Aspects avancés de la Spécialisation/Généralisation et de l'Héritage

Université de Montpellier - Faculté des Sciences
Licence informatique et Bi-Licence math-info 3e année
HAI401I

Plan

Plan

Modélisation conceptuelle

Classes, interfaces, énumérations

- Représentation des concepts d'un domaine
  - Intension : les attributs, les opérations, les méthodes
  - Extension : les instances/objets, les valeurs d'une énumération

Classification

- Hiérarchie de spécialisation/généralisation en UML
- Hiérarchie d'héritage en Java

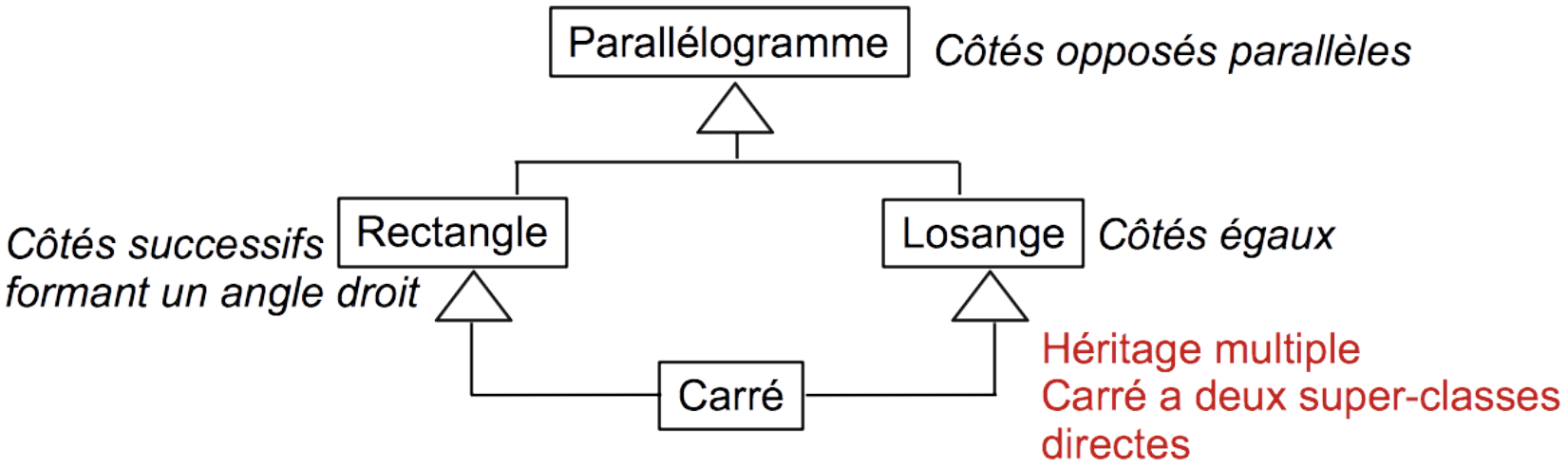
Héritage, instanciation, différents modèles

Modèles d'héritage

- Héritage simple : un concept (classe) n'a qu'un super-concept (super-classe) direct
  - La hiérarchie est un arbre
  - Modèle choisi par Java pour les classes
- Héritage multiple : un concept (classe ou interface) peut avoir plusieurs super-concepts (super-classe ou super-interface) directs
  - La hiérarchie est un graphe sans circuit
  - Modèle choisi par UML, C++ pour les classes, par Java pour les interfaces
  - Différents problèmes de conflits peuvent apparaître
  - Partage optimal (parfois la seule solution pour une factorisation maximale)

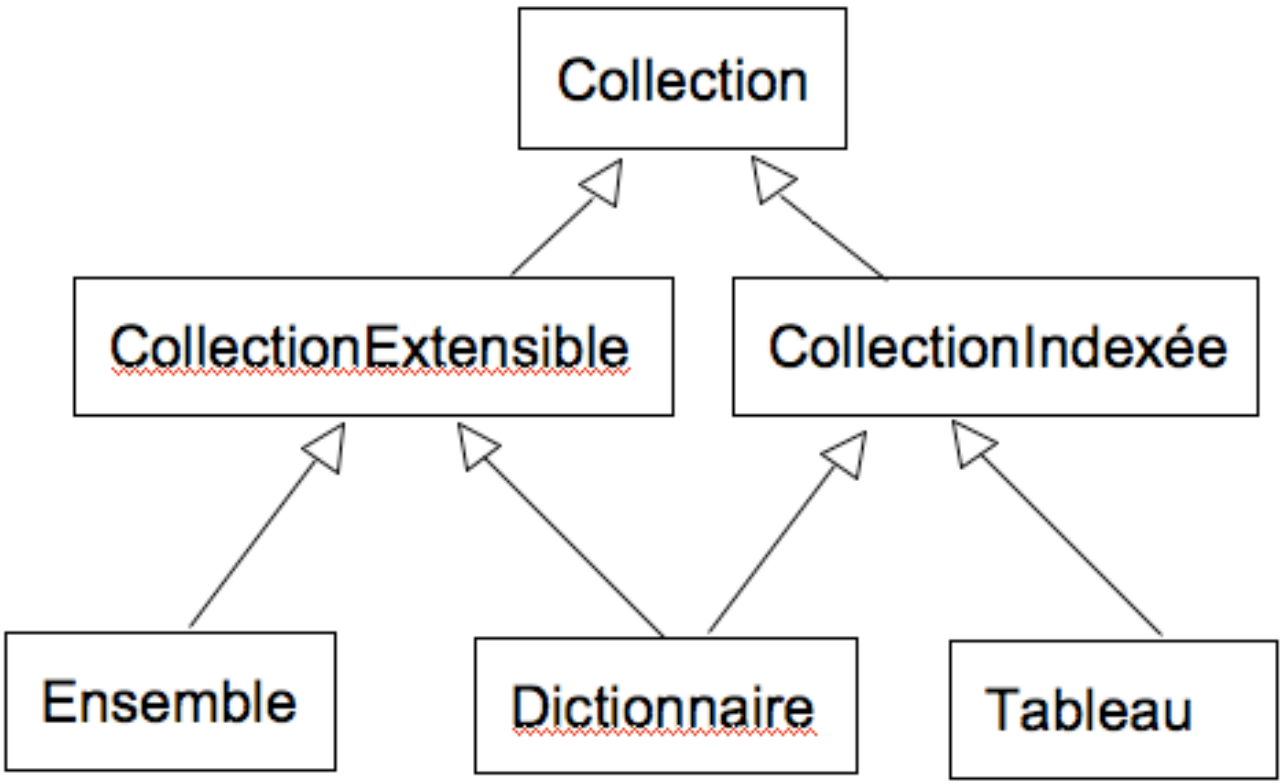
Multi-classification

Classification des polygones sur plusieurs propriétés : longueur des côtés, propriétés des côtés opposés, propriétés des côtés consécutifs.



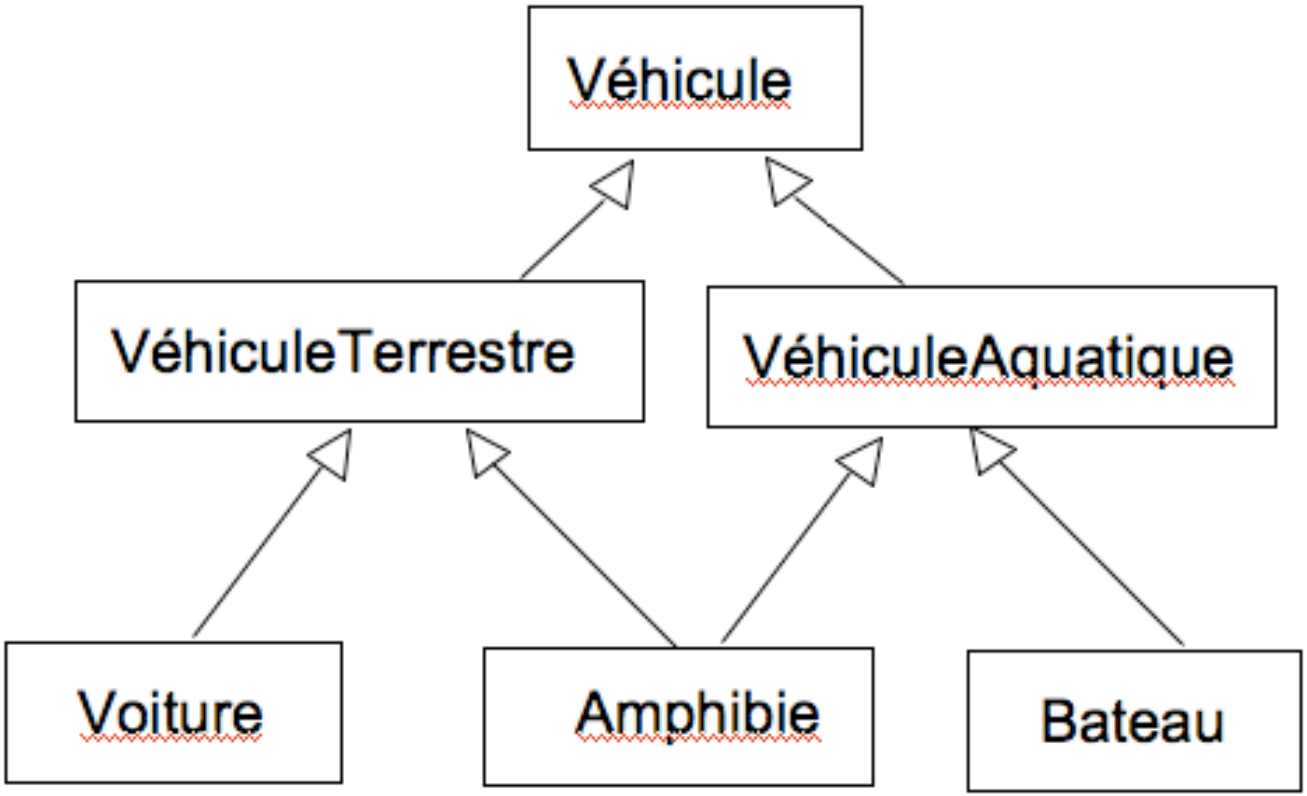
Multi-classification

Classification des collections sur plusieurs propriétés : extension possible, indexation des éléments (entiers, clefs)

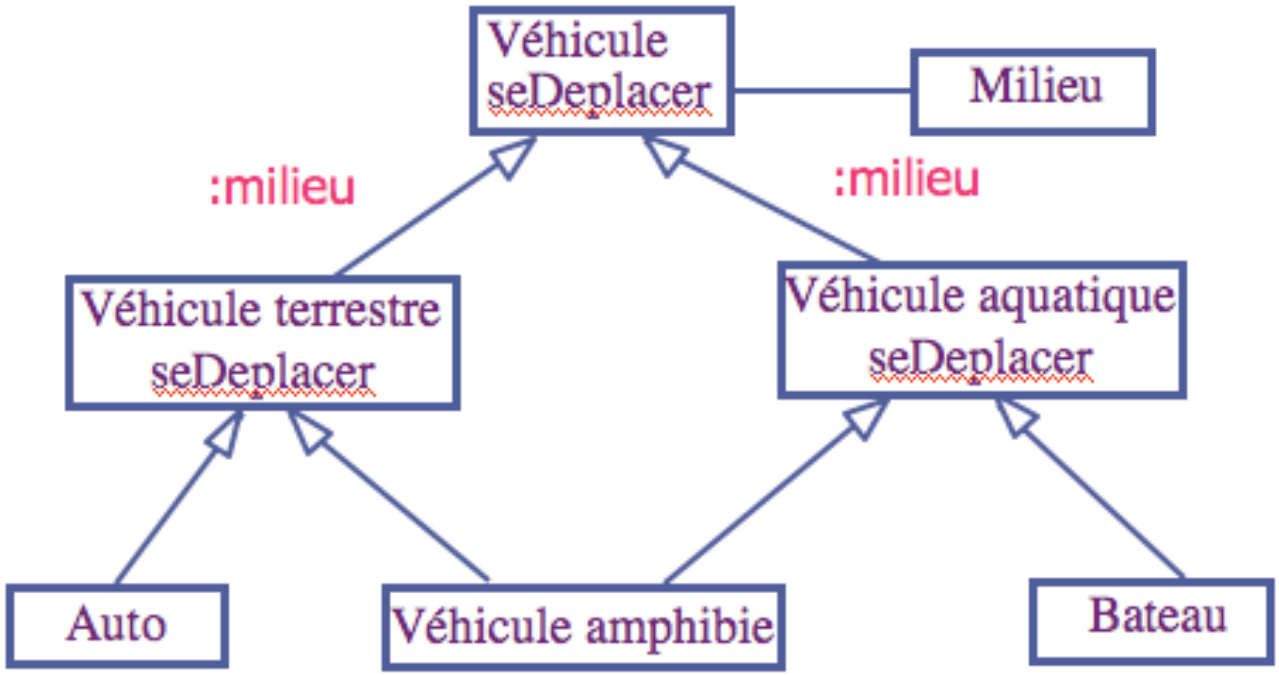


Multi-classification

Classification basée sur une propriété multi-valuée et dont les valeurs peuvent être cumulées par certains objets
Exemple : la caractéristique multi-valuée *milieu d'utilisation*

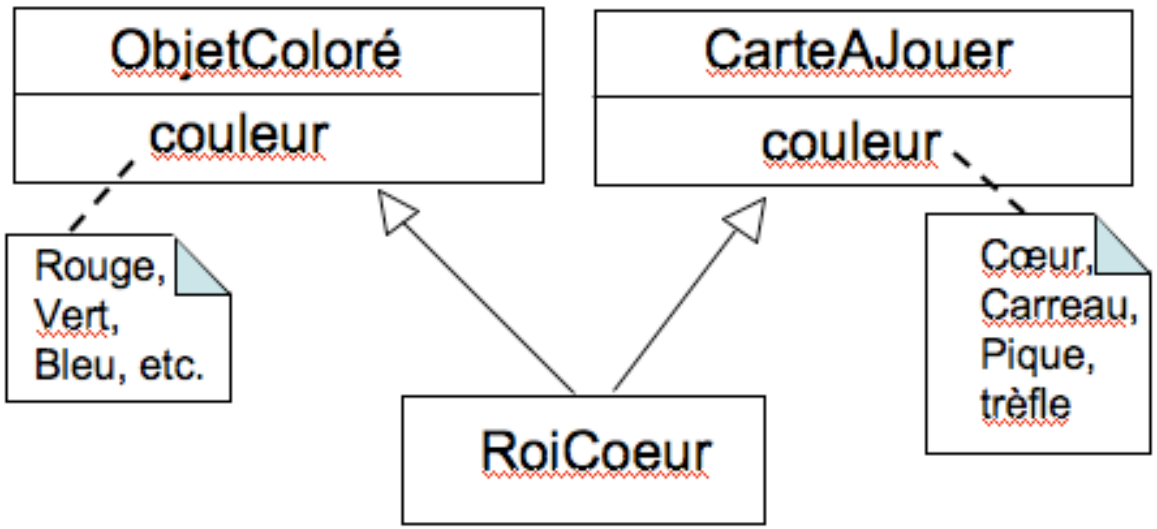


Conflits de valeurs



- Généralement, les propriétés qui produisent un conflit de valeur spécialisent une même propriété : dans notre exemple, il est logique que la classe des véhicules dispose également d'une méthode `seDeplacer`, même abstraite
- Les propriétés en conflit ont une origine commune

Conflit de nom



Généralement pas d'origine commune de la propriété

Conflits et leur résolution technique en Java

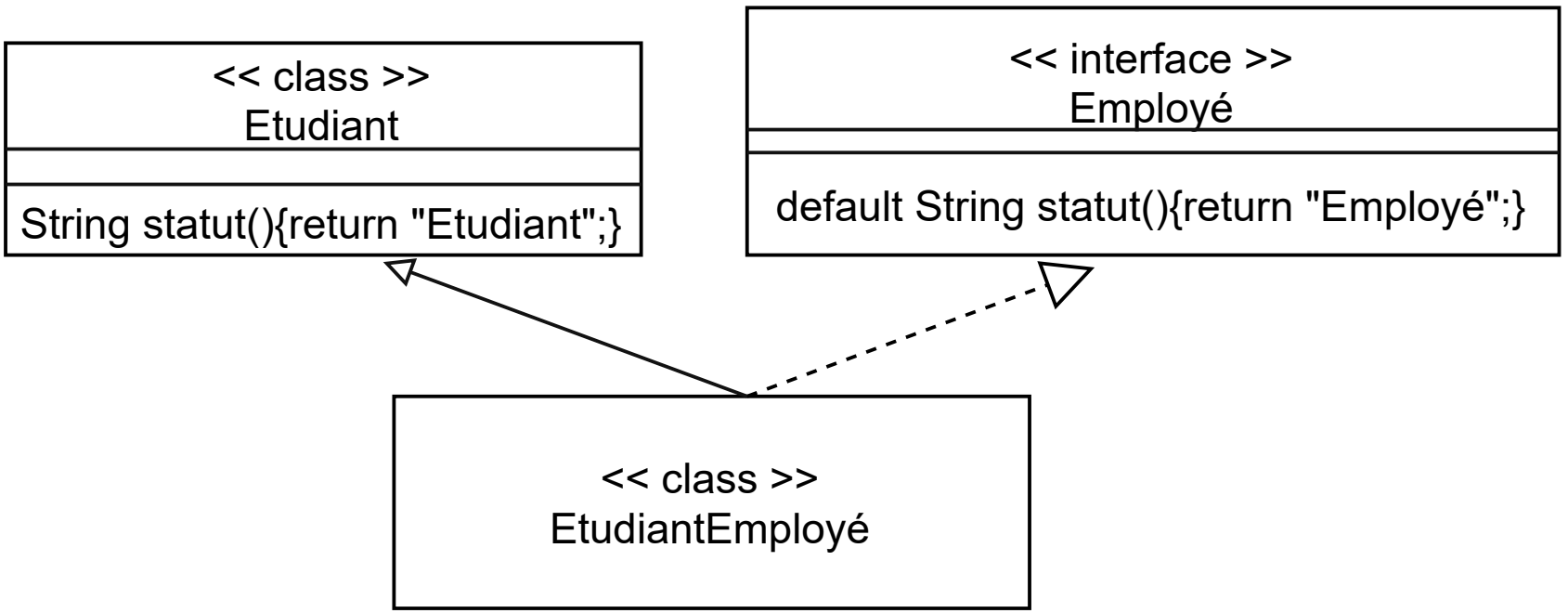
Depuis l'arrivée en Java des méthodes `default` dans les interfaces, des conflits peuvent se produire lorsqu'un type hérite de plusieurs méthodes avec la même signature. Principes de résolution :

- une méthode d'instance prend le dessus sur une méthode `default`
- une méthode déjà redéfinie est ignorée
- si un conflit persiste, on redéfinit la méthode concernée
- T.super.m() permet d'invoquer la méthode m d'un super-type direct T

Conflits et leur résolution technique en Java

Principe de résolution :

- une méthode d'instance prend le dessus sur une méthode `default`

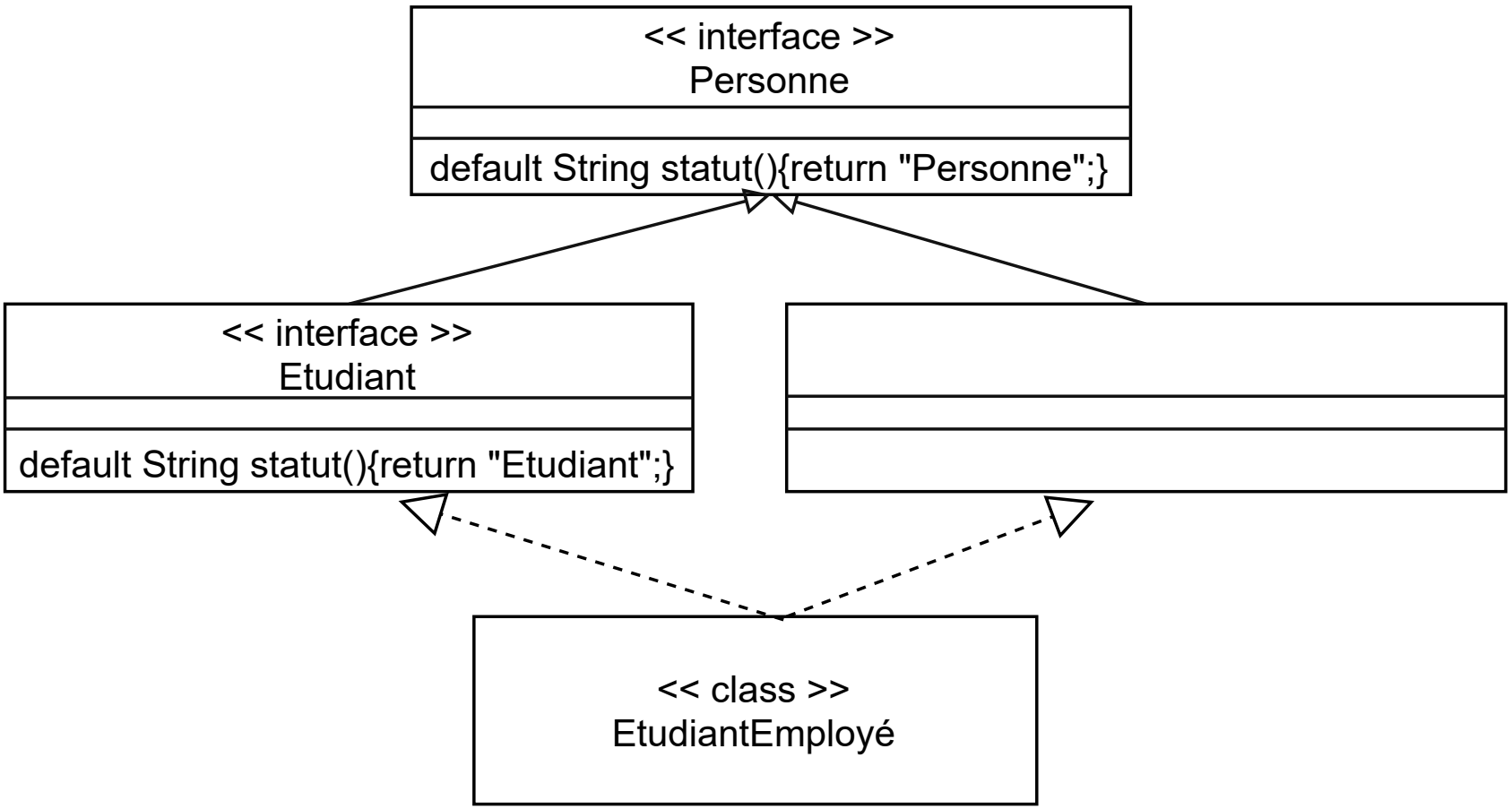



```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "Etudiant"
```

Conflits et leur résolution technique en Java

Principe de résolution :

- une méthode déjà redéfinie est ignorée

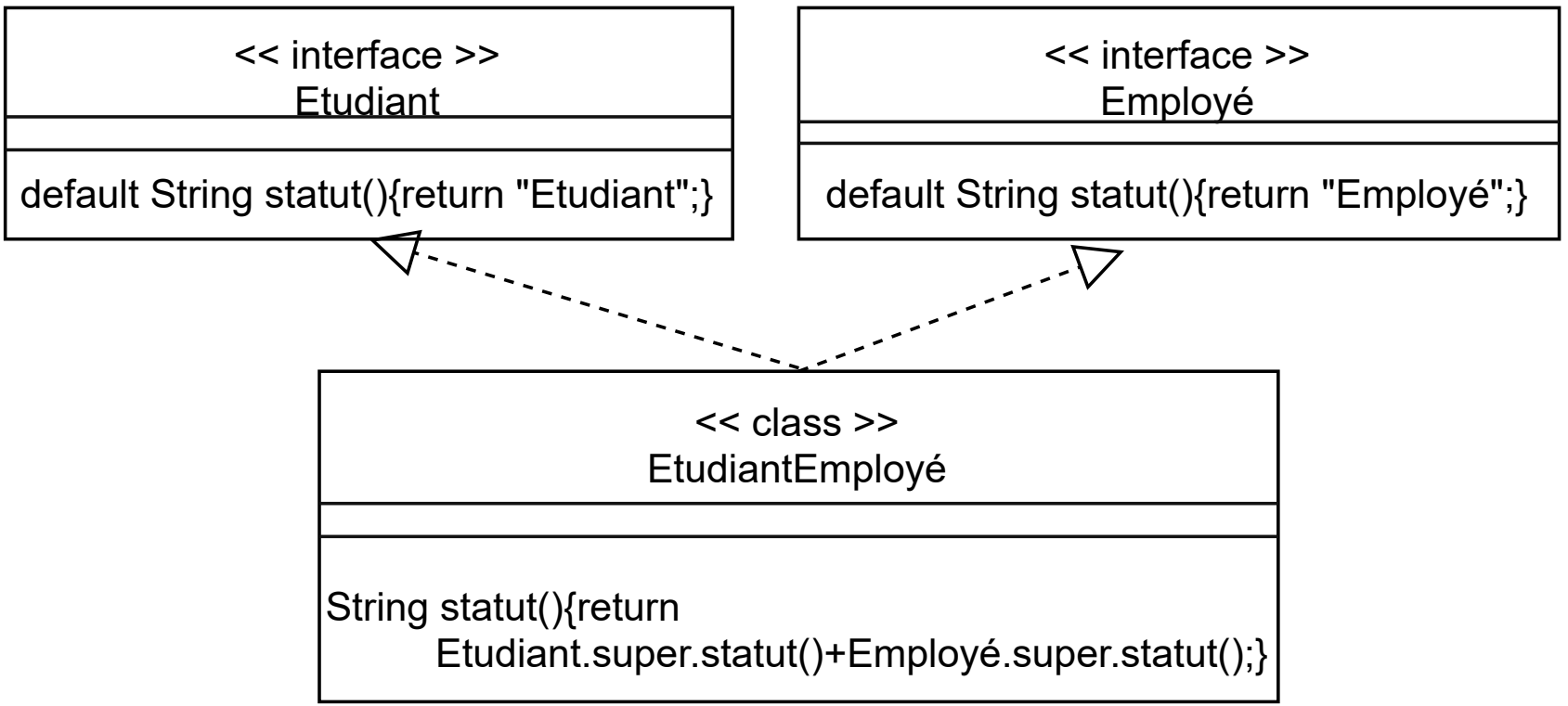



```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "Etudiant"
```

Conflits et leur résolution technique en Java

Principe de résolution :

- si un conflit persiste, on redéfinit la méthode concernée
- T.super.m() permet d'invoquer la méthode m d'un super-type direct T




```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "EtudiantEmployé"
```

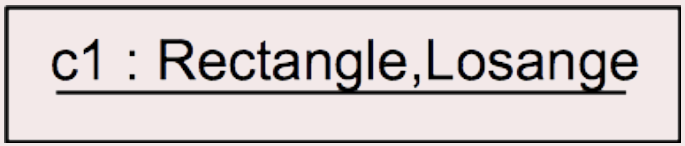
Héritage, instanciation, différents schémas

Modèles d'instanciation

- mono-instanciation, modèle de Java, une instance est rattachée directement à une seule classe

```
Carre c= new Carre(...);
```

- multi-instanciation, modèle d'UML, une instance est rattachée à plusieurs classes



Plan

Redéfinition versus surcharge statique

Observation à l'aide de la méthode `equals` d'une classe `Point`

```
public class Point{
    private int x,y;
    public Point() {this.x = 0; this.y = 0;}
    public Point(int x, int y) {this.x = x; this.y = y;}

    public int getX() {return x;}
    public void setX(int x) {this.x = x;}
    public int getY() {return y;}
    public void setY(int y) {this.y = y;}

    public boolean equals(Object p) {
    // redefinition de celle d'Object, avec test de type et ↩
        coercition
      if (p instanceof Point)
       return this.x==((Point)p).x && this.y==((Point)p).y;
      else return false;
     }
}
```

Cercle colorés - Redéfinition et surcharge statique

Observation à l'aide de la méthode `equals` d'une classe `Cercle` et de sa sous-classe `CercleColore`

```
public class Cercle {
    private Point centre; private double rayon;
    .....
    public boolean equals(Cercle c)
    {return this.getCentre().equals(c.getCentre()) &&
                this.getRayon() == c.getRayon();}}



public class CercleColore extends Cercle {
    private String couleur;
    .........
    public boolean equals(CercleColore c) {
        return super.equals(c) &&
                this.getCouleur().equals(c.getCouleur());}
}
....
public static void main(String[] args) {
  // ici on comparera des cercles et des cercles colores
  }
```

Redéfinition

Redéfinition de méthode

En Java une méthode m_r redéfinit une méthode m_i (la liaison dynamique sera appliquée) lorsque :

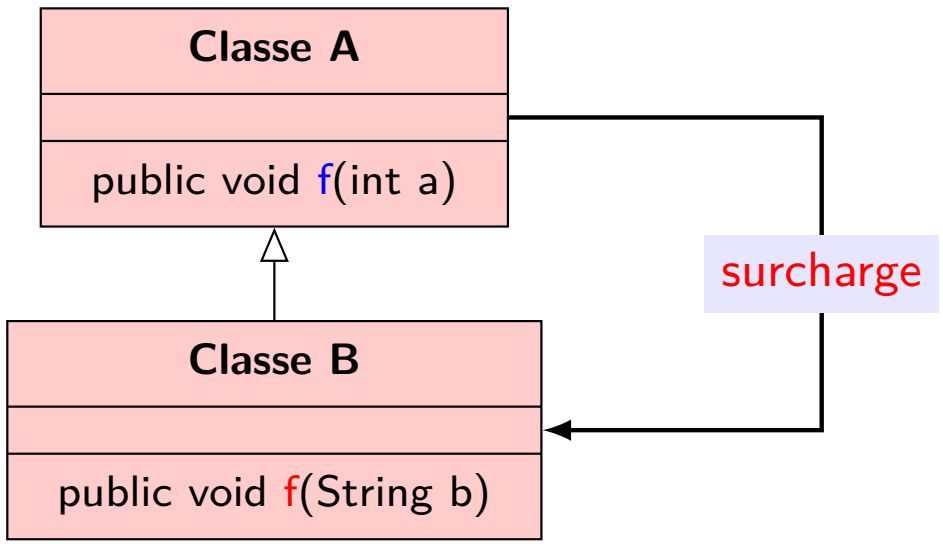
- m_r et m_i portent le même nom
- m_r a la même liste de types de paramètres que m_i (invariance de la liste des types de paramètres)
- le type de retour de m_r est le même ou une spécialisation de celui de m_i

Sinon il s'agit de surcharge statique.

De plus lors d'une redéfinition :

- On peut changer la visibilité pour l'augmenter (de protected à public par exemple).
- Des déclarations d'exceptions peuvent être retirées, ou spécialisées.

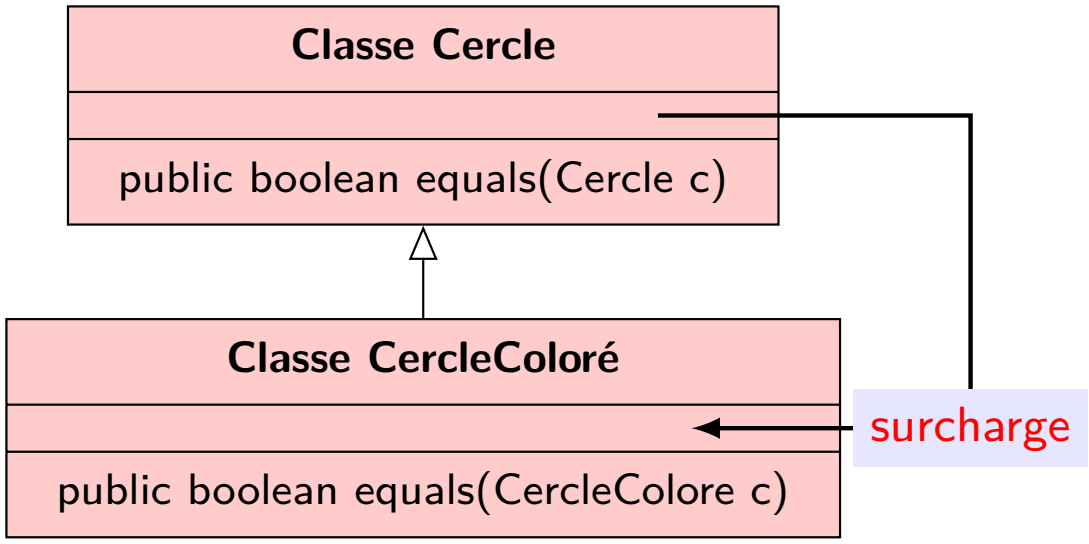
Surcharge statique



Une instance de B possède 2 méthodes f, c'est comme si les méthodes s'appelaient x et y !

```
1 B b = new B();
2 b.f(5);          // f de A
3 b.f("5");        // f de B
```

Retour sur les cercles colorés



- Le type de paramètre de la méthode de CercleColore spécialise celui de la méthode de Cercle = il est différent
- Liste de paramètres différents → surcharge statique
- On a donc deux méthodes distinctes `boolean equals(Cercle c)` et `boolean equals(CercleColore c)`
- Le choix de la signature de méthode à appeler est fait par le compilateur ; puis pour une signature choisie il peut y avoir liaison dynamique.

Retour sur les cercles colorés

Classe Cercle-> public boolean equals(Cercle c)
Classe CercleColoré extends Cercle-> public boolean equals(CercleColore c)

```
Cercle c1 = new CercleColore(new Point(1,2), 12, "bleu");
Cercle c2 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c1 egal c2 ?"+c1.equals(c2)); //--> TRUE
```

- c1 a pour type statique Cercle et pour type dynamique CercleColore.
- Compilateur : Une signature pour equals et admettant un Cercle (type statique de c2) en paramètre réel est cherchée dans Cercle : c'est equals(Cercle c).
- Interprète : Il regarde si equals(Cercle c) est redéfini dans le type dynamique de c1. Ce n'est pas le cas (la méthode equals(CercleColore c) n'étant pas une redéfinition.
- equals(Cercle c) est exécutée et ne compare que centre et rayon, sur ce plan c1 et c2 sont identiques.

Retour sur les cercles colorés

Classe Cercle-> public boolean equals(Cercle c)
Classe CercleColoré extends Cercle-> public boolean equals(CercleColore c)

```
CercleColore c3 = new CercleColore(new Point(1,2), 12, "bleu");
CercleColore c4 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c3 egal 4 ?"+c3.equals(c4));// --> FALSE
```

- c3 a pour type statique et pour type dynamique `CercleColore`.
- Compilateur : Une signature pour `equals` et admettant un `CercleColore` en paramètre réel est cherchée dans `CercleColore` : c'est `equals(CercleColore c)`.
- Interprète : Il cherche `equals(CercleColore c)` à partir de `CercleColore` (et la trouve là).
- `equals(CercleColore c)` est exécutée et compare centre, rayon, et couleur ; sur la couleur, c3 et c4 sont différents.

Retour sur les cercles colorés

Classe Cercle-> public boolean equals(Cercle c)
Classe CercleColoré extends Cercle-> public boolean equals(CercleColore c)

```
Cercle c1 = new CercleColore(new Point(1,2), 12, "bleu");
CercleColore c4 = new CercleColore(new Point(1,2), 12, "vert");
System.out.println("c4 egal c1 ?"+c4.equals(c1));// --> TRUE
```

- c4 a pour type statique et dynamique `CercleColore`.
- c1 a pour type statique `Cercle` et pour type dynamique `CercleColore`.
- Compilateur : Une signature pour `equals` et admettant un `Cercle` en paramètre réel est cherchée dans `CercleColore` : c'est `equals(Cercle c)` (la version héritée !), car `equals(CercleColore c)` ne peut accepter un cercle.
- Interprète : Il regarde si `equals(Cercle c)` est redéfini dans le type dynamique de c4. Ce n'est pas le cas (la méthode `equals(CercleColore c)` n'étant pas une redéfinition.
- `equals(Cercle c)` est exécutée et ne compare que centre et rayon, sur ce plan c1 et c4 sont identiques.

Plan

Coercition et tests de type

Cherchons à redéfinir correctement la méthode `boolean equals(Cercle c)` dans `CercleColore`.
Première approche :

```
1 public boolean equals(Cercle c) {
2    return super.equals(c) &&
3              this.getCouleur().equals(c.getCouleur());
4 }
```

Erreur de compilation !
La méthode `getCouleur()` n'existe pas dans `Cercle`
En effet le type statique de `c` est `Cercle`
Solution avec de la coercition (ou typecast) et un test de type

```
1 public boolean equals(Cercle c) {
2    if (c instanceof CercleColore)
3       return super.equals(c) &&
4          this.getCouleur().equals(((CercleColore)c).getCouleur());
5    else return false;
6 }
```

Coercition et tests de type

Rappel de la solution avec de la coercition (ou typecast) et un test de type

```
1 public boolean equals(Cercle c) {
2     if (c instanceof CercleColore)
3         return super.equals(c) &&
4             this.getCouleur().equals(((CercleColore)c).getCouleur());
5     else return false;
6 }
```

L'exemple donné (redéfinition d'une "biméthode") est un des cas où l'on s'autorise la coercition et les tests de type mais retenons :

Règle

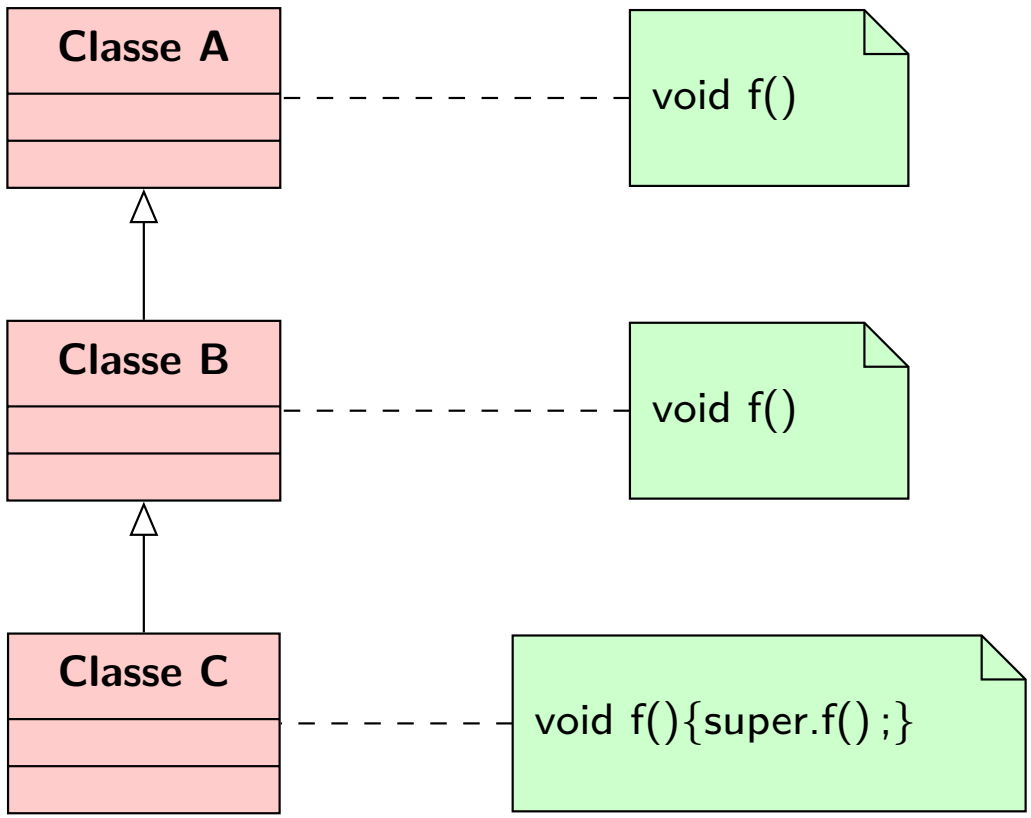
En général, on utilise le moins possible la coercition et les tests de type. Ils révèlent souvent une mauvaise conception.

Retenons aussi :

Ecriture de `equals`

Quand on écrit une méthode `equals`, on redéfinit celle de la classe `Object` qui a pour signature `boolean equals(Object o)`.
Exercice : le faire pour les classes `Cercle` et `CercleColore`.

La coercition peut-elle permettre d'appeler la méthode *masquée* ?



Aucun moyen d'appeler directement f() de A

~~((A)super).f();~~

Plan

Spécialisation naturelle versus substituabilité

Spécialisation naturelle

Un concept S spécialise un concept T lorsque l'ensemble des instances de S est inclus dans l'ensemble des instances de T.
C'est le point de vue notamment de la modélisation en UML.
Exemple : les aliments frais sont des aliments ; les carrés sont des rectangles

Principe de substitution de Liskov

Lorsqu'un type S est sous-type d'un type T, tout objet de type S peut remplacer un objet de type T sans que le programme n'ait de comportement imprévu ou ne perde de propriétés souhaitées.
C'est le point de vue des langages à typage statique, il sert à anticiper au maximum les erreurs dans les programmes.
Exemple : les aliments frais ne sont pas des aliments quelconques dans ce point de vue car on ne peut pas les ranger dans tous les lieux où on range des aliments quelconques (il leur faut un lieu réfrigéré).

Principe de substitution de Liskov

Il impose :

- des restrictions sur les signatures lors des redéfinitions
- des conditions comportementales (apparentes notamment sur les contrats)

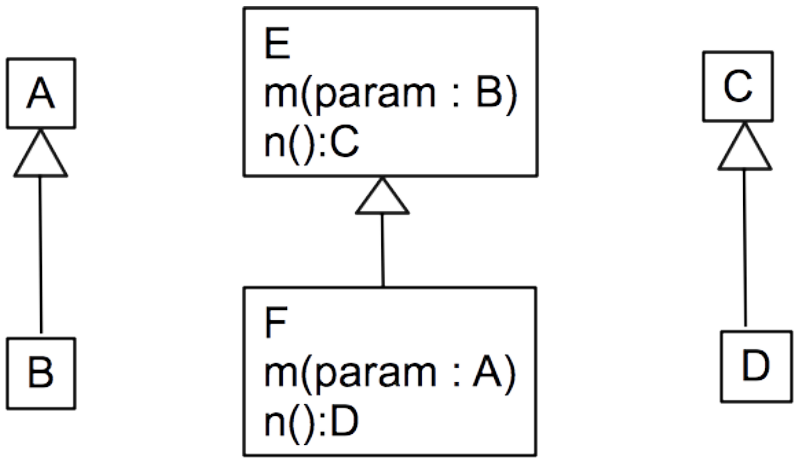
Il permet la règle :

- Une référence de type A peut repérer une instance de A ou de toute classe dérivée de A.

Substituabilité - Restriction sur les signatures

La substituabilité revient à demander moins et donner plus au site d'appel :

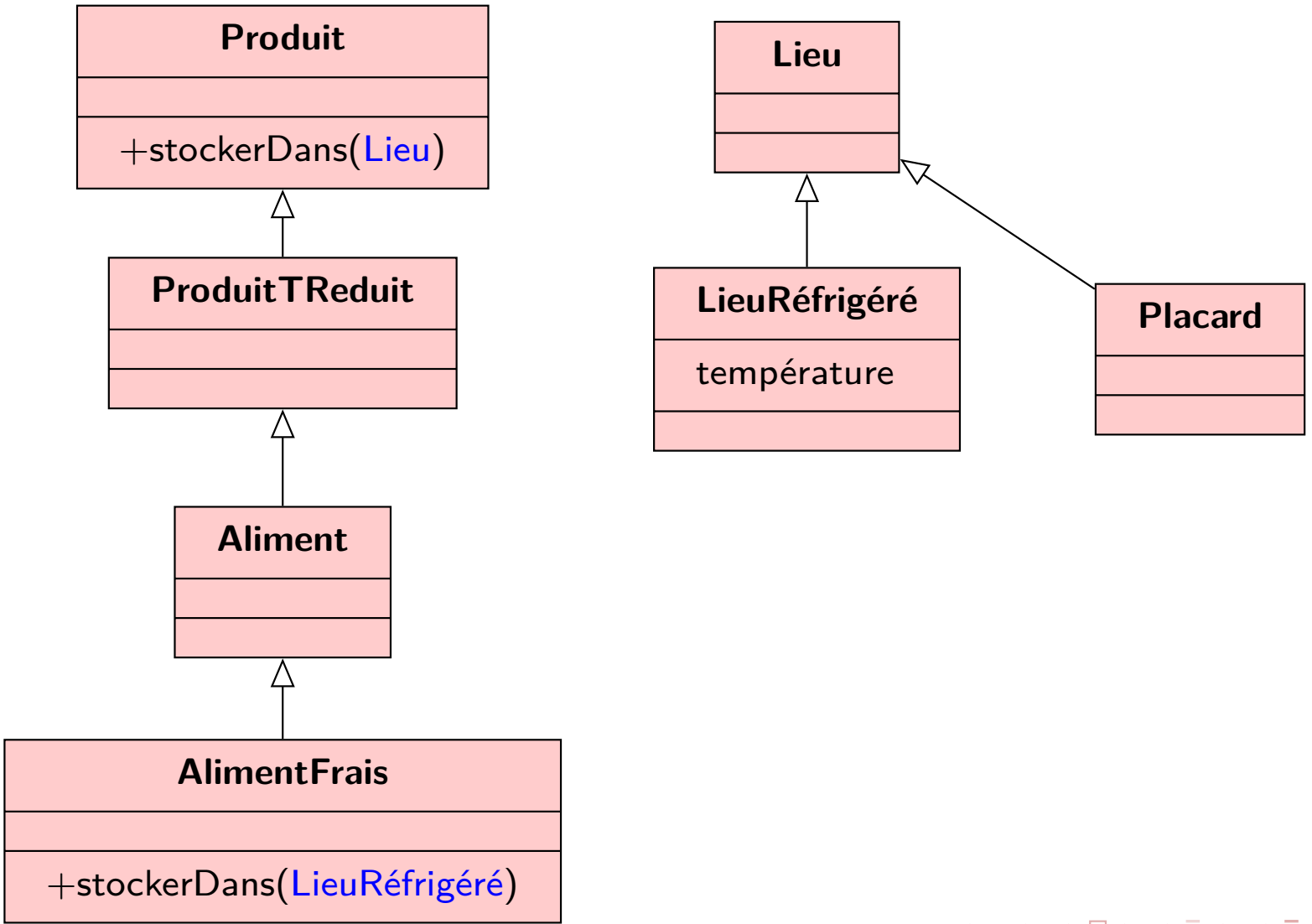
- contravariance des paramètres
  - les paramètres varient en sens inverse de la redéfinition
- covariance du type de retour
  - le type de retour varie dans le même sens que la redéfinition
- affaiblissement des pré-conditions
- renforcement des post-conditions

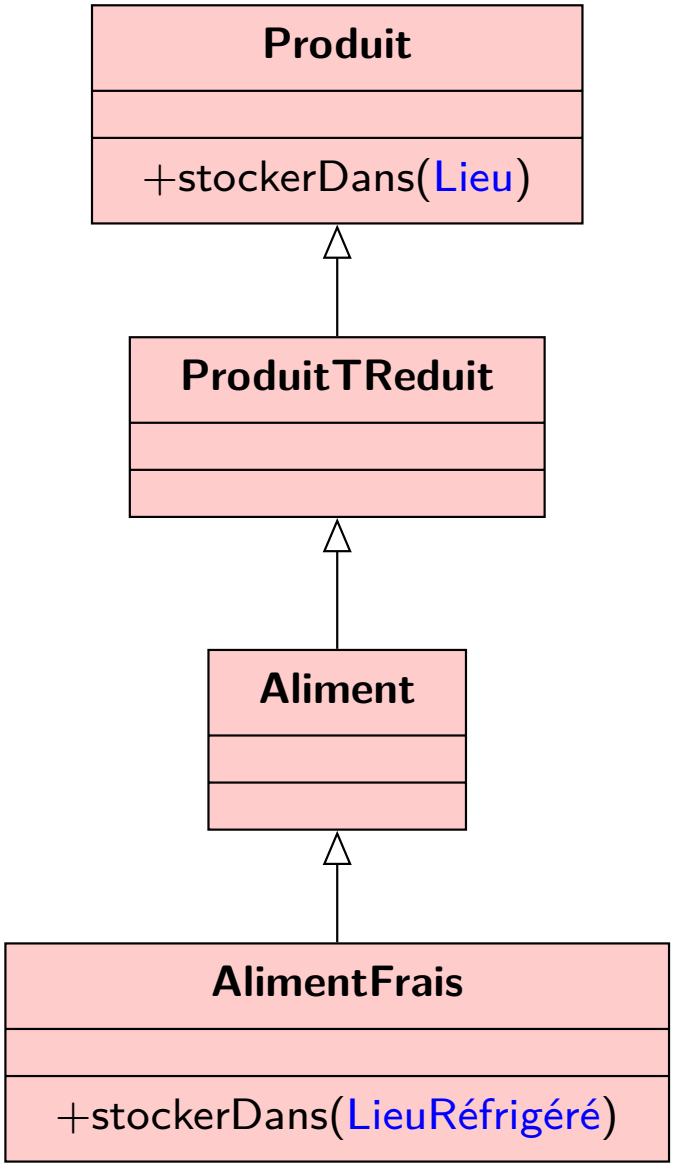


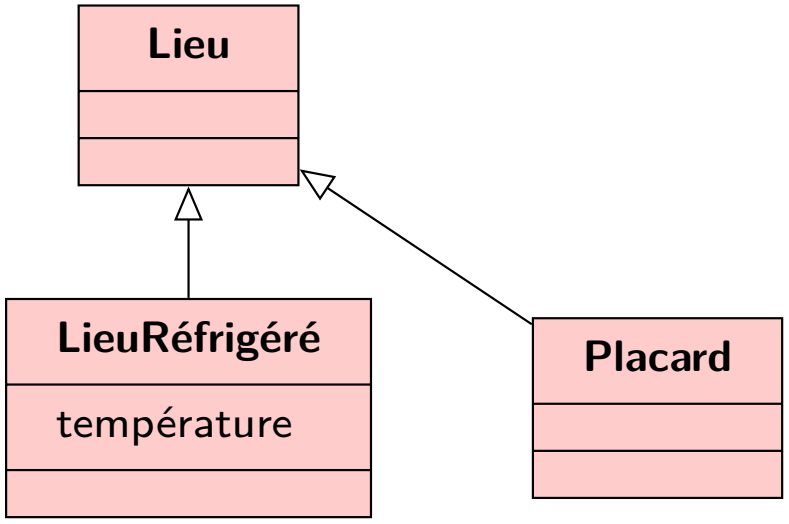
Contravariance
sur le type de paramètre
m(A) spécialise m(B)
B spécialise A

Covariance
sur le type de retour
n():D spécialise n():C
D spécialise C

Imaginons une redéfinition de méthode avec covariance du paramètre, contraire au principe de substitution de Liskov

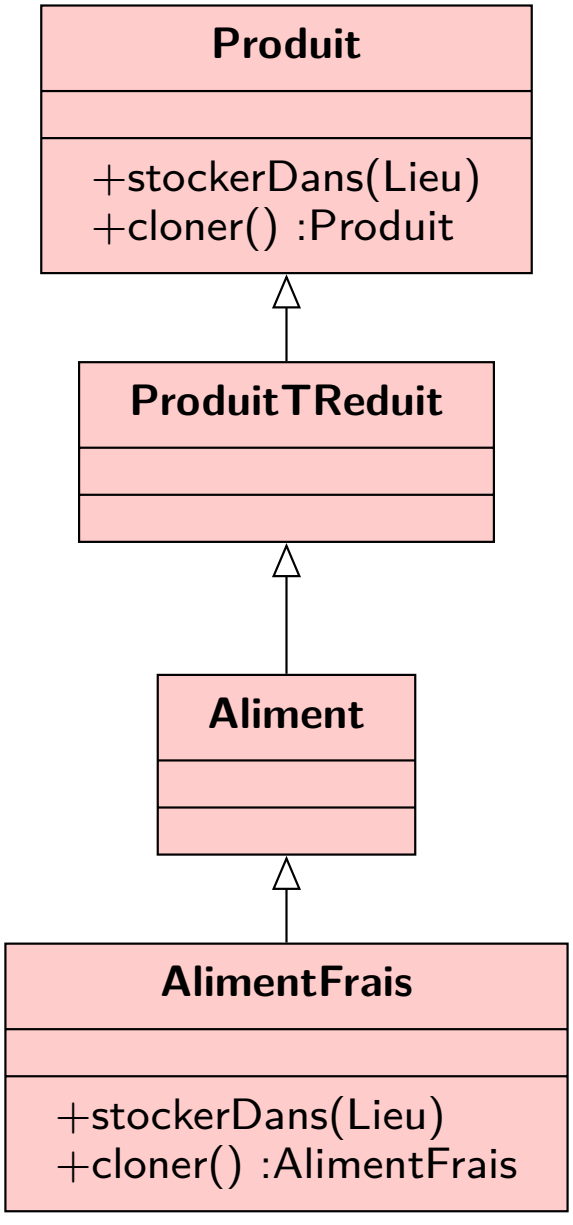


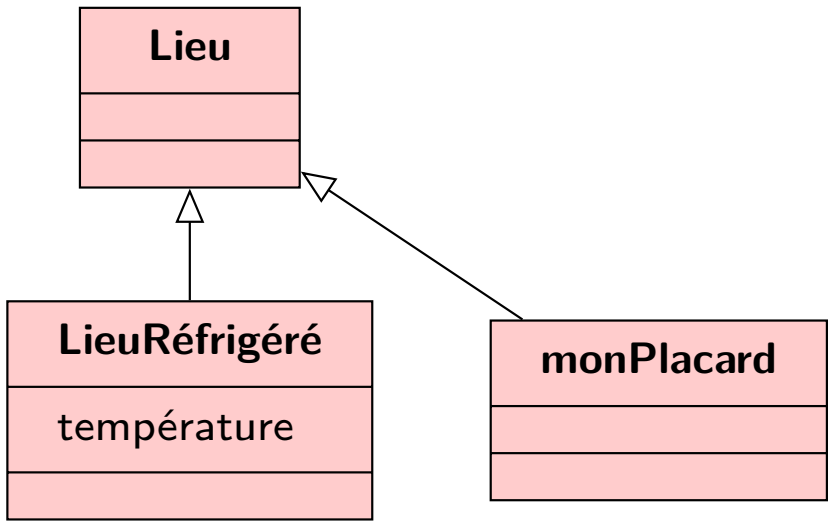





```
Produit p = new AlimentFrais (...);
Lieu monPlacard = new Placard(...);
p.stockerDans(monPlacard);
```

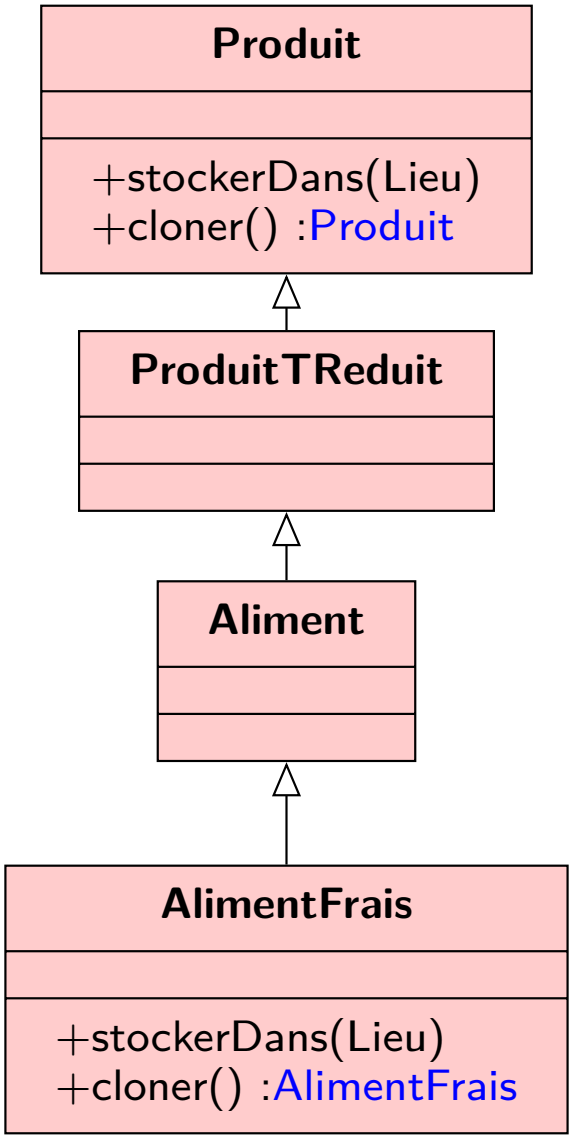
Compile mais à l'exécution poserait problème par exemple si stocker dans un lieu réfrigéré fait accès à la température, inconnue pour les lieux ordinaires.

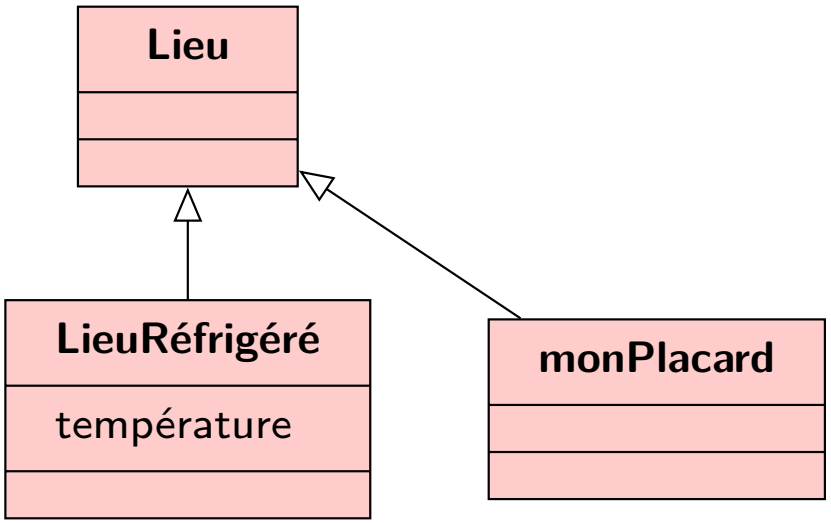




La contravariance sur les types des paramètres n'est pas en accord avec la spécialisation
L'invariance choisie par Java est un choix raisonnable.

Redéfinition légale pour le principe de substitution de Liskov (covariance du type de retour)





La covariance du type de retour,
conforme au principe de
substitution de Liskov
est de plus en accord
avec la spécialisation
La covariance choisie par Java
est en accord avec les deux

Substituabilité - Comportement

Liskov

Lorsqu'un type S est sous-type d'un type T, tout objet de type S peut remplacer une expression de type T sans que le programme n'ait de comportement imprévu ou ne perde de propriétés souhaitées.

Exemple de problème de substitution comportementale avec :

- S : Square
- T : Rectangle

Pourtant un carré est un rectangle du point de vue de la spécialisation

```
1  public class Rectangle {
2      private double width, height;
3
4      public Rectangle() {}
5      public Rectangle(double width, double height) {
6          this.width = width; this.height = height;
7          assert(this.getWidth()==width&&this.getHeight()==height);
8      }
9
10     public double getWidth()
11         { return width; }
12     public void setWidth(double width)
13         { this.width = width; assert(this.getWidth() == width); }
14     public double getHeight()
15         { return height; }
16     public void setHeight(double height) {
17         this.height=height;
18         assert(this.getHeight() == height);
19     }
20     public String toString()
21         { return "Rectangle:"+this.getWidth()+" "+this.getHeight();}
22 }
```

```
1 public static void mystery(Rectangle r) {
2         r.setWidth(5.0);
3         System.out.println(r);
4         r.setHeight(4.0);
5         System.out.println(r);
6         assert(r.getWidth() * r.getHeight() == 20.0);
7 }
```

Attention :

- utiliser r. car nom du paramètre
- Méthode statique : this n'a pas sens
- Les attributs sont privés : on utilise les accesseurs

```
1  public class Square extends Rectangle {
2          public Square() {}
3          public Square(double width) {
4                  super(width, width);
5                  assert(getWidth()==getHeight());
6          }
7          public void setWidth(double width) {
8                  super.setWidth(width);
9                  super.setHeight(width);
10                 assert(getWidth()==getHeight());
11                 assert(this.getWidth() == width);
12         }
13         public void setHeight(double height) {
14                 this.setWidth(height);
15         }
16         public String toString() {
17                 return super.toString()+"Square:"+this.getWidth();
18         }
19 }
```

Attention :

- les attributs sont privés : on utilise les accesseurs
- notez un cas d'appel super.setHeight() dans une méthode nommée setWidth(), ce qui est rare !

```
1  public static void main(String[] args) {
2                  Rectangle r = new Rectangle(8.0, 7.0);
3                  System.out.println(r);
4                  mystery(r); // se passe bien...
5
6                   r = new Square(3.0);
7                  System.out.println(r);
8                  mystery(r);
9                  // la derniere instruction de mystery,
10                 // r.setHeight() attribue 4 a la hauteur
11                 // ET a la largeur donc
12                 // r.getWidth() * r.getHeight() == 16.0
13                 // Declenchement de l'assertionError
14 }
```

Conclusion sur l'exemple rectangles et carrés

Conclusion :

- Même si un square peut remplacer un rectangle dans le programme Java sans erreur de compilation...
- ...il ne se comporte pas de la même manière.
- Le principe de substitution de Liskov n'est pas vérifié en ce qui concerne la partie comportementale.
- Justification par le programme `main` précédent où le comportement du programme est sain pour un rectangle et est altéré pour le carré.

Plan

Deux regards sur l'envoi de message

Envoi de message

class CEXP { void x(){CREC rec ; ... rec.m() ....} }
CEXP exp ; ... exp.x() ; ...

- l'expéditeur *exp* (objet courant - receveur de la méthode *x* où se trouve l'envoi de message, le type statique de *exp* est la classe *CEXP*) ;
- le message *m* (nom de méthode, types de paramètres et retour pour les cas simples) ;
- le receveur *rec* (objet sur lequel est appliquée la méthode, cet objet a un type statique, type de la variable qui désigne l'objet, *CREC*).

Contrôles

- Compilation : le receveur peut-il recevoir le message ?
- Visibilité, contrôle d'accès statique : l'expéditeur a-t-il le droit d'envoyer le message au receveur ?

Modèles possibles de contrôle d'accès statique dans différents langages

Eléments concernés (qu'est-ce qui est contrôlé par un accès ?)

- types
- attributs, méthodes
- relation d'héritage

Granularité de l'accès

- niveau de la classe (même accès quel que soit l'objet)
- niveau de l'objet

Mode

- nominatif (en désignant explicitement une entité qui peut accéder)
- anonyme

Modèle de contrôle d'accès statique en Java

Eléments concernés (qu'est-ce qui est contrôlé par un accès ?)

- types
- attributs, méthodes
- ~~relation d'héritage~~

Granularité de l'accès

- niveau de la classe (même accès quel que soit l'objet)
- ~~niveau de l'objet~~

Mode

- ~~nominatif (en désignant explicitement une entité qui peut accéder)~~
- anonyme

Niveaux de protection/visibilité en JAVA

Quatre directives de visibilité peuvent être placées sur les membres d'une classe (attributs, méthodes, classes internes)

private - (privé) accessible seulement par les méthodes de cette classe

rien ∼ (package) accessible seulement par les méthodes des classes du même package

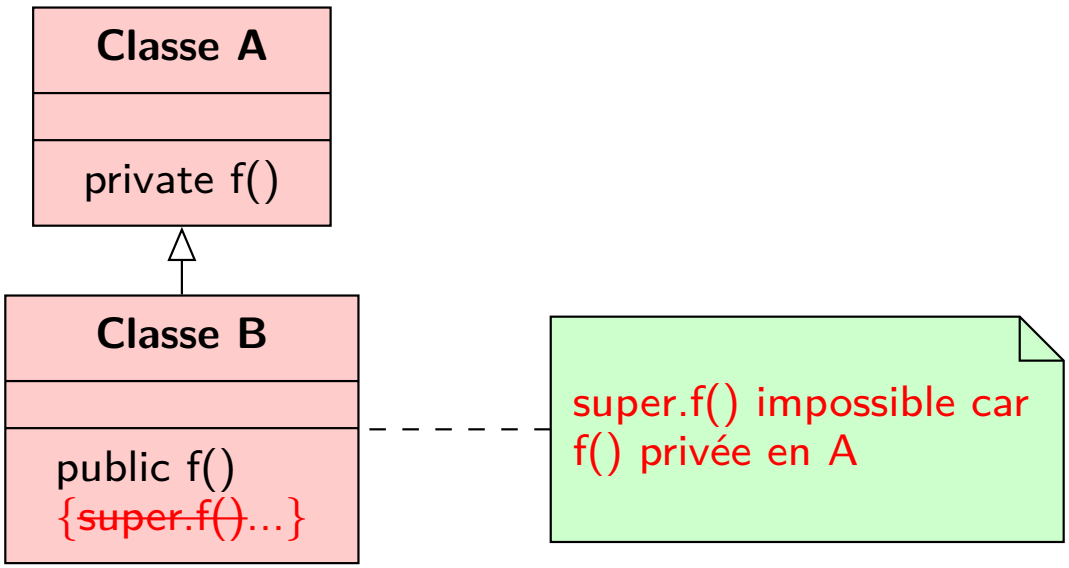
protected # (protégé) accessible seulement par les méthodes des classes du même package et des classes dérivées
Hors du package, les propriétés protégées de C sont accessibles par une sous-classe Cs de C soit sur ses propres instances, créées avec new Cs(...), soit sur des instances de ses sous-classes Css, créées avec new Css(...), et jamais en remontant

public + (public) accessible sans restriction

Recommandation

De manière générale, les attributs sont privés (sauf cas particuliers, comme certaines constantes).

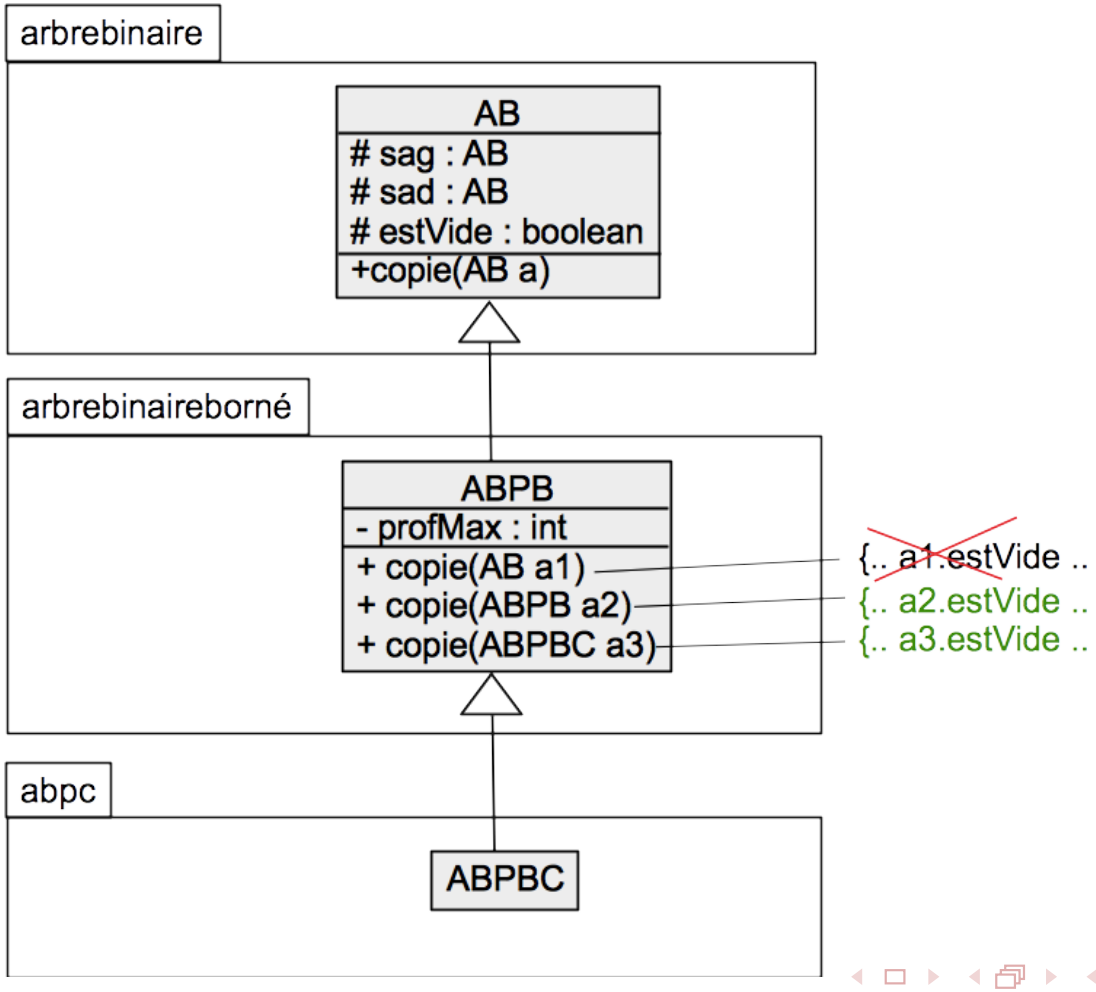
Peut-on "redéfinir" une méthode privée ?



On ne redéfinit pas une méthode privée, ce qui est présenté doit être compris comme le fait que B définit une nouvelle méthode.

Héritage et visibilité

Hors du package, les propriétés protégées de C sont accessibles par une sous-classe Cs de C soit sur ses propres instances, créées avec new Cs(...), soit sur des instances de ses sous-classes Css, créées avec new Css(...), et jamais en remontant



Héritage et visibilité

```
package visibilite.arbrebinaire;
public class ArbreBinaire {
        protected ArbreBinaire sag, sad;
        protected boolean estVide; // dans ce cas, sag et sad ↩
            inutiles

        public void copie(ArbreBinaire a)
        {
                this.estVide = a.estVide;
                if (! a.estVide)
                // code qui copie a dans this sans regarder la ↩
                    profondeur
                // ...
                {

                } ...} ...}
```

Héritage et visibilité

```
package visibilite.arbrebinaireborne;
import visibilite.abpc.ArbreBinaireProfBorneeColore;
import visibilite.arbrebinaire.ArbreBinaire;
public class ArbreBinaireProfondeurBornee extends ArbreBinaire{
        // la structure est deja definie, on va seulement verifier la profondeur de l'arbre
        private int profondeurMax;

        // on ne peut pas acceder au champ protected sur un arbre binaire qui est la superclasse et cette methode est bien une redefinition de copie de la superclasse
        public void copie(ArbreBinaire a){
                this.estVide = a.estVide; // PB
                if (! a.estVide){ // PB
                // code qui copie a dans this jusqu'a profondeurMax
                ....}
        }

        // on  peut acceder au champ protected sur un arbre de cette classe mais cette methode n'est pas une redefinition de copie de la superclasse
        public void copie(ArbreBinaireProfondeurBornee a){
                this.estVide = a.estVide;
                if (! a.estVide){
                // code qui copie a dans this jusqu'a profondeurMax
```

Héritage et visibilité

```
package visibilite.abpc;
import visibilite.arbrebinaire.ArbreBinaire;
import visibilite.arbrebinaireborne.ArbreBinaireProfondeurBornee;

public class ArbreBinaireProfBorneeColore extends ↩
    ArbreBinaireProfondeurBornee {


}
```

Héritage et visibilité

```
package visibilite.arbrebinaireborne;
import visibilite.abpc.ArbreBinaireProfBorneeColore;
import visibilite.arbrebinaire.ArbreBinaire;

public class ArbreBinaireProfondeurBornee extends ArbreBinaire{
// suite
        // on  peut acceder au champ protected sur un arbre d'une ↩
             sous-classe de cette classe
        // mais cette methode n'est pas une redefinition de copie ↩
             de la superclasse
        public void copie(ArbreBinaireProfBorneeColore a)
        {
                this.estVide = a.estVide;
                if (! a.estVide){
                // code qui copie a dans this jusqu'a ↩
                     profondeurMax
                // ...
                }
        }
}
```

Héritage et visibilité

Solutions :

- Les classes représentant les arbres sont dans le même package
- Toutes les manipulations de la structure passent par des méthodes
- Autres idées ?

Plan

Synthèse

1. Eléments sur la structure des hiérarchies (héritage, instanciation)
2. Redéfinition versus surcharge statique (attention aux règles de redéfinition)
3. Coercition et tests de type ((T), instanceof)
4. Spécialisation naturelle versus substituabilité
   - Des décisions à prendre lors de la modélisation et de la programmation
   - La spécialisation naturelle est structurante pour la pensée, l'organisation et la compréhension du programme et ne demande pas de tout prévoir
   - La substituabilité permet de limiter les erreurs à l'exécution, mais demande d'anticiper
5. Visibilité (public $<$ protected $<$ package $<$ private)