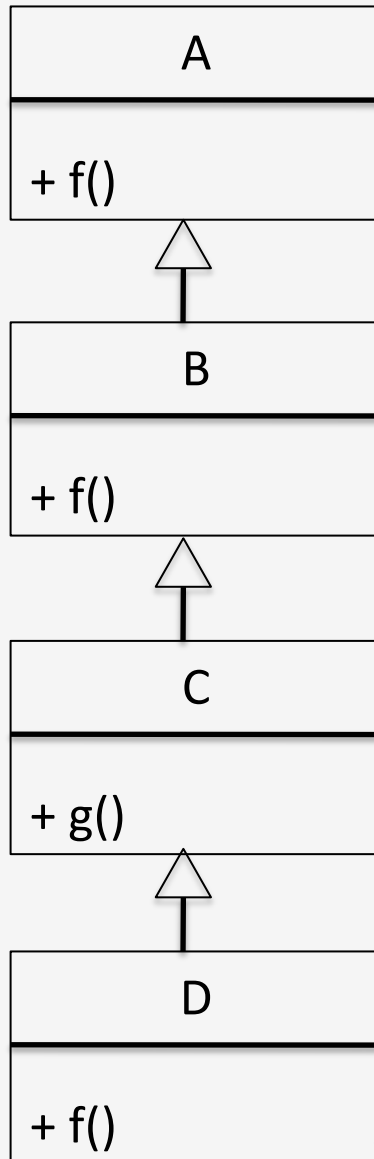
```
}
```

Utiliser une méthode de la classe mère c'est **super**

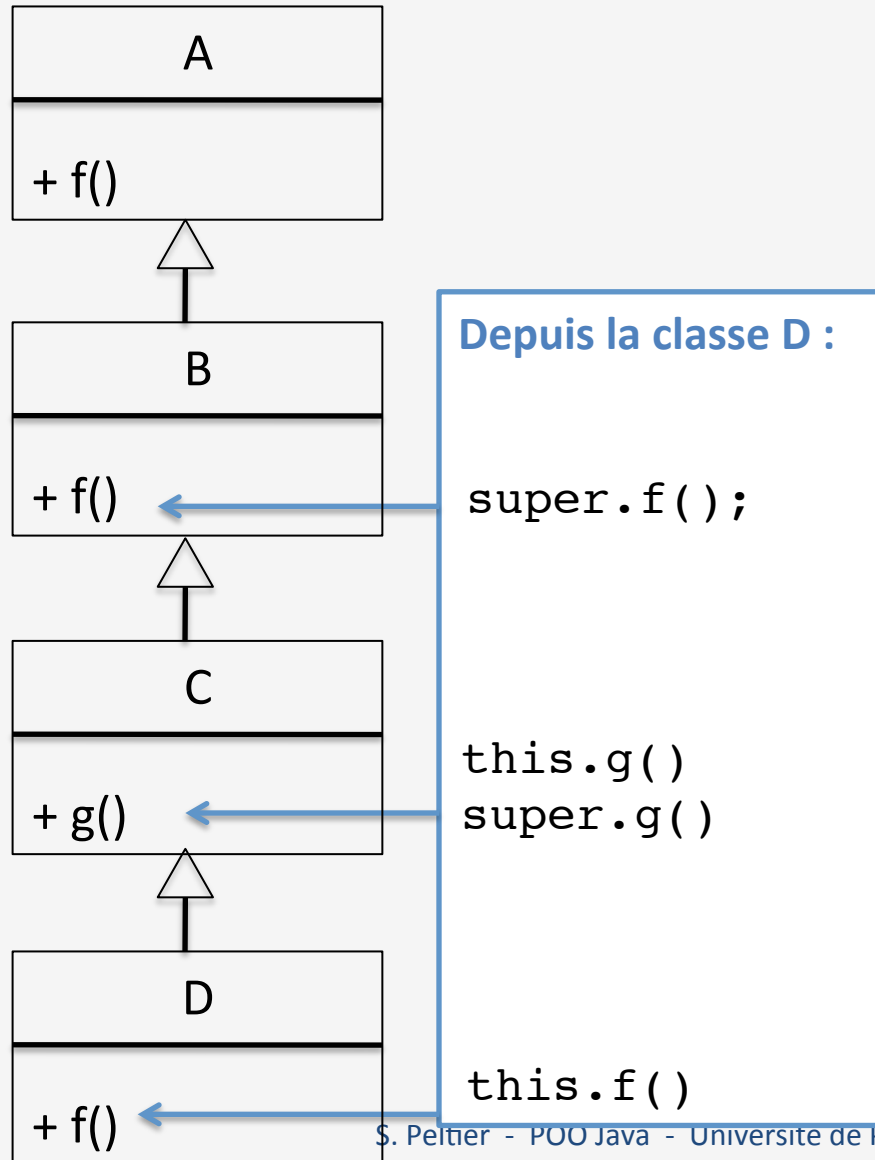
- Appel d'une méthode de la classe mère :
`super.method(...)`
- **Limitation** : en Java, on ne peut remonter qu'un niveau de définition.

~~`super.super.method();`~~

Héritage à plusieurs niveaux



Héritage à plusieurs niveaux



La surcharge de méthode

- Principe de la **surcharge** :

Possibilité de définir des méthodes possédant le **même nom** mais dont les paramètres d'entrée (et éventuellement les valeurs de retour) diffèrent.

Deux méthodes surchargées peuvent avoir des type de retour différents à condition qu'elles aient des paramètres différents.

La surcharge de méthode

- Par exemple, on ne peut pas avoir dans une même classe les deux méthodes suivantes :

```
public boolean method(int x) {  
    ...  
}
```

```
public int method(int y) {  
    ...  
}
```

Surcharge vs Redéfinition

- **Redéfinition :**
spécialisation d'une méthode existante
- **Surcharge :**
ajout d'une méthode

Contrôle de l'héritage

- Le mot clé **final** ne s'applique pas qu'aux données (constantes)

- Il permet aussi :

- d'empêcher la redéfinition dans une classe fille.

```
public final void aMethod() {  
    ...  
}
```

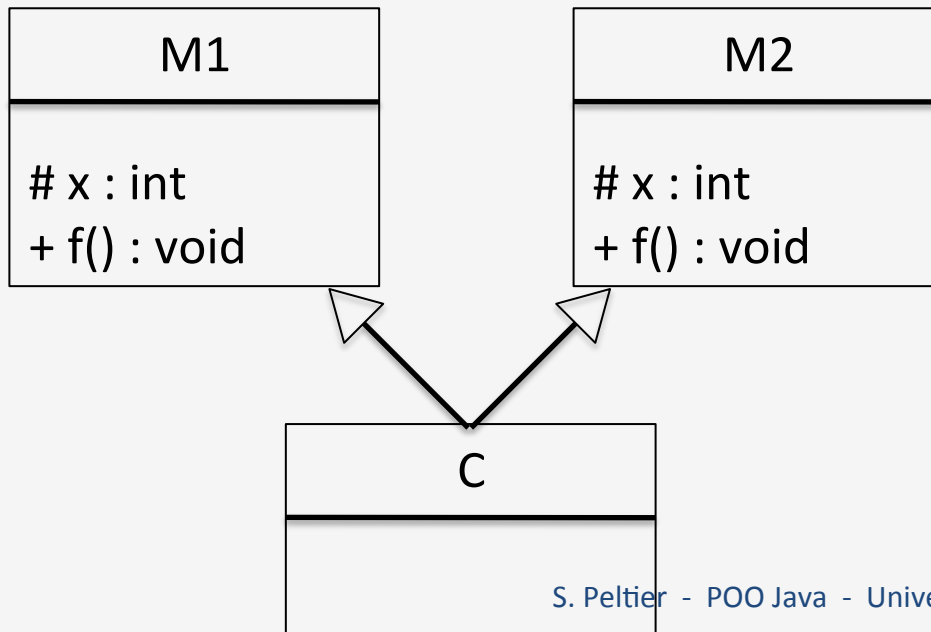
- d'empêcher la création de sous-classes

```
public final class MyClass {  
    ...  
}
```

Héritage multiple

Principe : une classe hérite de plus d'une classe.

**Différents problèmes
peuvent apparaître**



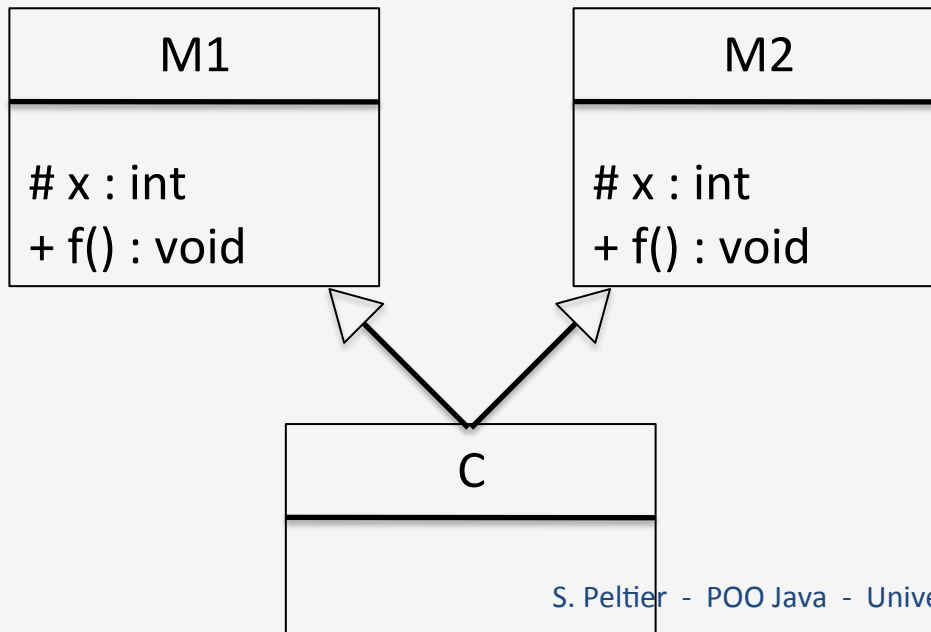
Héritage multiple

Principe : une classe hérite de plus d'une classe.

**Différents problèmes
peuvent apparaître**

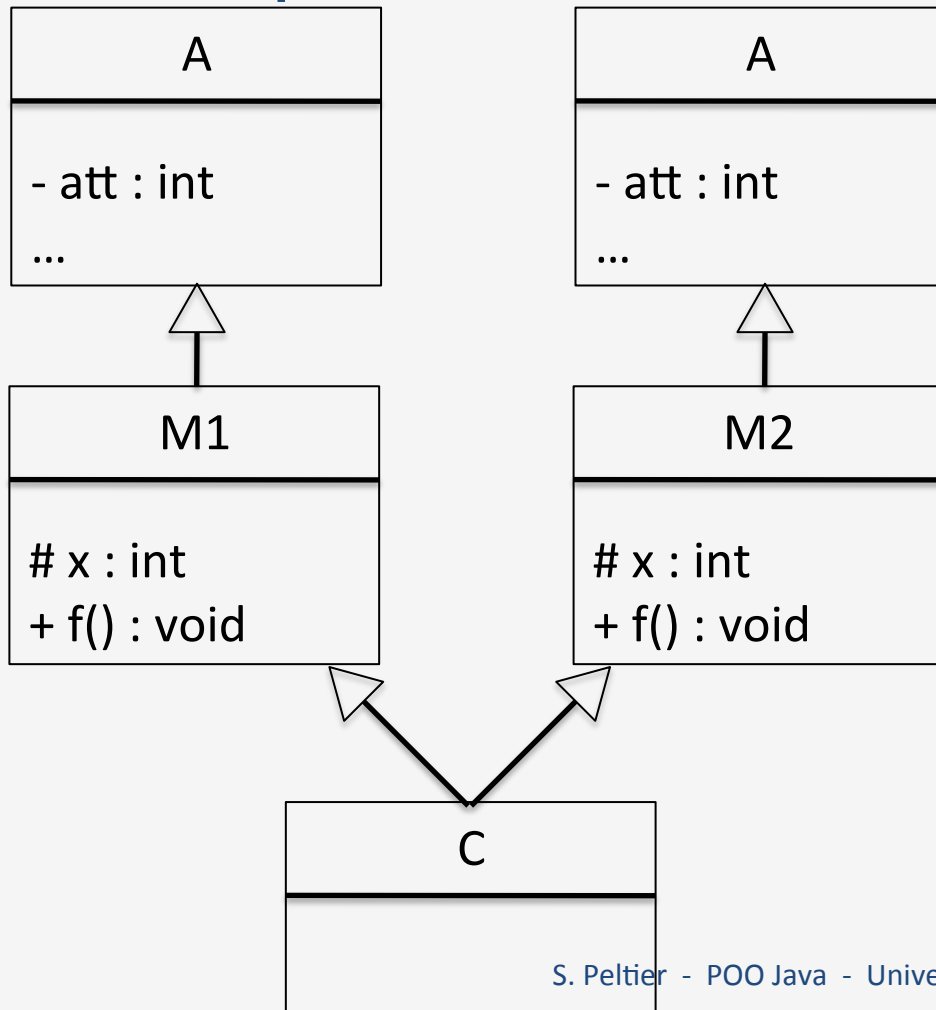
Dans C :

```
this.x ?  
this.f() ?
```



Héritage multiple

Principe : une classe hérite de plus d'une classe.

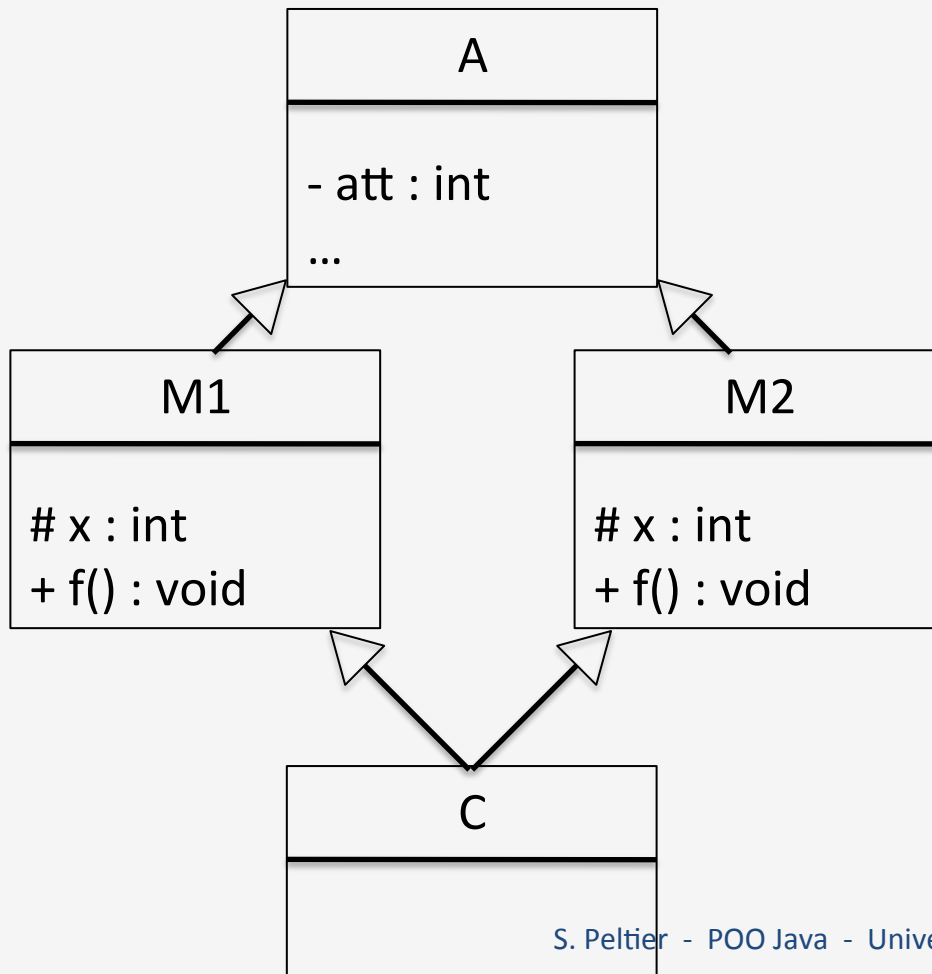


**Différents problèmes
peuvent apparaître**

Si M1 et M2 héritent de A
Des données peuvent
être dupliquées

Héritage multiple

Principe : une classe hérite de plus d'une classe.



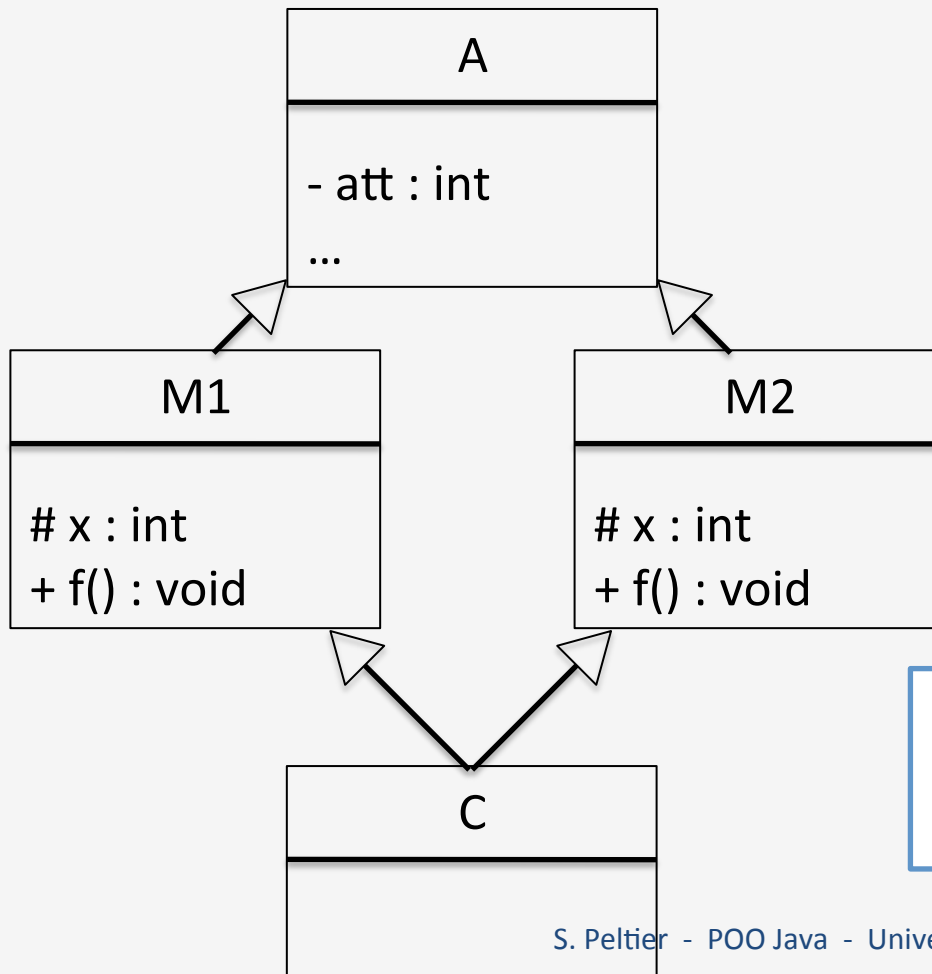
Différents problèmes peuvent apparaître

Héritage en diamant

Construction de la partie A ?

Héritage multiple

Principe : une classe hérite de plus d'une classe.



**Différents problèmes
peuvent apparaître**

Héritage en diamant

Construction de
la partie A ?

Réponse en Java :
Pas d'héritage multiple.

Polymorphisme

- **Le surclassement (upcasting) :**

Utiliser un objet de type sous-classe comme un objet de type surclasse.

```
SuperClass obj = new SubClass(...);
```

Polymorphisme

- **A la compilation :**
 - Lorsqu'un objet est surclassé, il est vu par le compilateur comme un objet du type défini lors de la déclaration.
 - Ses fonctionnalités sont donc restreintes à celles proposées par la surclasse.

Exemple

Polymorphisme

- **A l'exécution :**
 - Lorsqu'une méthode d'un objet surclassé est appelée, c'est la méthode de la classe effective de l'objet qui est invoquée.
 - La méthode à exécuter est déterminée à l'exécution et non à la compilation.

On parle de **liaison tardive**, ou **lien dynamique**

Exemple

Polymorphisme

Caractéristique essentielle d'un langage orienté objet avec l'abstraction, l'encapsulation et l'héritage.

- **Avantages :**
 - Pas besoin de différencier différents cas en fonction de la classe des objets (avec **instanceof**)
 - Maintenance du code plus aisée.
 - Code facilement extensible

Polymorphisme

- **Un exemple :**
 - Gestion d'un ensemble de formes géométrique.
On souhaite afficher le périmètre de chaque forme géométrique sans se soucier de chaque forme.

Polymorphisme

- **Le downcasting :**
 - **Intérêt :** Forcer un objet à libérer les fonctionnalités cachées par le surclassement
 - Conversion de type explicite (cast) déjà vu pour les types primitifs

```
SuperClass obj = new SubClass(...);  
( (SubClass)obj ).methodOfSubClass();
```


Polymorphisme

- **Le downcasting :**

Pour que le «cast »fonctionne, il faut qu'à l'exécution le type effectif de obj soit compatible avec le type SubClass.

Sinon, une exception `ClassCastException` est levée.

On peut tester la compatibilité par le mot clé **instanceof**

```
if(obj instanceof SubClass){  
    ((SubClass)obj).method(...);  
}
```

La classe Object

La classe **Object** :

- classe de plus haut niveau dans la hiérarchie,
- Seule classe qui ne possède pas de sur-classe

➤ Toute instance est donc de type Object.

Une classe qui n'a pas de clause extends hérite implicitement de la classe Object.

La classe Object

- Quelques méthodes de la classe Object :

Override

```
public boolean equals(Object o){  
    if (o instanceof MyClass){  
        MyClass obj = (MyClass)o ;  
        return ...  
    }  
    return false;  
}
```

La classe Object

- Quelques méthodes de la classe Object :

Override

```
public boolean equals(Object o){  
    if (o instanceof MyClass){  
        MyClass obj = (MyClass)o ;  
        return ...  
    }  
    return false;  
}
```

Note : redéfinir `public int hashCode()`

La classe Object

- Quelques méthodes de la classe Object :

```
protected void finalize()
```

```
public String toString()
```

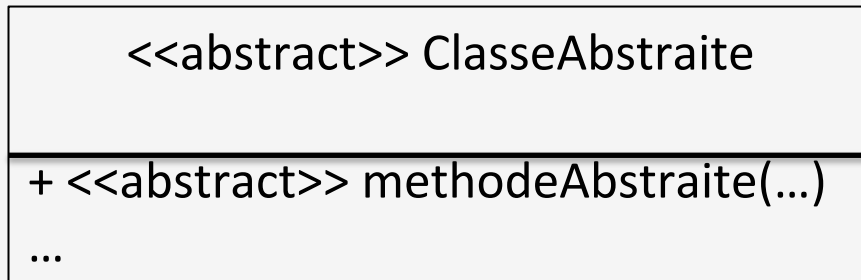
Les classes abstraites

- **Intérêt :**
 - Concevoir des applications avec un haut niveau d'abstraction
 - Il n'y a pas toujours un sens à donner une implantation par défaut d'une opération qui doit être redéfinie dans les sous-classes (par exemple, le périmètre d'une forme géométrique).
 - Force toutes les sous-classes concrètes à donner une implantation des méthodes abstraites

On peut écrire et compiler du code utilisant des méthodes abstraites.

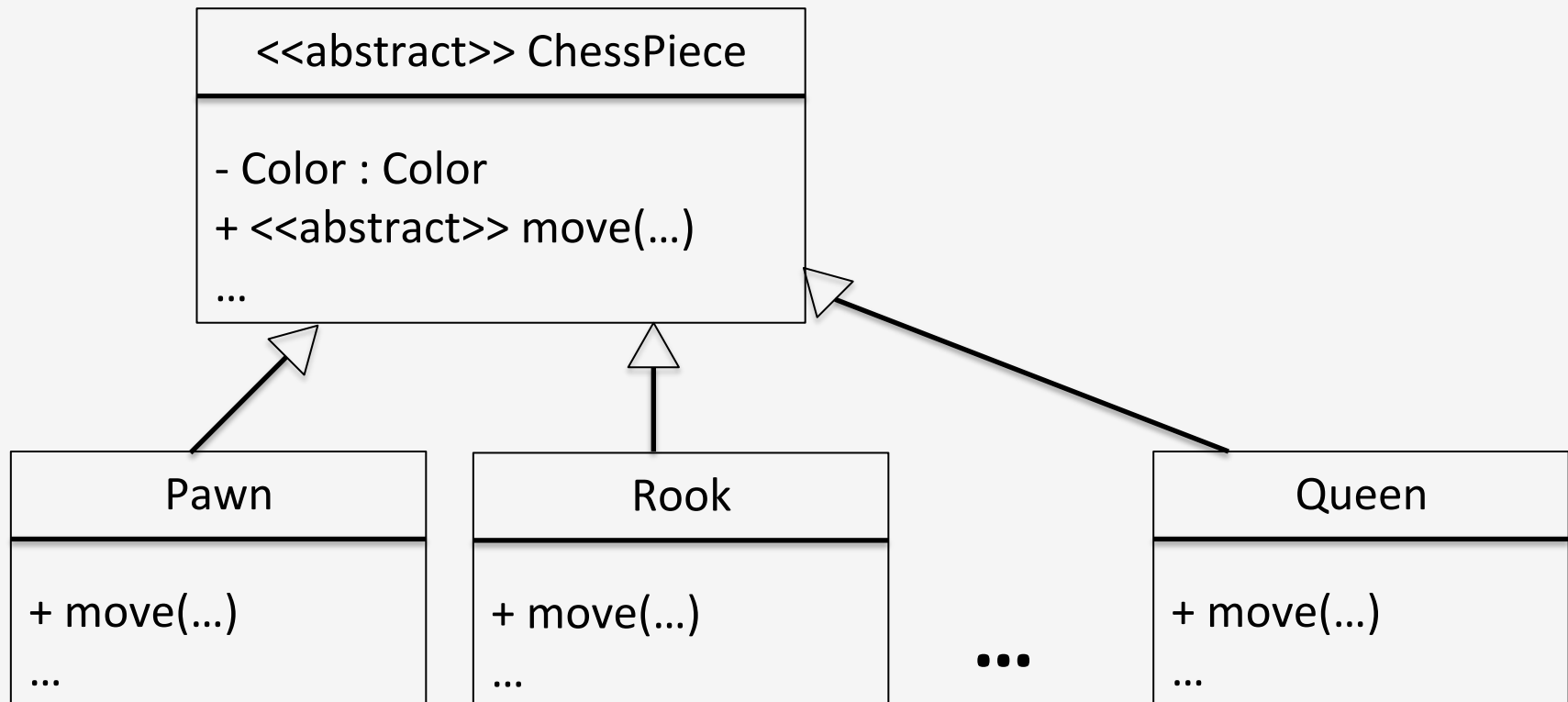
Les classes abstraites

- Notation UML : *italique* ou <<abstract>>



Les classes abstraites

exemple : pièces d'un jeu d'échecs.



Les classes abstraites

- Une classe abstraite **peut** avoir des méthodes abstraites (éventuellement aucune)
- Une classe contenant au moins une méthode abstraite **est abstraite**
- **On ne peut pas instancier** une classe abstraite.
- **Pour être une classe concrète**, une classe héritant d'une classe abstraite **doit implémenter toutes les méthodes abstraites** de sa classe mère.

Les classes abstraites

- Mot clé **abstract** :

- Pour la classe

```
public abstract class ChessPiece{  
    ...  
}
```

- Pour les méthodes

```
public abstract void move(...);
```

- **exemple** : ensemble de formes géométriques

Les classes abstraites

- Instanciation :

- **Directe :**

~~ChessPiece obj = new ChessPiece();~~ **ILLÉGAL**

- **Via une sous-classe concrète :**

ChessPiece obj = new Rook();

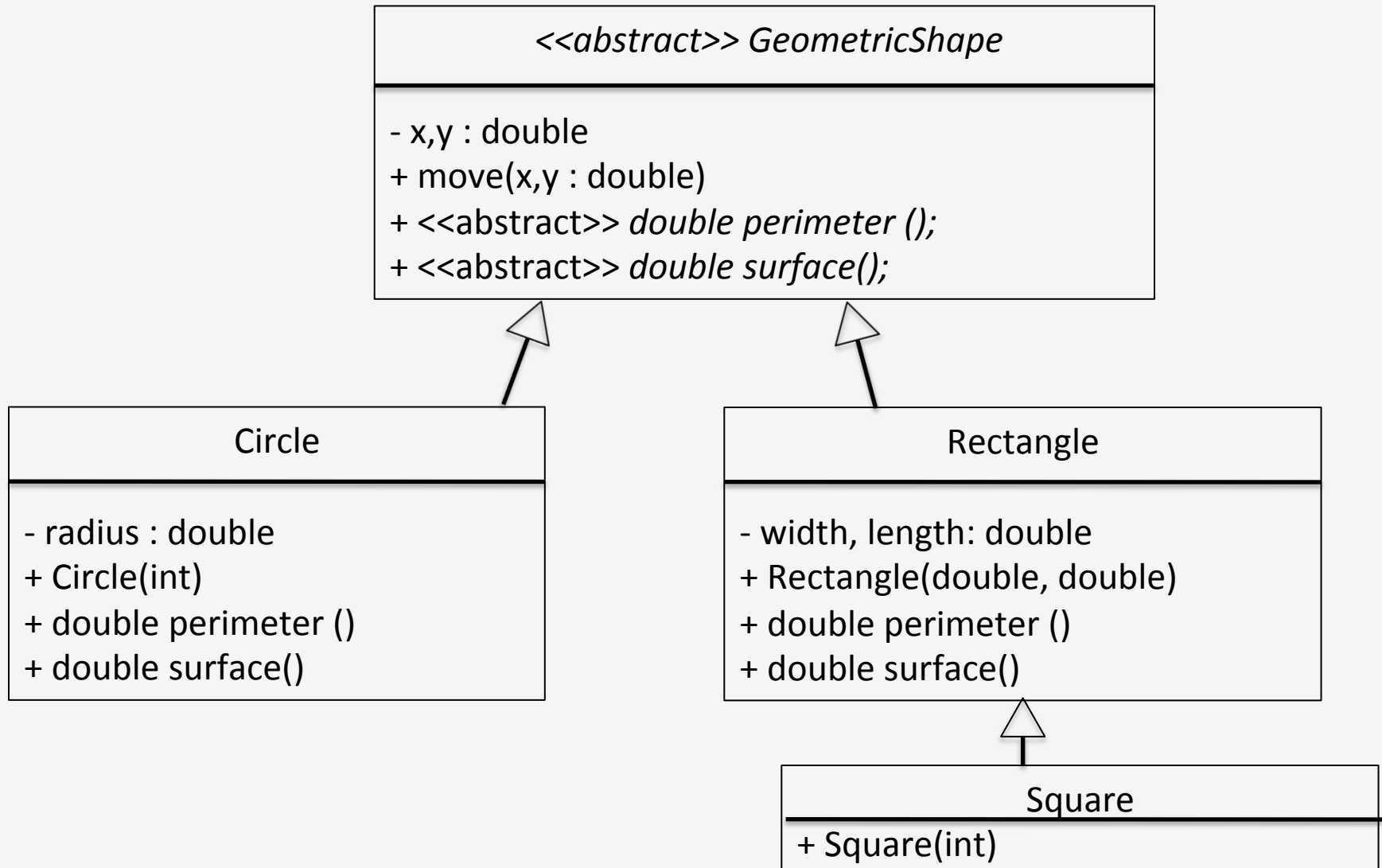
- **Via une classe anonyme :**

```
ChessPiece obj = new ChessPiece(){  
    @Override  
    public void move(...){  
        ...  
    }  
};
```

**Implantation des
méthodes abstraites**



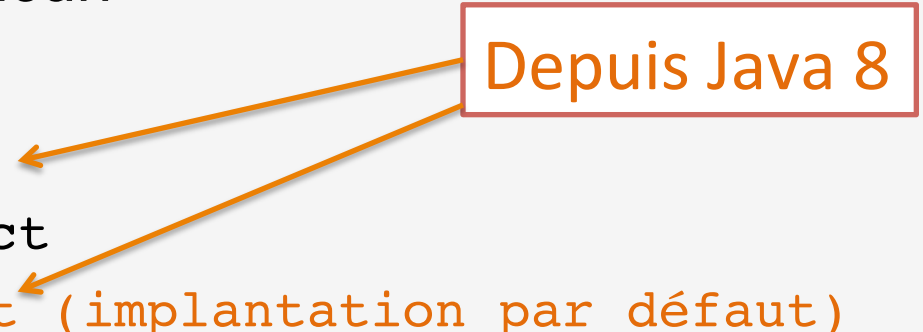
Les classes abstraites



Les interfaces

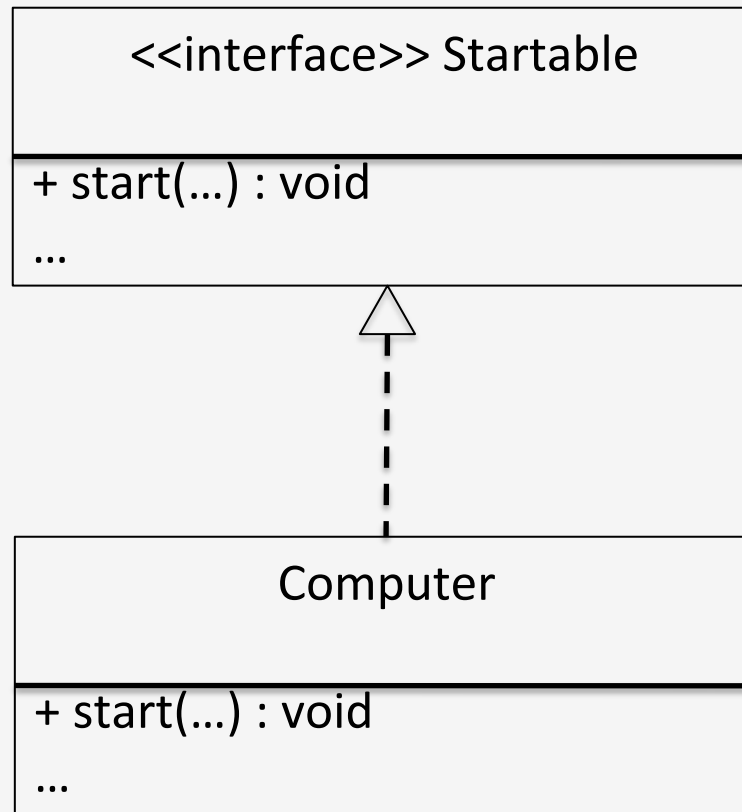
- À force d'abstraction, on arrive finalement à la notion d'interface :
 - **Attributs** : constantes publiques statiques.
 - **Constructeurs** : aucun
 - **Méthodes** :
 - `public static`
 - `public abstract`
 - `public default` (implantation par défaut)

Les interfaces

- À force d'abstraction, on arrive finalement à la notion d'interface :
 - Attributs : constantes publiques statiques.
 - Constructeurs : aucun
 - Méthodes :
 - `public static`
 - `public abstract`
 - `public default` (implantation par défaut)
- 

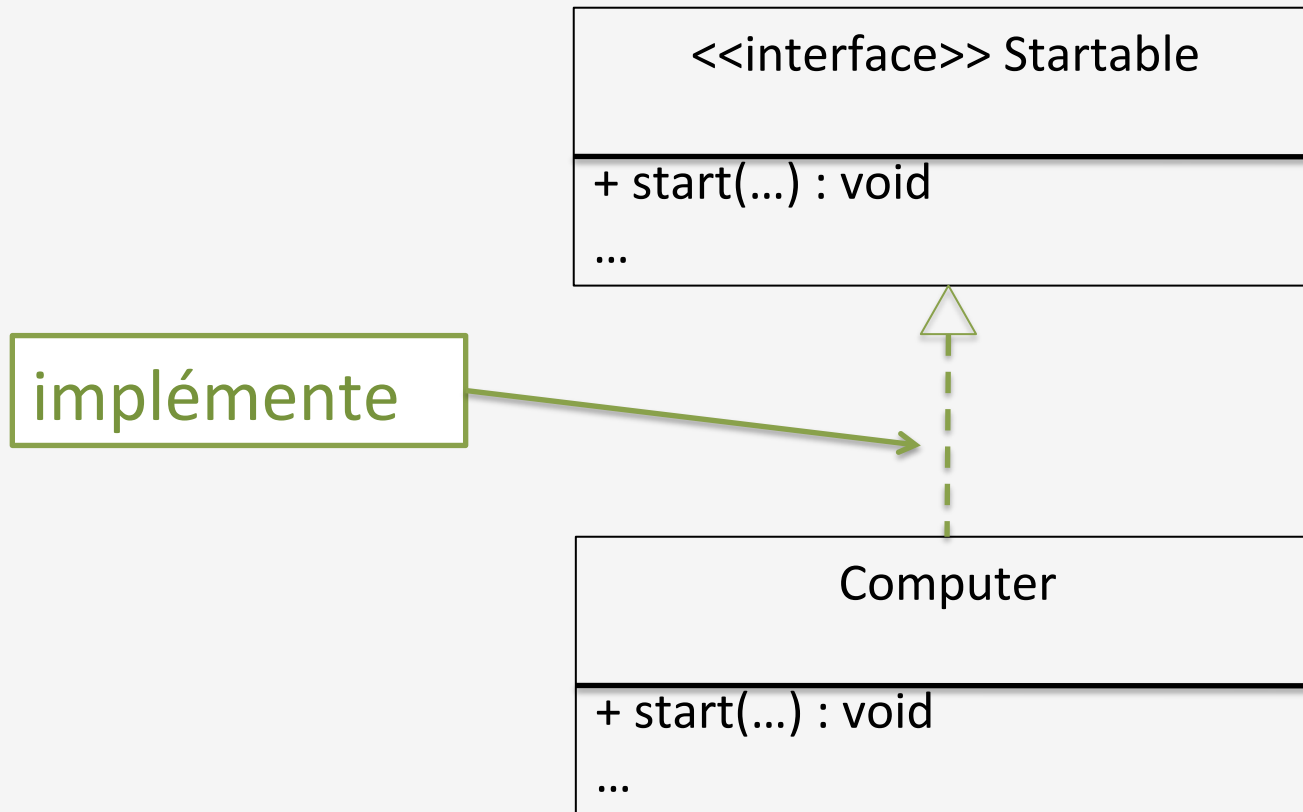
Les interfaces

- Notation UML :



Les interfaces

- Notation UML :



Les interfaces

Pour créer une interface : mot clé **interface**

```
public interface Recyclable {  
...  
}
```

- **exemple** : interfaces Startable Recyclable

Les interfaces

Une classe **implements** une interface

```
public class Car implements Startable{  
...  
}
```

Une classe peut implémenter plusieurs interfaces

```
public class Car implements Startable,  
                             Runnable, Recyclable{  
...  
}
```

Les interfaces

- Instanciation :

- **Directe :**

~~Startable obj = new Startable();~~ **ILLÉGAL**

- **Via une classe concrète qui l'implémente :**

Startable obj = new Computer();

- **Via une classe anonyme :**

Startable obj = new Startable(){

// implantation de toutes les méthodes
// (facultatif pour celles qui ont une
// implantation par défaut)

};

Sous-interfaces

Une interface peut hériter d'une ou plusieurs interfaces, on parle alors de sous-interfaces :

```
public interface MyInt extends Int1, Int2{  
...  
}
```

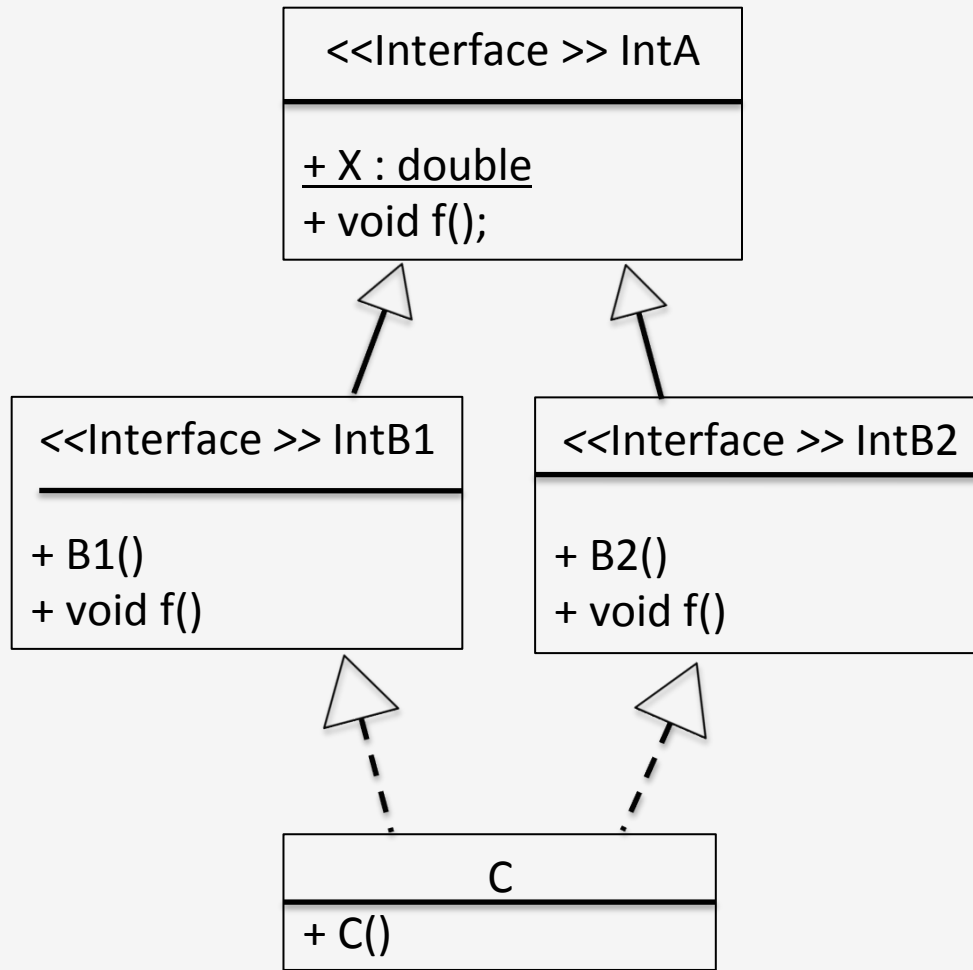
Les interfaces

- **Implantation par défaut (Java 8)**

```
default void toto() {  
    //default implantation  
}
```

Et les problèmes d'héritage en diamant alors ?

Les interfaces



Pas de problème lié à la construction de l'objet

```
IntA obj = new C();  
obj.f(); //?
```

Les interfaces

Lorsqu'une classe « hérite » d'une méthode depuis plusieurs endroits (classe et/ou interfaces) :

1. La définition dans une **classe** est prioritaire
2. S'il n'y en a pas, la définition « **default** » d'une **sous-interface** est prioritaire.
3. S'il y a ambiguïté, la méthode doit être **redéfinie dans la classe**.

Les interfaces

Accès explicites aux implémentations par défaut

```
public class C implements IntB1, IntB2 {  
    ...  
    public void myMethod() {  
        IntB1.super.f();  
        IntB2.super.f();  
    }  
    ...  
}
```


Interfaces fonctionnelles

(Java 8) Interface qui possède une et une seule méthode sans implémentation par défaut.

Objectif : simplification d'écriture avec l'utilisation de lambda expressions

Interfaces fonctionnelles

(Java 8) Interface qui possède une et une seule méthode sans implémentation par défaut.

Objectif : simplification d'écriture avec l'utilisation de lambda expressions

```
@FunctionalInterface
public interface Operation{
    ...

    public int eval(int x, int y);
}
```

Interfaces fonctionnelles

(Java 8) Interface qui possède une et une seule méthode sans implémentation par défaut.

Objectif : simplification d'écriture avec l'utilisation de lambda expressions

```
@FunctionalInterface ← Annotation
public interface Operation{
    ...

    public int eval(int x, int y);
}
```

Lambda expression

Lambda expression

(Paramètres d'entrée) -> {implémentation};

exemples (cf. java.util.function) :

Lambda expression

Sans Lambda :

```
IntPredicate even = new IntPredicate(){  
    @Override  
    public boolean test(int value){  
        return value%2==0;  
    }  
};
```

Lambda expression

Sans Lambda :

```
IntPredicate even = new IntPredicate(){  
    @Override  
    public boolean test(int value){  
        return value%2==0;  
    }  
};
```

Avec Lambda :

```
IntPredicate even = (x) -> x % 2 == 0;
```

Lambda expression

Sans Lambda :

```
IntPredicate even = new IntPredicate(){  
    @Override  
    public boolean test(int value){  
        return value%2==0;  
    }  
};
```

méthode abstraite test



Avec Lambda :

```
IntPredicate even = (x) -> x % 2 == 0;
```

Interfaces, classes, classes abstraites : bilan

- **Une classe (abstraite ou concrète) :**
 - Peut hériter d'une seule classe
 - Peut implémenter plusieurs interfaces
 - Peut être dérivée en classes abstraites ou concrète
- **Une classe concrète :**
 - Ne possède aucune méthode abstraite
- **Une classe abstraite :**
 - Peut posséder des méthodes abstraites (éventuellement aucune)

Interfaces, classes, classes abstraites : bilan

- **Une interface :**

- Peut posséder des **attributs public final static**
- Peut **hériter d'une ou plusieurs interfaces**
- Peut posséder des **méthodes public static** (Java 8)
- Peut posséder des **méthodes public abstract**
- Peut fournir **une implantation par défaut** de ses méthodes (Java 8)
- **fonctionnelle** : possède une et une seule méthode abstraite