IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Héritage et concepts avancés

F. Dagnat & J. Mallet & A. Beugnard & J.C. Royer
DLR – MAPD – C3
2023

Objectif de ce cours

Fin des rappels :

1. Héritage
2. Notion de type
3. Liaison dynamique
4. Interface
5. Exception
6. Généricité (usage)

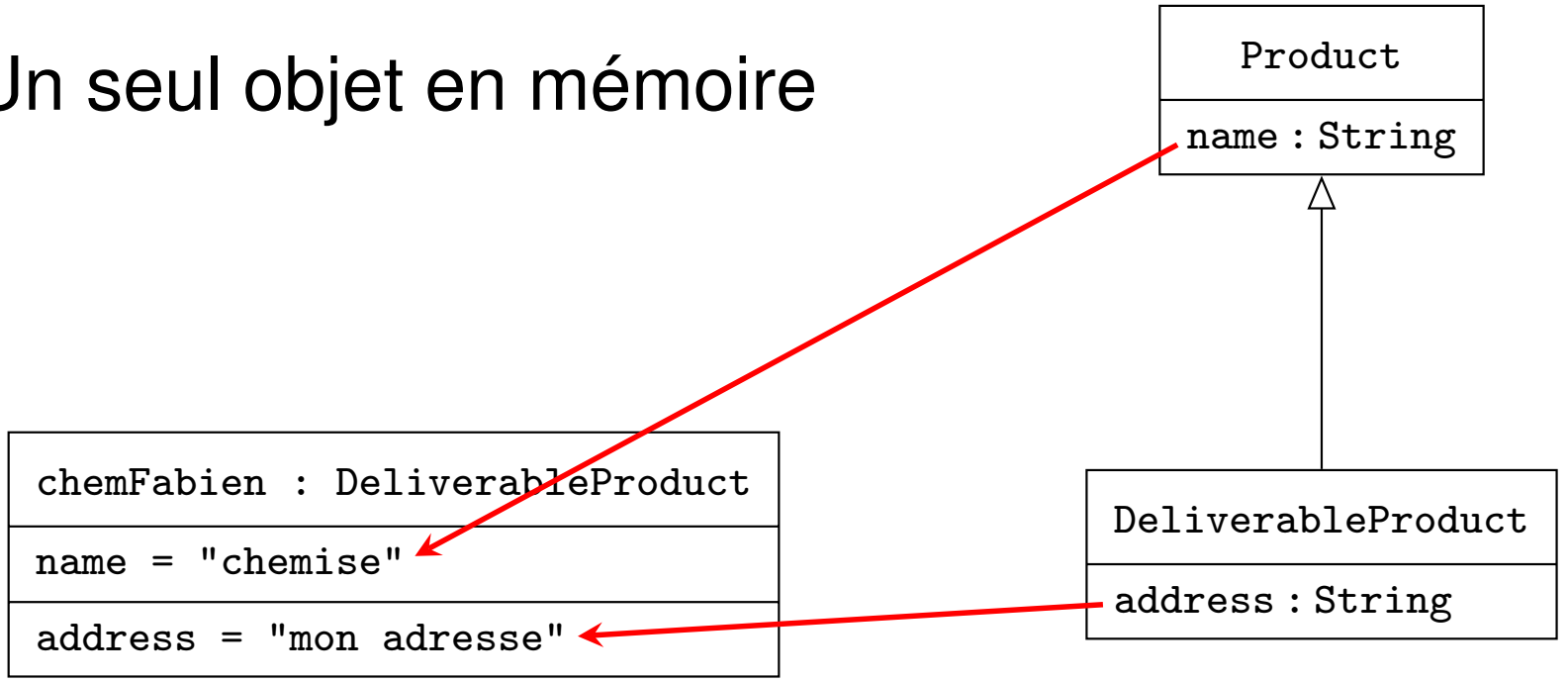
Contribution aux compétences

1. Conception
2. Développement

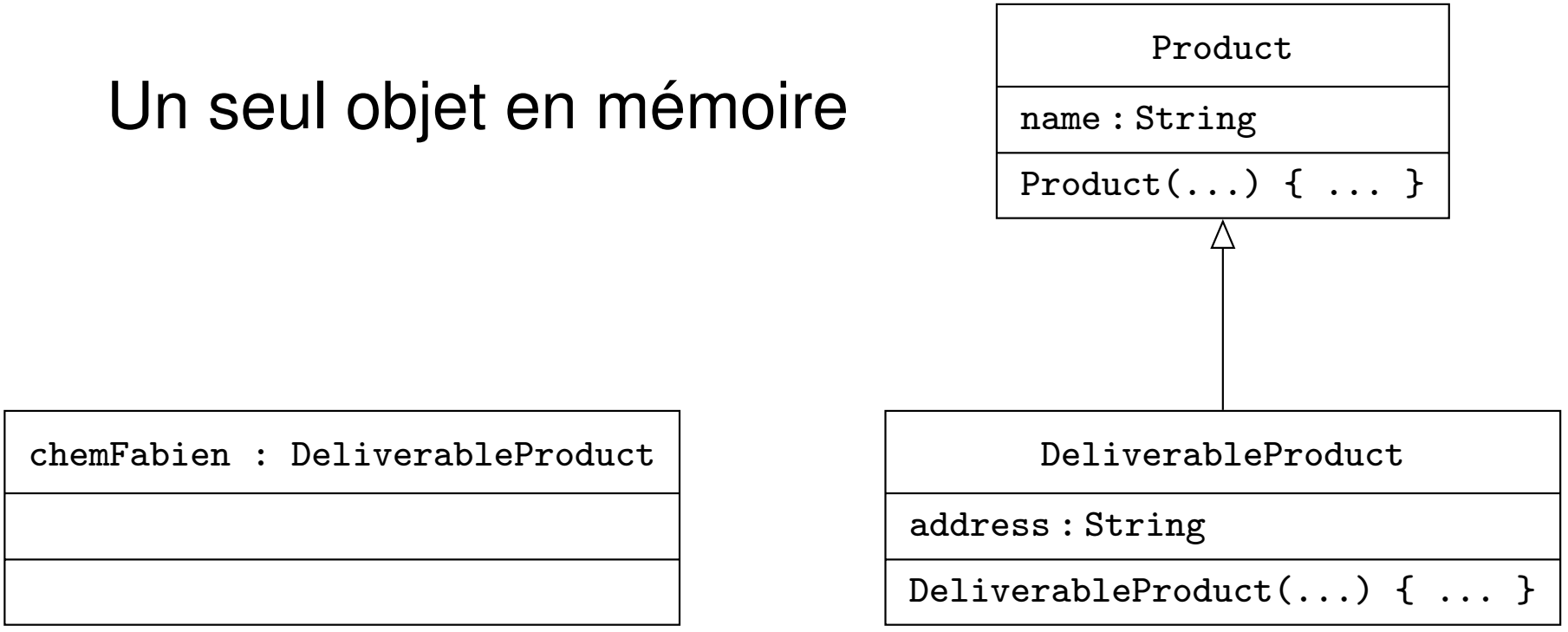
Je vous invite à prendre des notes. . .

```java
public class DeliverableProduct extends Product {
  String address ; // un attribut ajouté
  public DeliverableProduct(String name, String address) {
    super(name);
    this.address = address;
  }
  public String getAddress() { // méthode ajoutée
    return this.address;
  }
  public String getName() { // méthode redéfinie
    return super.getName() + " pour destination "
           + this.address;
  }
}
```

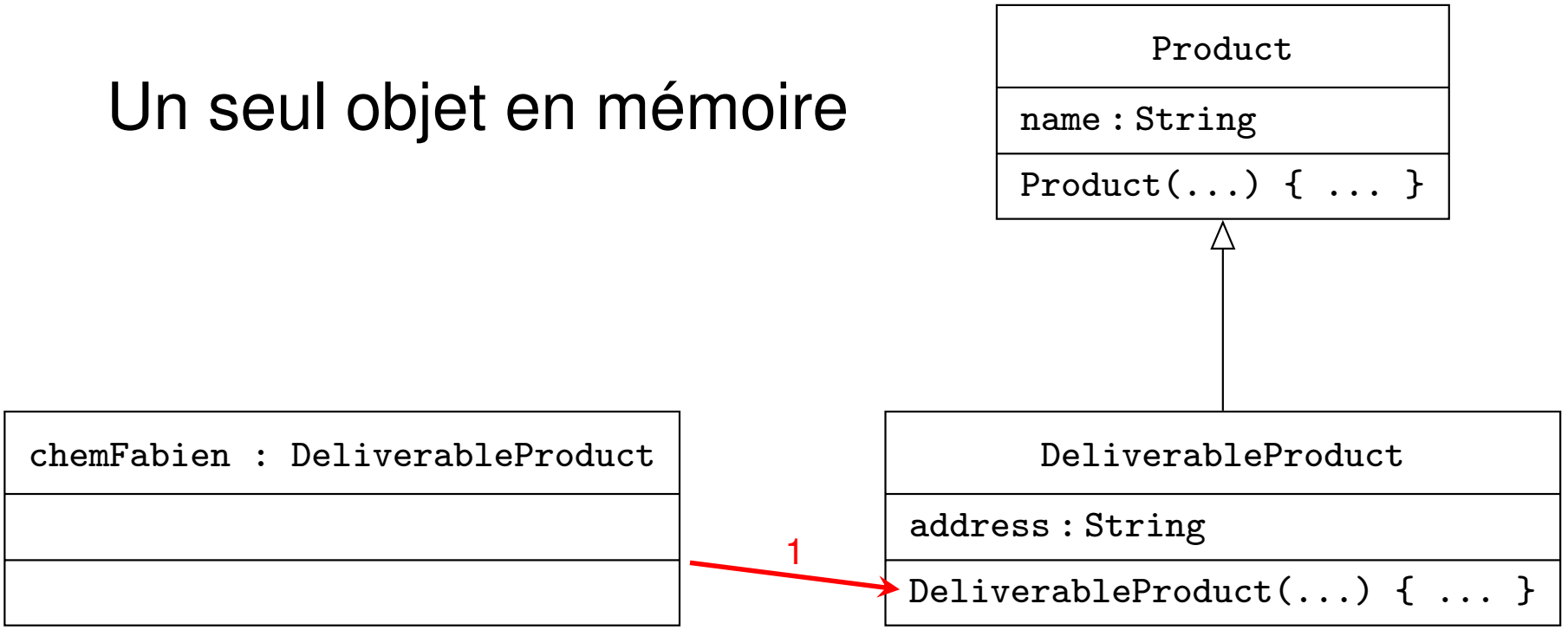
Un seul objet en mémoire



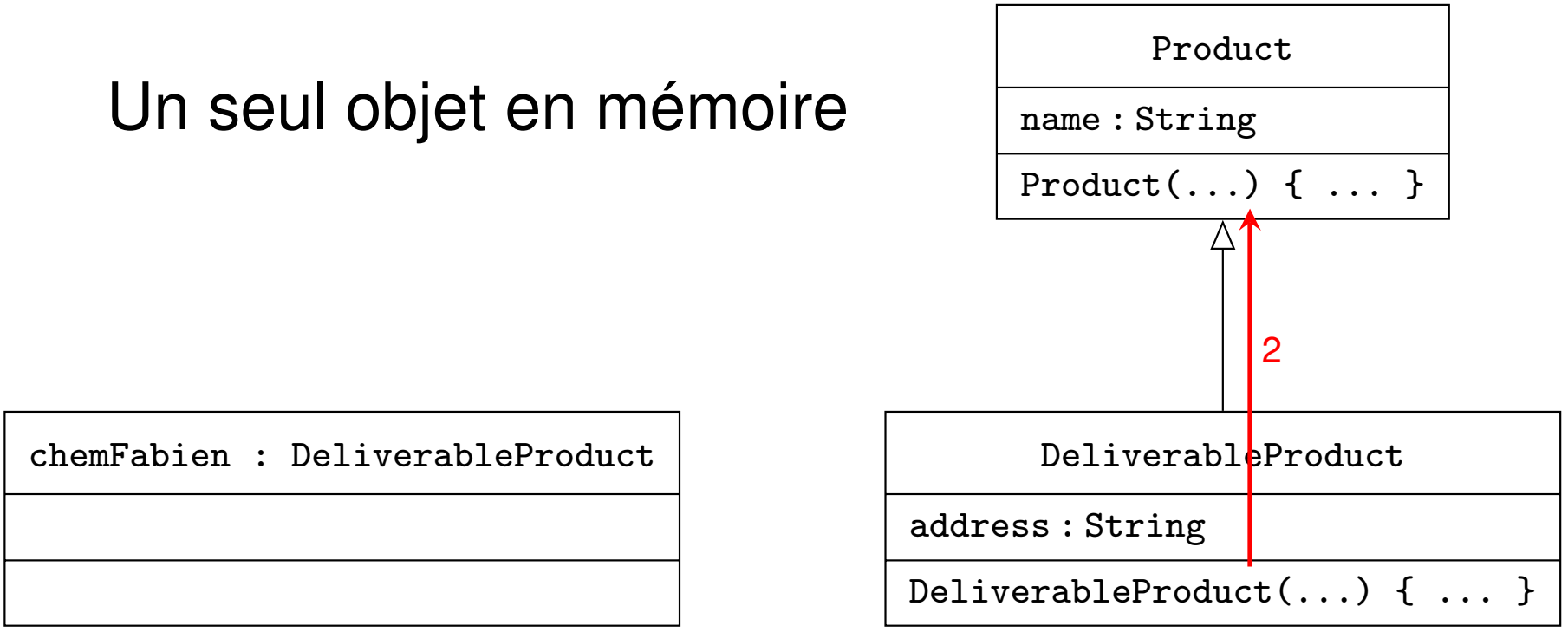
Un seul objet en mémoire



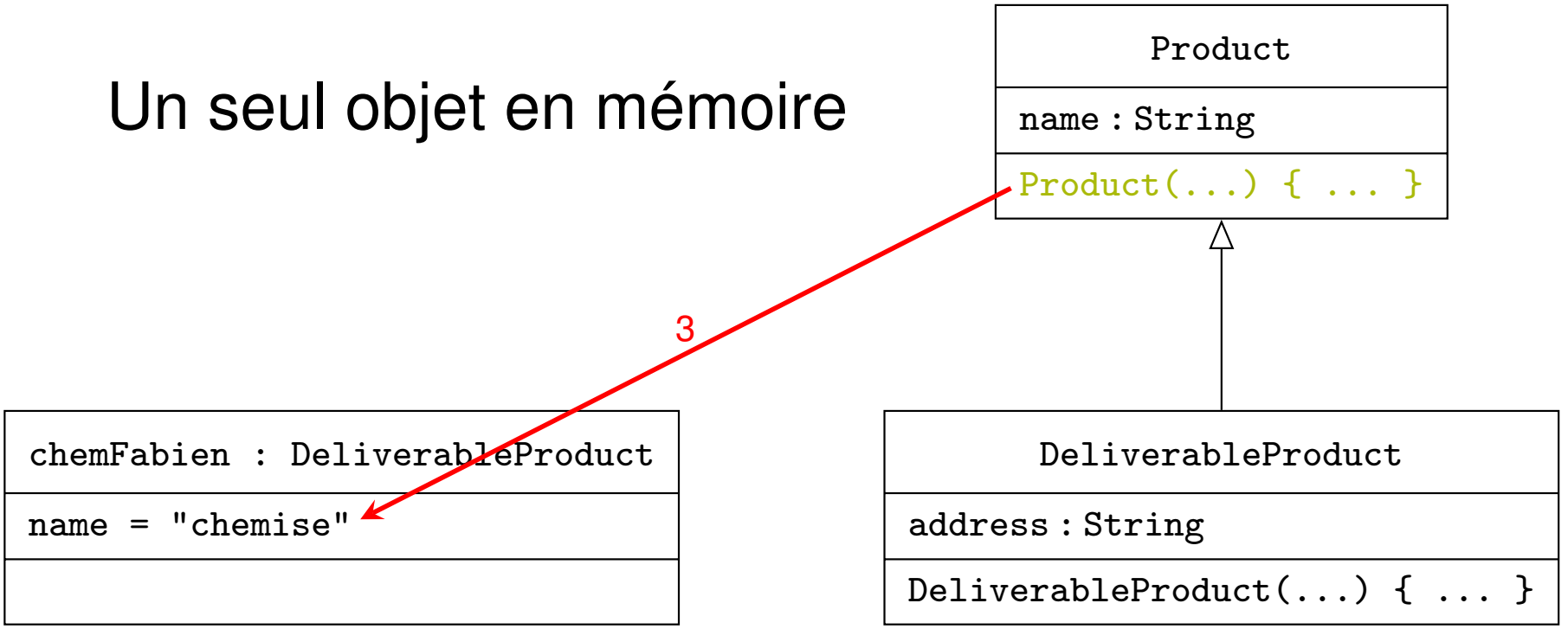
Un seul objet en mémoire



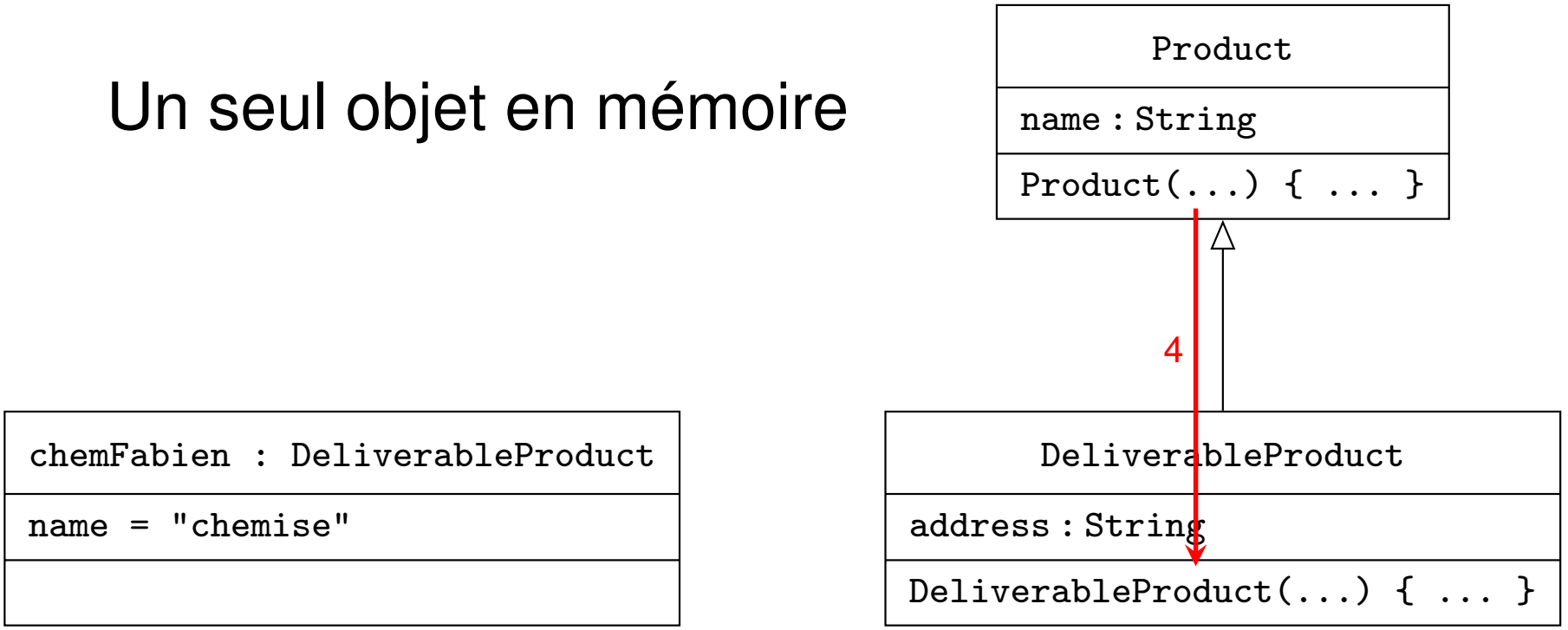
Un seul objet en mémoire



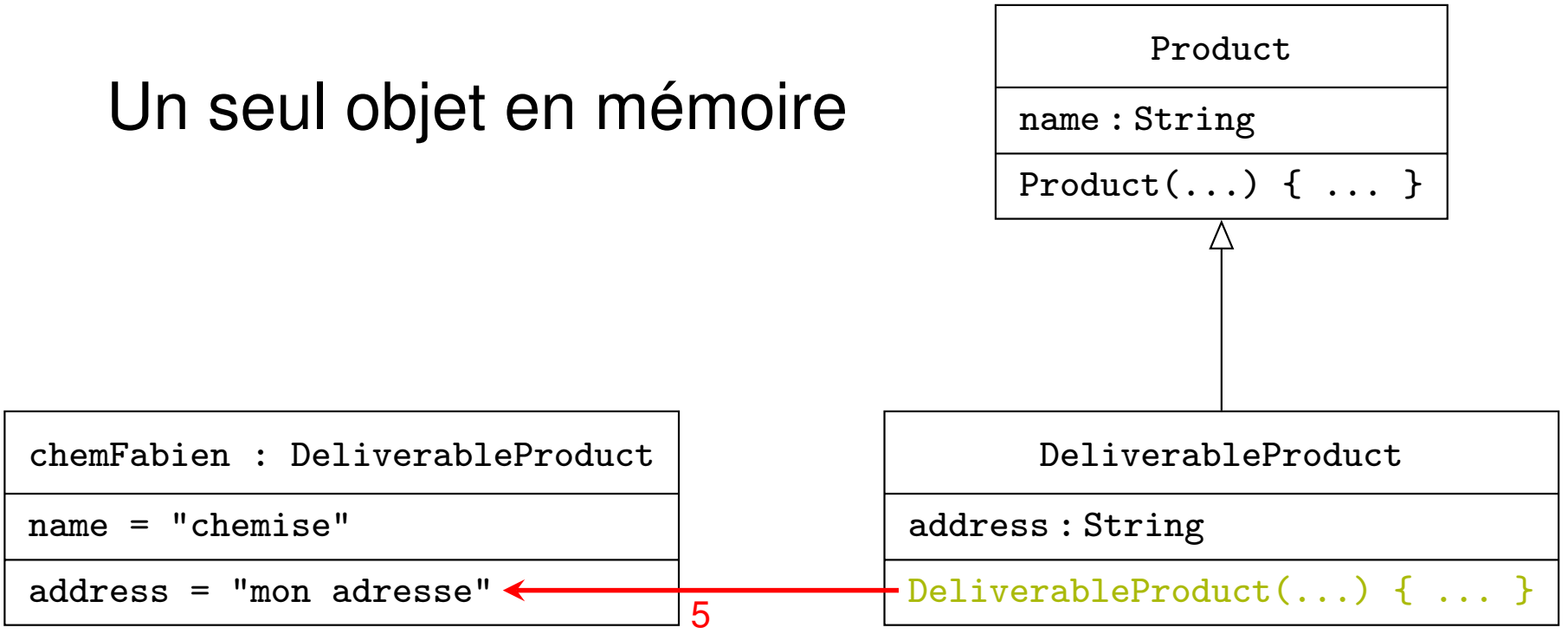
Un seul objet en mémoire



Un seul objet en mémoire



Un seul objet en mémoire



- À un autre constructeur de la même classe

```
1 public BankAccount(String first, String last) {
2   this(first, last, 0); ... }
```

- Explicite à un constructeur de la classe mère

```
1 public BankAccount(String first, String last) {
2   super(first, last); ... }
```

- Implicite à un constructeur de la classe mère (ajouté par le compilateur)

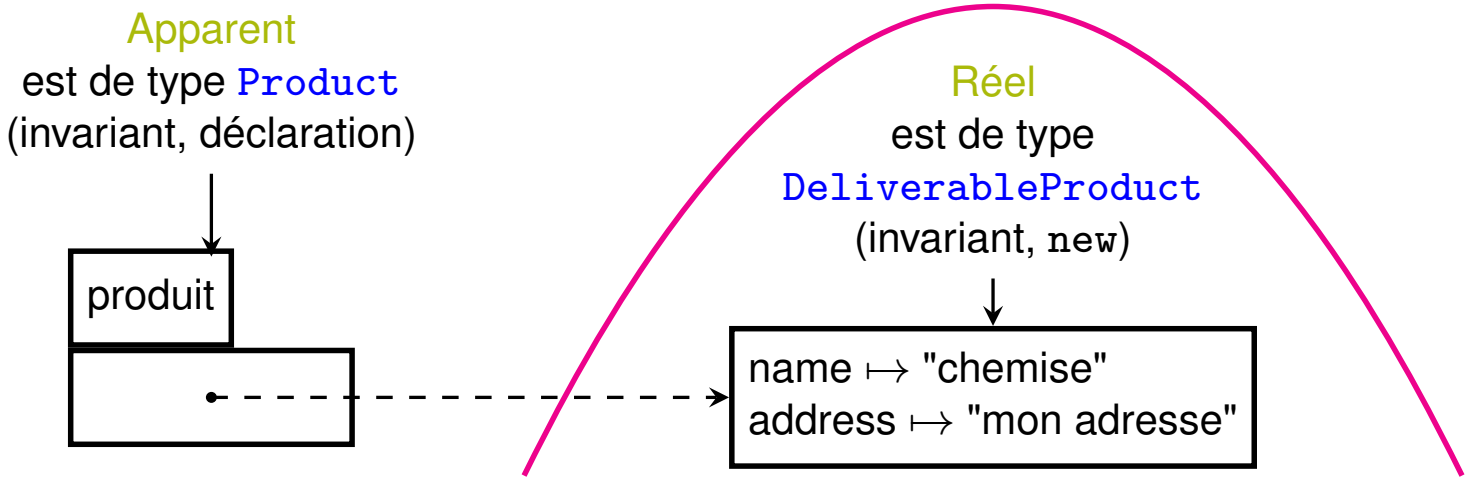
```
1 public BankAccount(String first, String last) {
2   /* n'importe quoi sauf this ou super */ ... }
```

- Ajout d'un « modifieur » de visibilité

Java	UML	accessible par
`private`	`-`	les objets de même classe
rien	`~`	les objets de classes du même paquetage
`protected`	`#`	les objets de classes héritant ou du même paquetage
`public`	`+`	tous (plus d'encapsulation)

- *rien* = visibilité dite *default*

- `Product produit = new DeliverableProduct("chemise","mon adresse");`



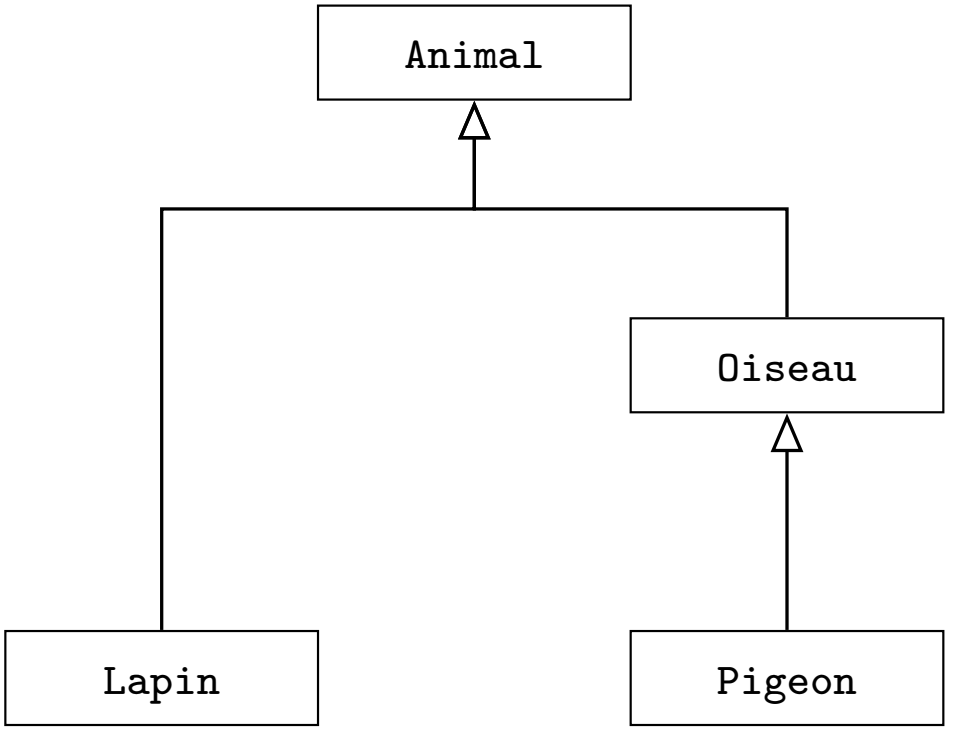
- Réel $<:$ Apparent
- Une expression Java aura aussi ces deux types

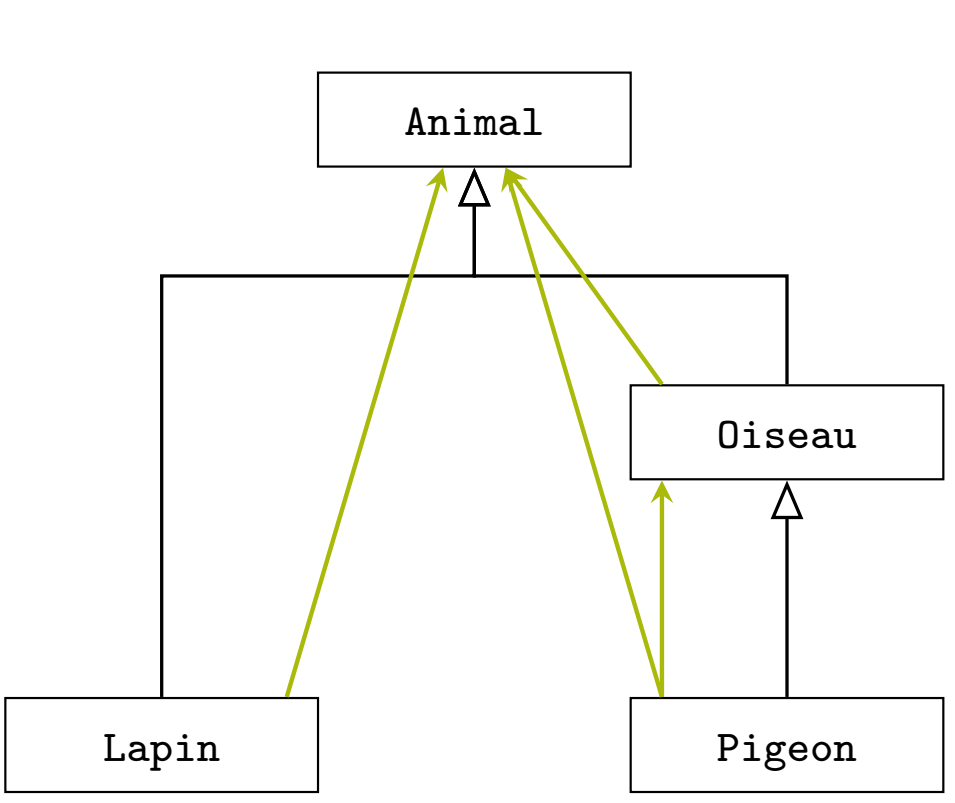
On utilise aussi le vocabulaire type statique (apparent) et dynamique (réel)

- Comment utiliser `produit` comme un `DeliverableProduct` ?
- On transtype (*cast*) vers le type `DeliverableProduct`
  - `(DeliverableProduct) produit`
- `ClassCastException` levée si l'objet n'est pas du type demandé
- On peut tester si un objet est d'un type

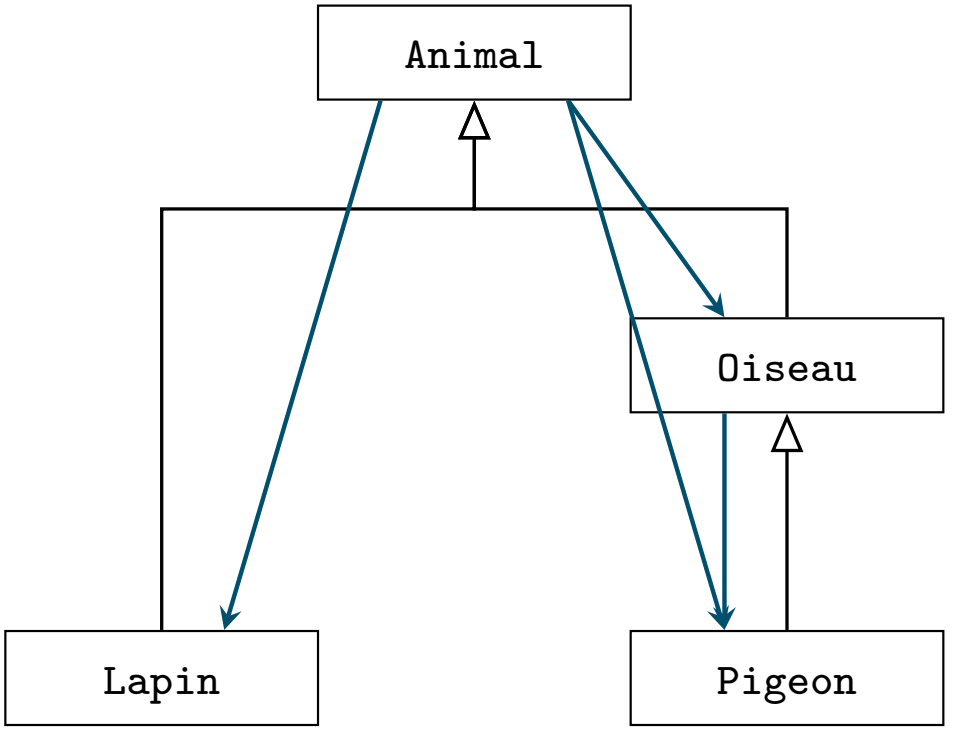
```
1 DeliverableProduct pl;
2 if (produit instanceof DeliverableProduct)
3   pl = (DeliverableProduct) produit;
```

S'utilise avec parcimonie


Animal
Oiseau
Lapin
Pigeon

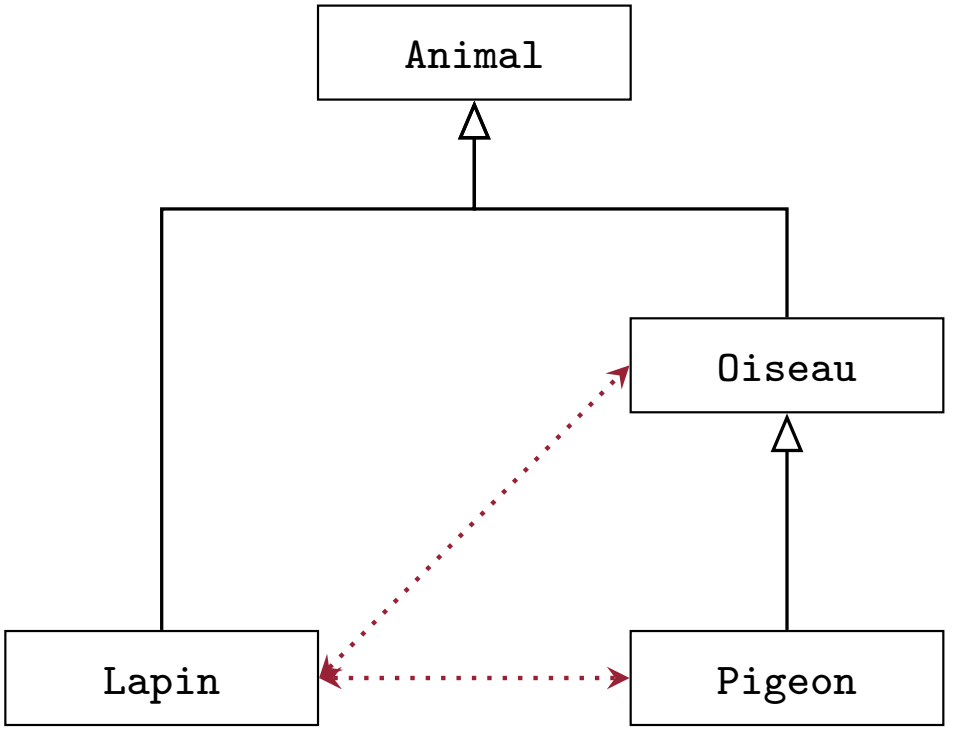



```
Animal a = new Lapin();
```



Conversion nécessaire (*cast*)

```
Animal a = new Lapin();
Lapin l = (Lapin) a;
Pigeon p = (Pigeon) a;
```

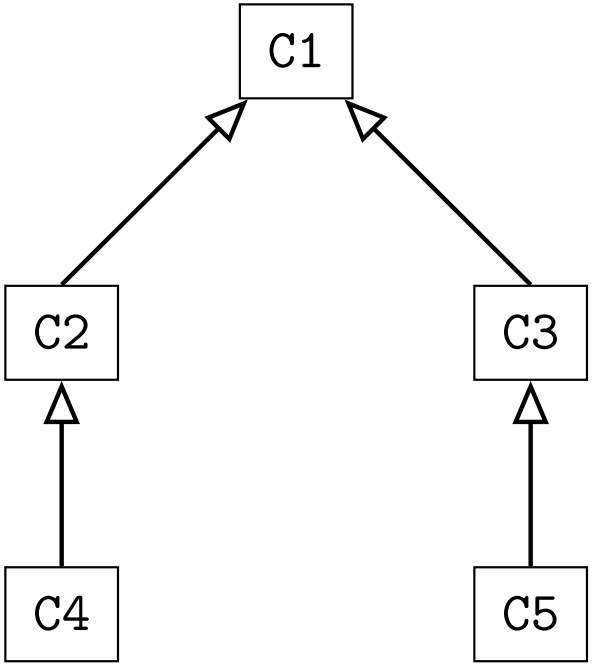


Erreur de compilation

```
Animal a = new Lapin();
Lapin l = (Lapin) a;
Pigeon p = (Pigeon) a;
Pigeon p = new Lapin();
```

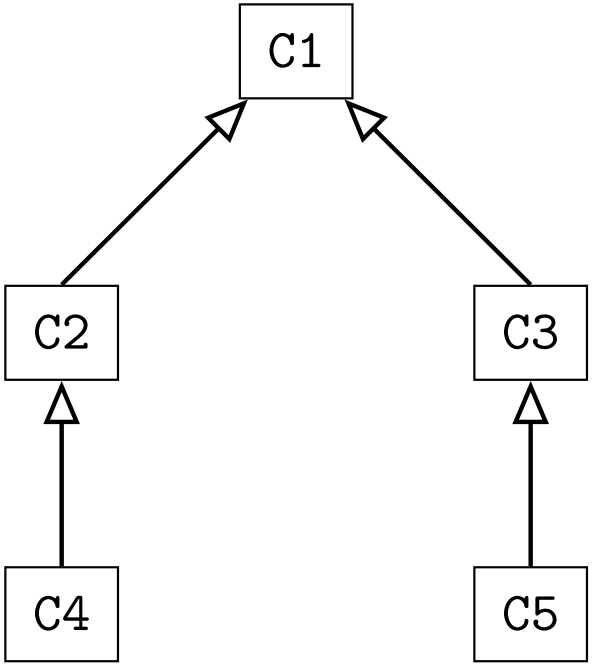
```
C1 a = new C1();
C3 b = new C5();
```

Variable	Type apparent	Type réel
a		
b		



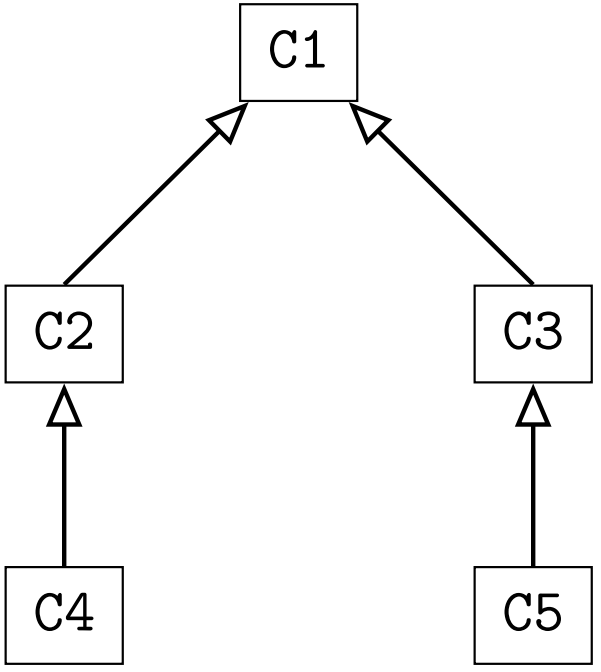
```
C1 a = new C1();
C3 b = new C5();
```

Variable	Type apparent	Type réel
a	C1	C1
b	C3	C5



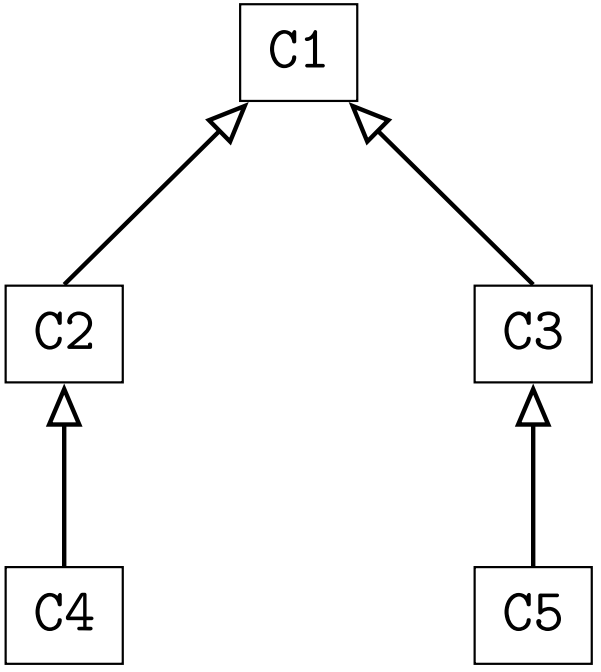
```
1  C1 a = new C1();
2  C3 b = new C5();
3  C1 c = new C5();
4  C1 d = new C4();
5  C3 e = c;
6  d = c;
7  b = new C2();
8  C4 f = d;
```

L.	Comp.	Correction	Exéc.
3			
4			
5			
6			
7			
8			



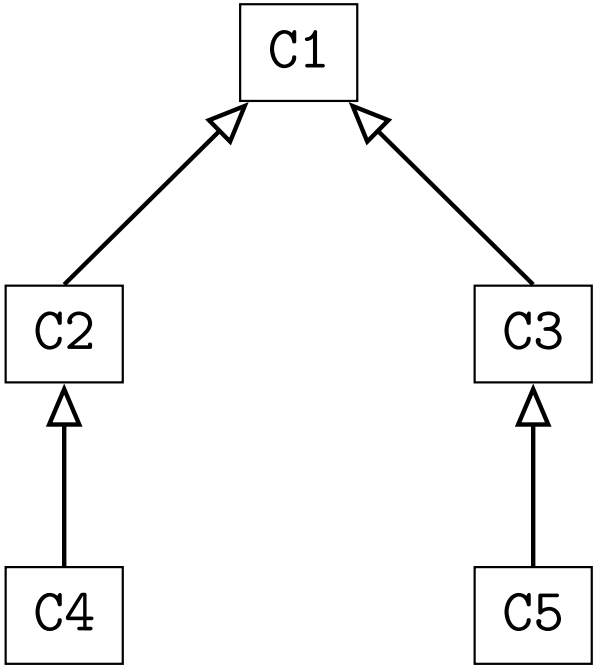
```
1  C1 a = new C1();
2  C3 b = new C5();
3  C1 c = new C5();
4  C1 d = new C4();
5  C3 e = c;
6  d = c;
7  b = new C2();
8  C4 f = d;
```

L.	Comp.	Correction	Exéc.
3	OK		OK
4	OK		OK
5	Err		
6	OK		OK
7	Err		
8	Err		



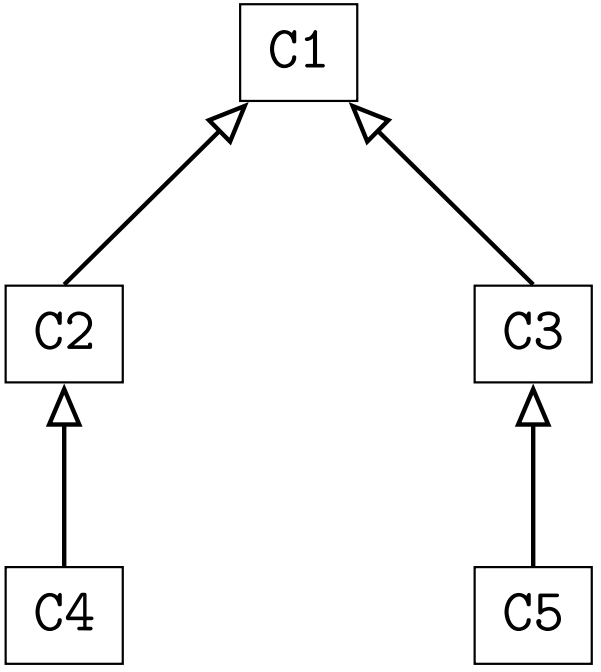
```
1  C1 a = new C1();
2  C3 b = new C5();
3  C1 c = new C5();
4  C1 d = new C4();
5  C3 e = c;
6  d = c;
7  b = new C2();
8  C4 f = d;
```

L.	Comp.	Correction	Exéc.
3	OK		OK
4	OK		OK
5	Err	`C3 e = (C3) c`	OK
6	OK		OK
7	Err		
8	Err		



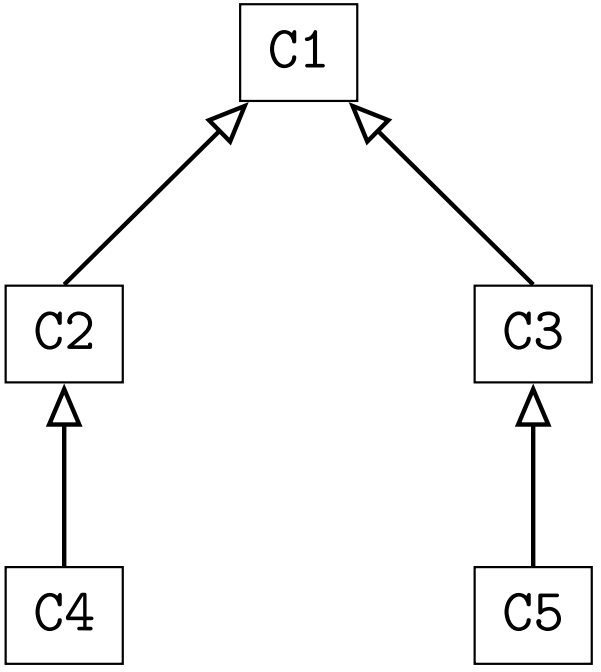
```
1  C1 a = new C1();
2  C3 b = new C5();
3  C1 c = new C5();
4  C1 d = new C4();
5  C3 e = c;
6  d = c;
7  b = new C2();
8  C4 f = d;
```

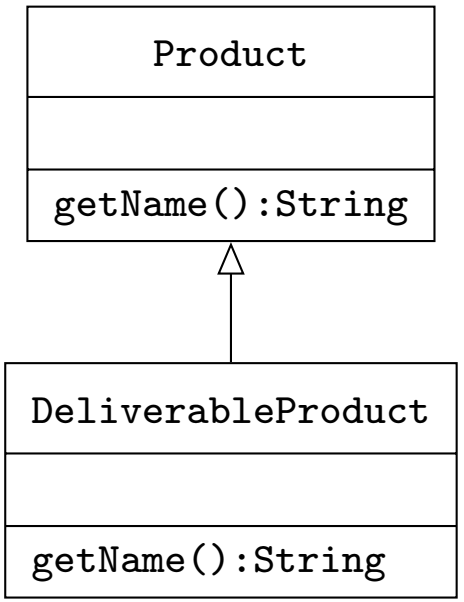
L.	Comp.	Correction	Exéc.
3	OK		OK
4	OK		OK
5	Err	`C3 e = (C3) c`	OK
6	OK		OK
7	Err	Pas possible	
8	Err		



```
1  C1 a = new C1();
2  C3 b = new C5();
3  C1 c = new C5();
4  C1 d = new C4();
5  C3 e = c;
6  d = c;
7  b = new C2();
8  C4 f = d;
```

L.	Comp.	Correction	Exéc.
3	OK		OK
4	OK		OK
5	Err	`C3 e = (C3) c`	OK
6	OK		OK
7	Err	Pas possible	
8	Err	`C4 f = (C4) d`	Err

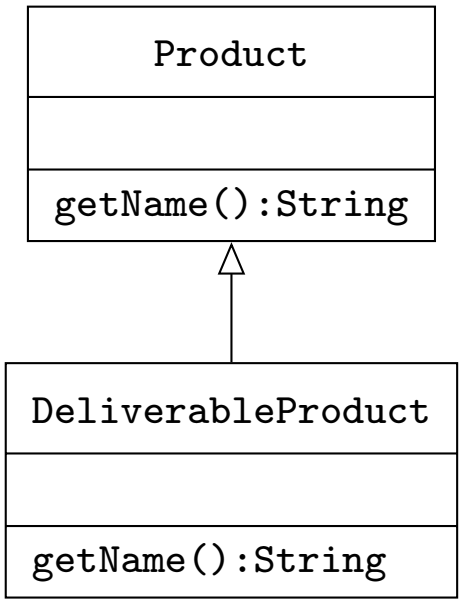





```
public class Product{
...
  public String getName(){return name;}
}

public class DelivrableProduct extends Product{
...
  public String getName(){
    return super.getName()+" pour destination "
           +adress;
  }
}
```

```
Product p = new DelivrableProduct("courgette","TB");
System.out.println(p.getName());
```




```
public class Product{
...
  public String getName(){return name;}
}

public class DelivrableProduct extends Product
...
  public String getName(){
    return super.getName()+" pour destination
           +adress;
  }
}
```

```
int i = ...;
Product p;
if (i==1) p = new DelivrableProduct("carotte","TB");
else p = new Product("poireau");
System.out.println(p.getName());
```

- Invocation $\Rightarrow$ recherche de la méthode dans la classe de l'objet cible :
  - Si elle est trouvée, elle est exécutée
  - Sinon, on la recherche dans la classe mère
- Recherche dynamique car type réel de l'objet non connu avant l'exécution
- L'exécution d'une méthode repose sur le type :
  - apparent de sa référence pour l'existence de la méthode et sa visibilité (fait par le compilateur)
  - réel de l'objet pour le code exécuté (fait par la machine virtuelle)
- Appelé liaison tardive ou liaison dynamique

- Il est possible de redéfinir une méthode *non statique*.
- Les deux méthodes doivent :
  - avoir le même nom,
  - avoir la même signature [1],
  - le type de retour de la redéfinition est un sous-type de celui de la méthode initiale.
- On peut accéder à l'ancienne version par `super`.
- On peut relâcher la visibilité (p.ex. `protected` $\rightarrow$ `public`) mais pas la réduire (sous-typage).

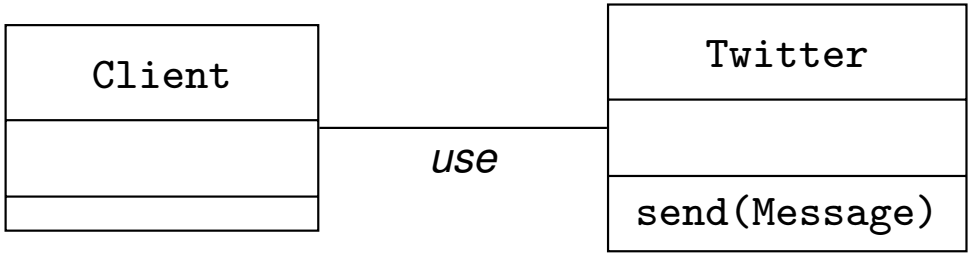
1. On parle de redéfinition *invariante*.

- En Java, une classe qui n'étend personne étend la classe `Object`
- Cette classe est la racine de l'arbre d'héritage
- En fait : toutes les références sont de type `Object` (même tableaux)
- Conséquence : toutes les méthodes de `Object` peuvent être utilisées sur les références
- De plus, par la liaison dynamique, c'est le corps le plus spécialisé qui sera exécuté

```
1  package java.lang;
2
3  public class Object {
4    public String toString() {
5      return getClass().getName() + "@" + Integer.toHexString(hashCode());
6    }
7    public boolean equals(Object obj) { return (this == obj); }
8
9    protected native Object clone() throws CloneNotSupportedException;
10
11   public final native Class getClass();
12   public native int hashCode();
13   ...
14 }
```

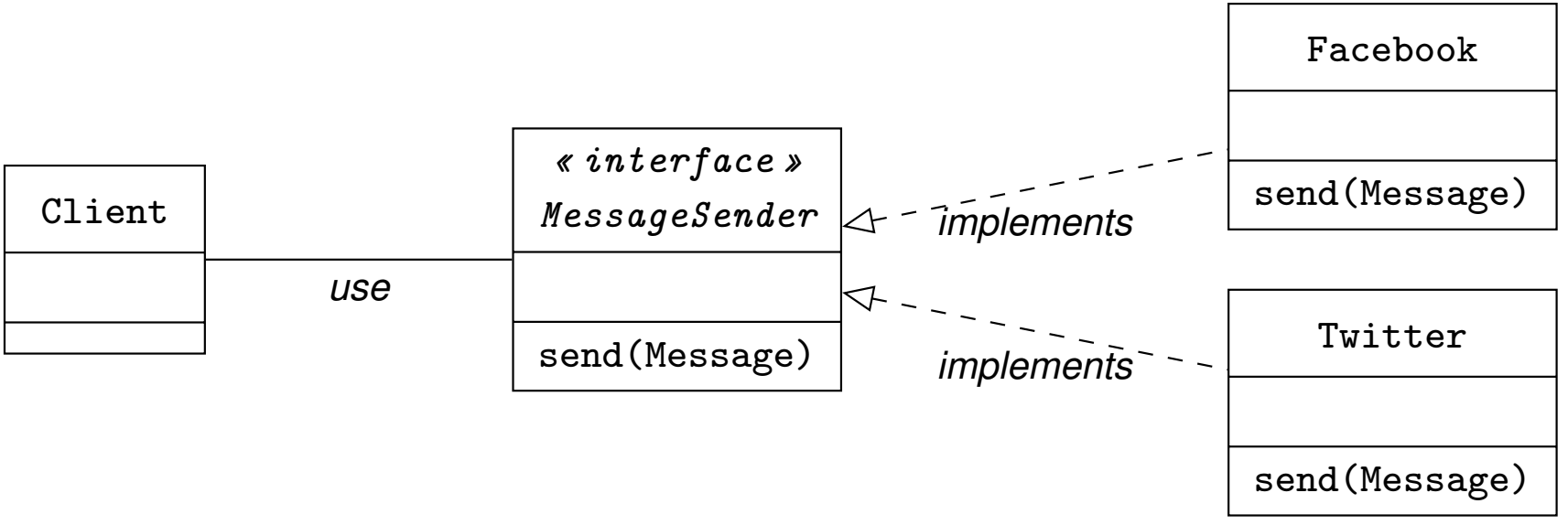
- Rappel : == teste l'égalité « d'adresse » sur les références.
- Rappel : Pour l'égalité logique, il faut utiliser la méthode `equals`.
- Tous les objets ont cette méthode.
- Attention, par défaut utilise ==, il faut donc la redéfinir dans vos classes.
- Mais attention à ne pas changer sa signature
- Bonne pratique : redéfinir de concert `hashcode` car deux objets `equals` doivent avoir le même hashcode

- `Client` utilise `Twitter` pour envoyer des messages (`send`) :



- Association à `Twitter` impose aux `Client` de n'être en lien qu'avec des sous-types de `Twitter`
- `Client` n'utilise que `send` (utiliser par exemple des instances de `Twitter`)

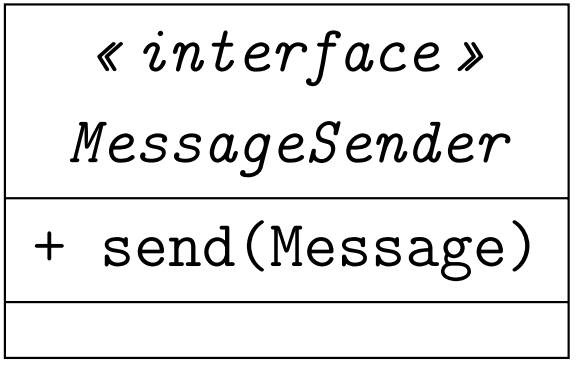
- On définit un contrat (l'*interface*)
- `Client` peut être en lien avec tout objet respectant le contrat (*réalisant* l'interface)



Définit un type "abstrait" et permet de lui associer plusieurs implémentations

```
1  public class Twitter implements MessageSender{
2    ...
3    public Twitter(String server) {
4      ...
5    }
6    public void send(Message m) {
7      ...
8    }
9    ...
10 }
```

```
1  public interface MessageSender {
2    public void send(Message m);
3  }
```



- C'est une liste de services $\Rightarrow$ un Type
- Une (sorte de) classe :
  - sans code (les méthodes n'ont pas de corps*)
  - sans état
- Une interface peut hériter (mot clé `extends`) de plusieurs interfaces
- Une classe peut réaliser (mot clé `implements`) plusieurs interfaces
- Une classe qui réalise une interface doit fournir toutes les méthodes de l'interface
- (*)Dans les dernières versions de Java les interfaces peuvent proposer des corps de méthode par défaut (`default`)

- On veut pouvoir appliquer des opérations sur des objets
- Les opérations doivent s'appliquer sur des classes différentes (`Twitter`, `Facebook`)
- Les classes n'ont pas la même classe mère, donc impossible d'utiliser l'héritage
- On veut être indépendant des fournisseurs...

C'est aussi une manière de séparer :

- la spécification (ici les signatures des méthodes) qui décrit le quoi
- des implémentations (les classes) qui décrivent le comment.

TypeApparent ?
TypeRéel ?

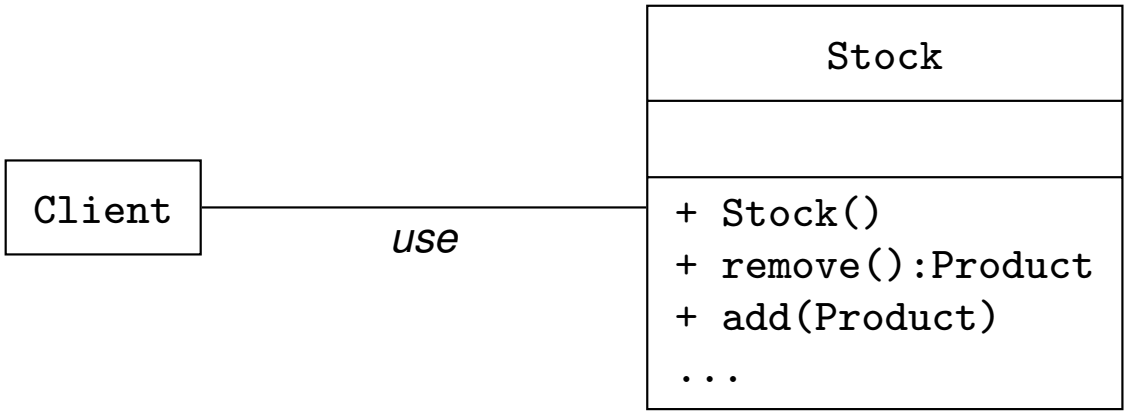
- On veut pouvoir appliquer des opérations sur des objets
- Les opérations doivent s'appliquer sur des classes différentes (`Twitter`, `Facebook`)
- Les classes n'ont pas la même classe mère, donc impossible d'utiliser l'héritage
- On veut être indépendant des fournisseurs...

C'est aussi une manière de séparer :

- la spécification (ici les signatures des méthodes) qui décrit le quoi
- des implémentations (les classes) qui décrivent le comment.

TypeApparent ? C'est le quoi !
TypeRéel ? C'est le comment !

- Prise en compte de tous les cas d'exécution :




```
Stock s = new Stock();
s.remove();
```

- Traitement du cas exceptionnel dépend du client

En Java, trois aspects pour gérer les cas exceptionnels :

1. Déclaration de l'exception
2. Détection de l'anomalie et levée de l'exception
3. Récupération et traitement de l'exception

Le compilateur s'assure que les exceptions sont bien gérées : attrapées ou renvoyées.

- Création d'une classe par cas exceptionnel :

```
public class EmptyStockException extends Exception { };
```

- Déclaration de l'exception dans la signature des méthodes avec `throws`
- Détection de l'anomalie et levée de l'exception avec `throw`

```
  public void remove () throws EmptyStockException {
    ...
    if (isEmpty()) throw new EmptyStockException();
    ...
}
```

- Dans le code appelant :

```
public void do() { // ne propage pas l'exception
  try {
    ...
    // action susceptible de lever une exception
    s.remove();
    ...
  } catch (EmptyStockException erreur){
        // traitement de l'exception
        ...
  }
}
```

- Soit on traite localement soit on propage l'exception

```
public void do() throws EmptyStockException {
    s.remove();
  }
}
```

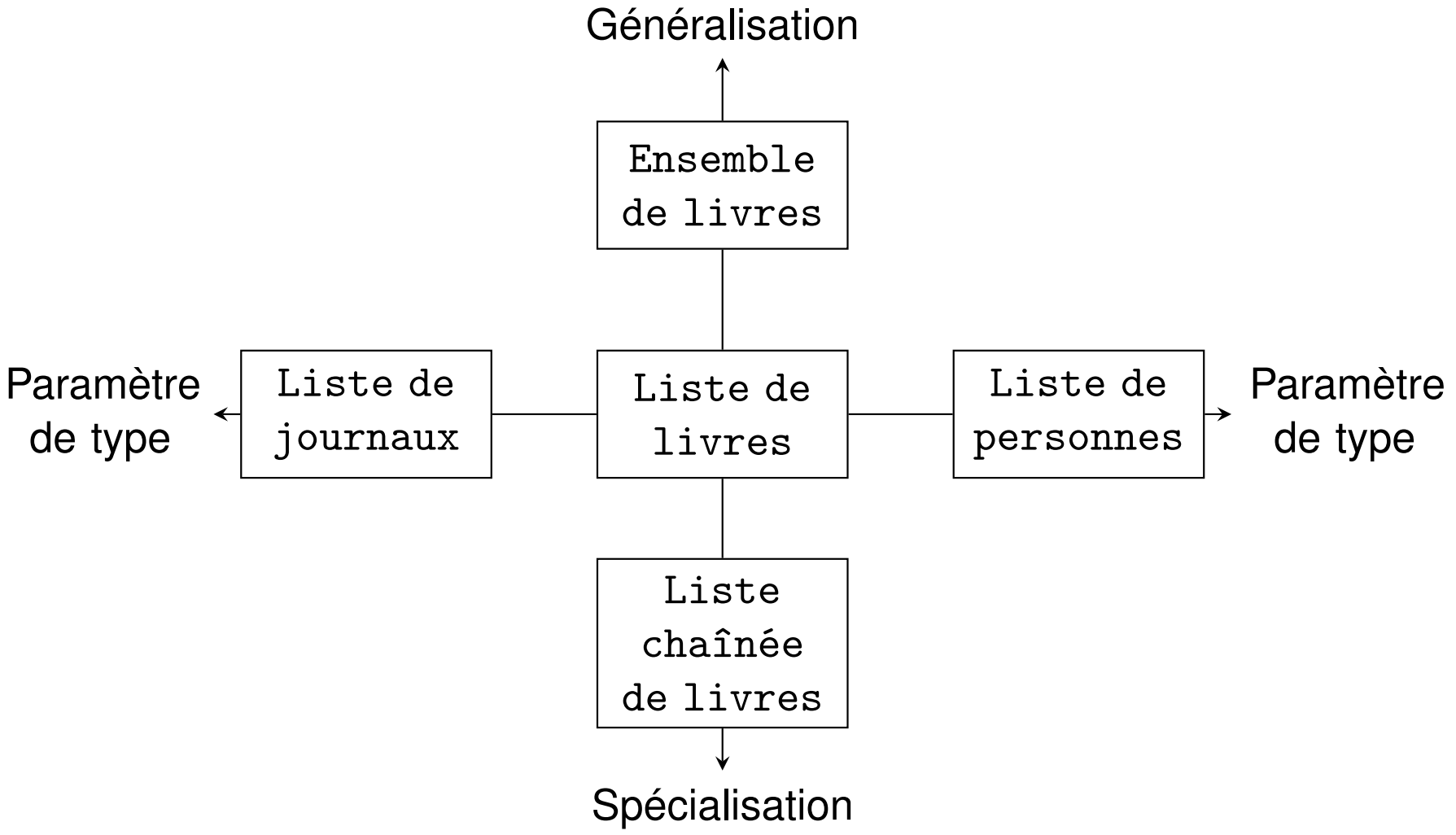
- Paramétrer une classe par un (ou plusieurs) type

```
1 class Stock<T> {
2   T[] content;
3   void add(T e) { ... }
4   T get(String name) { ... }
5 }
```

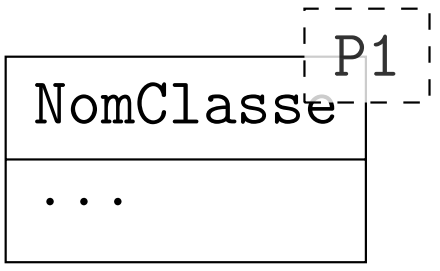
- La valeur réelle de `T` doit être connue avant l'instanciation

```
1 class Bibliotheque {
2   Stock<Media> leslivres;
3 }
```

- Ces valeurs réelles peuvent être différentes d'une instance à une autre


Généralisation
Ensemble de livres
Paramètre de type
Liste de journaux
Liste de livres
Liste de personnes
Paramètre de type
Liste chaînée de livres
Spécialisation

UML (diag. de classe)



Java

```
1  class NomClasse<P1> {
2    ...
3  }
```

- C'est une sorte de création de type
- Le principe syntaxique est voisin de celui d'un appel de fonction
- Quand ? Pour créer un nouveau type, on donne les paramètres réels : `LinkedList<Product>`
- Permet des collections typées

```
1 LinkedList<Product> le = new LinkedList<Product>();
2 le.add(new Product("banane");
3 ...
4 for (int i=0; i<le.size(); i++){
5   Product element = le.get(i);
6   ...
7 }
```

- Généricité simple et naturelle à utiliser
- Écriture de type générique difficile
  - les signatures ne sont pas faciles à écrire
  - il faut faire attention aux structures mutables

- Programmer objet favorise la réutilisation
- Il faut changer votre façon de programmer pour favoriser l'héritage, la redéfinition, la composition et la généricité
- Mais en échange les programmes sont plus difficiles à écrire (il faut un compromis)

Organisation

- Documenter
- Hiérarchiser

Conception

- Chercher la modularité
- Réutiliser

Programmation

- Séparer les responsabilités
- Réifier
- Héritage parcimonieux
- Déclarer abstrait, instancier concret
- Pas de fuite d'exceptions

- Réifier
- Héritage
- Transtyper
- Redéfinir
- Surcharger
- Propager
- Capturer

- type apparent
- type réel
- sous-type
- interface
- substituabilité
- redéfinition
- liaison dynamique