



IMT Atlantique

Bretagne-Pays de la Loire
École Mines-Télécom

Héritage

A. Beugnard, F. Dagnat &
J. Mallet

Cours – UE-IR-S6 – C1

1^{er} semestre 2025

- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface
- 7 Conclusion

- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface
- 7 Conclusion

Comment ajouter des propriétés à une classe ?

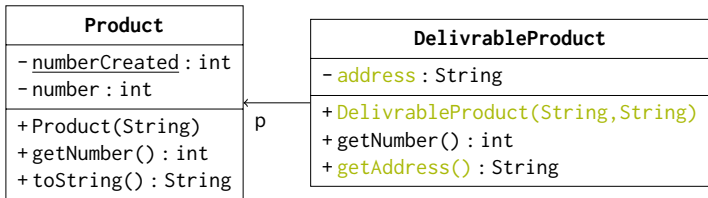
- ▶ un attribut : nom, couleur, délai, age, ...
- ▶ une fonction : `vieillir()`, `afficherCouleur()`, ...

Par exemple, on dispose de la classe `Product` et on souhaite ajouter une adresse de livraison.

Comment faire ?

1. On modifie le code source directement
 - ▶ Il faut posséder le code source.
 - ▶ Il faut mesurer les conséquences sur les autres objets qui utilisent cette classe.
 - ▶ Il faut faire de nouveaux tests.
2. On crée une nouvelle classe qui redirige vers la classe existante : *délégation*.
3. On crée une nouvelle classe qui étend la classe existante : *héritage*.

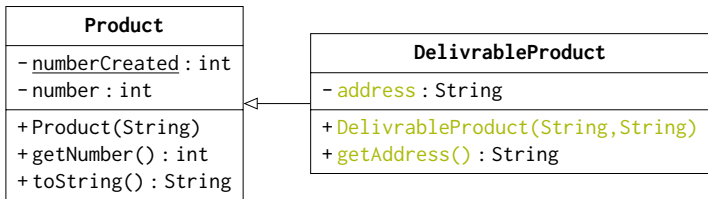
consiste à transférer la responsabilité d'un traitement d'un objet à un autre objet, permettant une composition flexible et une réutilisation du comportement



Déléguer, c'est rediriger vers...`p`.

```
1 public DeliverableProduct(String n,String a) {
2     p = new Product(n); // le délégué
3     address = a;
4 }
5 public int getNumber() {
6     return p.getNumber();
7 }
```

permet à une classe de dériver des propriétés et des comportements d'une autre classe, favorisant ainsi la réutilisation et l'évolutivité du code



Hériter, c'est écrire uniquement les propriétés supplémentaires.

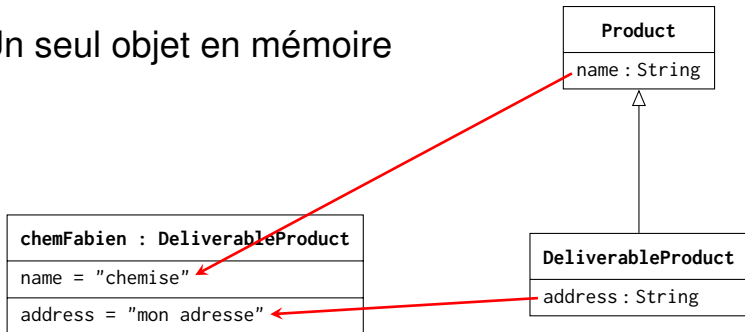
```
1 public DeliverableProduct(String n,String a) {
2     // pas de délégué
3     ...
4     address = a;
5 }
6 /** redirections inutiles */
```

- ▶ Changer le code source
 - ▶ pas toujours possible
 - ▶ pas structurant (toujours tout dans la même classe)
- ▶ Déléguer
 - ▶ on sépare bien, mais on écrit toute la mécanique à la main
 - ▶ si le code d'origine change, il faut aussi adapter
- ▶ Hériter
 - ▶ on sépare et la mécanique est automatique

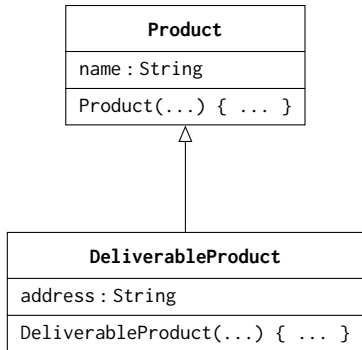
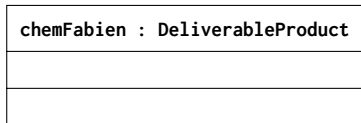
- 1 Héritage : principes
- 2 Héritage en java**
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface
- 7 Conclusion

```
1 public class DeliverableProduct extends Product {
2     private String address ; // un attribut ajouté
3     public DeliverableProduct(String name, String address) {
4         super(name);
5         this.address = address;
6     }
7     public String getAddress() { // méthode ajoutée
8         return this.address;
9     }
10    public String getName() { // méthode redéfinie
11        return super.getName() + " pour destination "
12            + this.address;
13    }
14 }
```

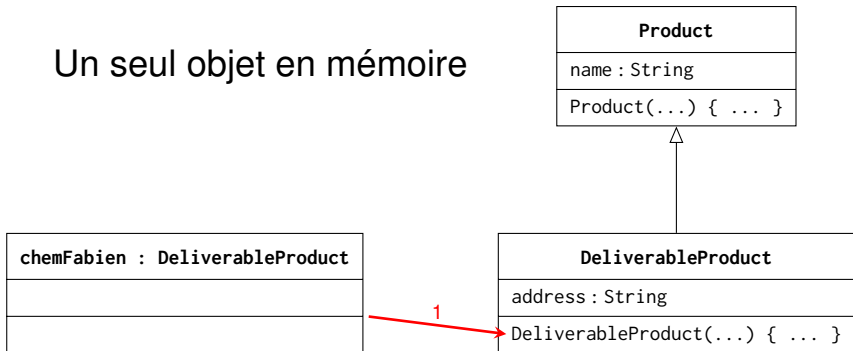
Un seul objet en mémoire



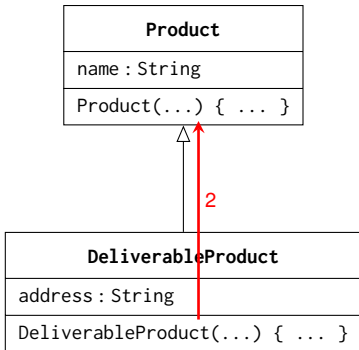
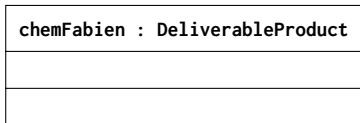
Un seul objet en mémoire



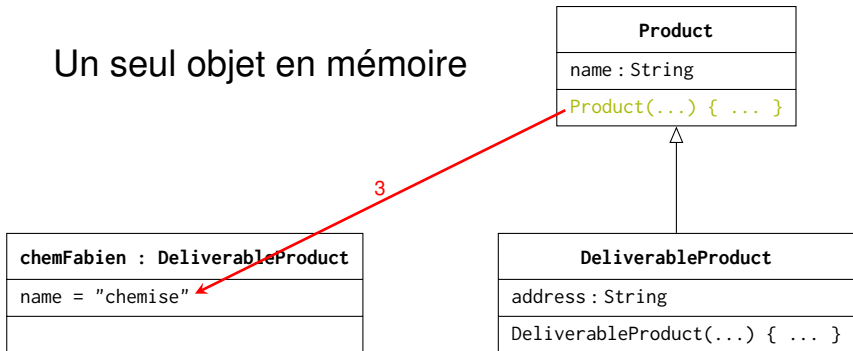
Un seul objet en mémoire



Un seul objet en mémoire

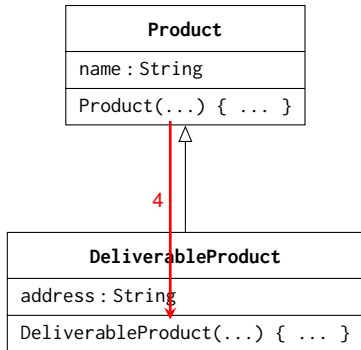


Un seul objet en mémoire

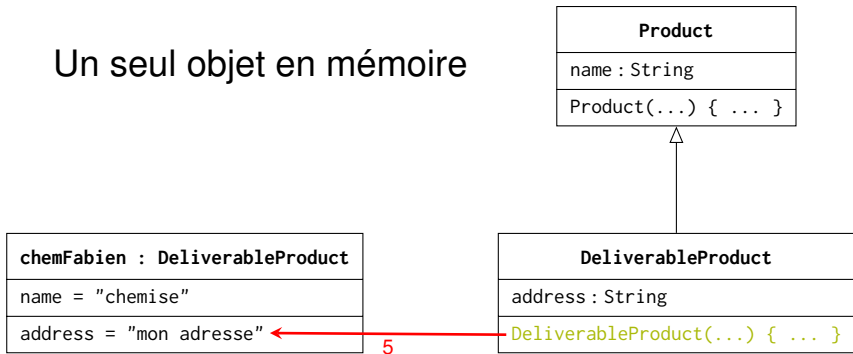


Un seul objet en mémoire

chemFabien : DeliverableProduct
name = "chemise"



Un seul objet en mémoire



- ▶ À un autre constructeur de la même classe

```
1 public BankAccount(String firstName, String lastName) {  
2     this(firstName, lastName, 0); ... }
```

- ▶ Explicite à un constructeur de la classe mère

```
1 public BankAccount(String firstName, String lastName) {  
2     super(firstName, lastName); ... }
```

- ▶ Implicite au constructeur par défaut de la classe mère (ajouté par le compilateur)

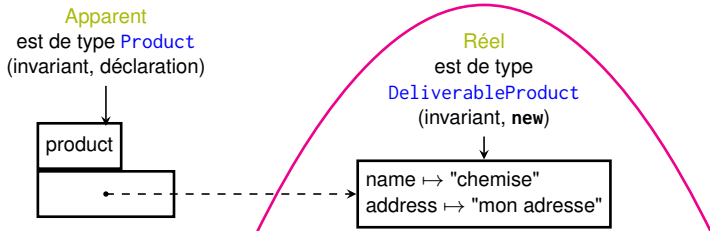
```
1 public BankAccount(String firstName, String lastName) {  
2     /* n'importe quoi sauf this ou super */ ... }
```

► Ajout du modifieur de visibilité **protected**

Java	accessible par
<code>private</code>	les objets de même classe
<code>rien</code>	les objets de classes du même paquetage
<code>protected</code>	les objets de classes héritant de cette classe ou les objets de classes du même paquetage
<code>public</code>	tous (pas d'encapsulation)

- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types**
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface
- 7 Conclusion

► `Product product = new DeliverableProduct("chemise", "mon adresse");`

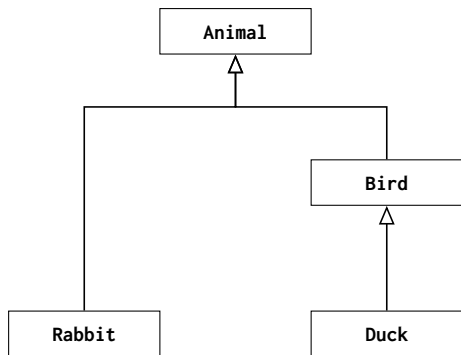


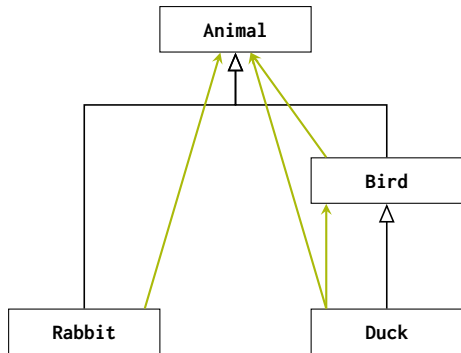
- Le type réel peut être un sous-type du type apparent
- Une expression Java aura aussi ces deux types

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

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

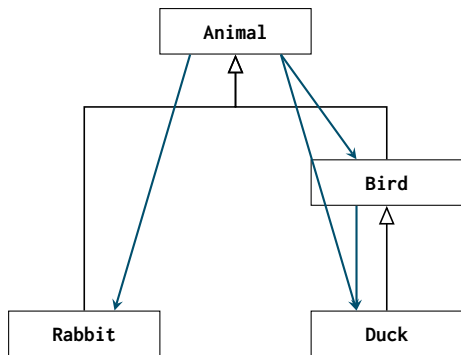
```
1 DeliverableProduct pl;  
2 if (product instanceof DeliverableProduct)  
3     pl = (DeliverableProduct) product;
```





— Automatique

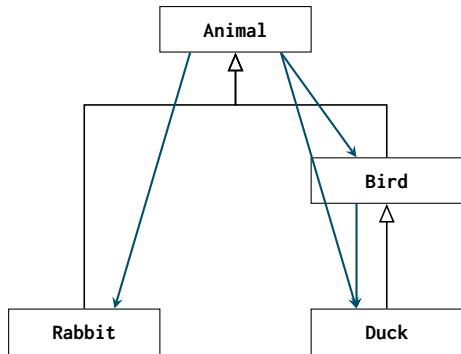
```
Animal a = new Rabbit();
```



— Conversion nécessaire (cast)

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

```
Rabbit l = (Rabbit) a;
```

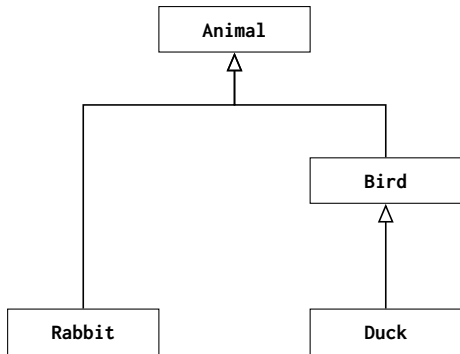


— Conversion nécessaire (*cast*)

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

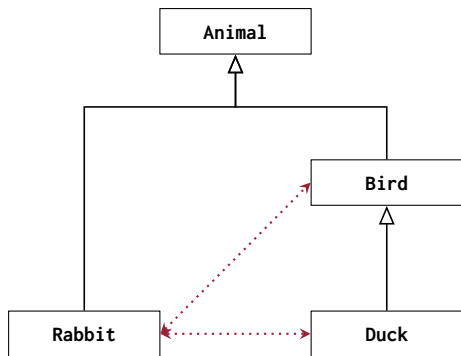
```
Rabbit l = (Rabbit) a;
```

```
Duck p = (Duck) a;
```



```
Animal a = new Rabbit();
Rabbit l = (Rabbit) a;
Duck p = (Duck) a;
```

Erreur à l'exécution : "class Rabbit cannot be cast to class Duck "

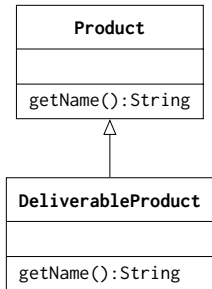


..... Erreur de compilation

```
Animal a = new Rabbit();
Rabbit l = (Rabbit) a;
Duck p = (Duck) a;
Duck d = new Rabbit();
```

Msg : "incompatible types : Rabbit cannot be converted to Duck"

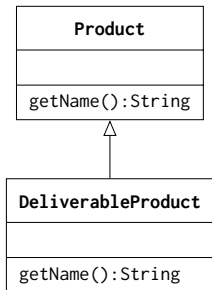
- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique**
- 5 La classe Object
- 6 Interface
- 7 Conclusion



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

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

```
Product p = new DeliverableProduct("courgette", "IMT Atlantique");
System.out.println(p.getName());
```



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

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

```
int i = ...;
Product p;
if (i==1) p = new DeliverableProduct("carotte","IMT Atlantique");
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.
- ▶ Les deux méthodes doivent :
 - ▶ avoir le même nom,
 - ▶ avoir la même signature,
 - ▶ 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** → **public**) mais pas la réduire (sous-typage).

- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object**
- 6 Interface
- 7 Conclusion

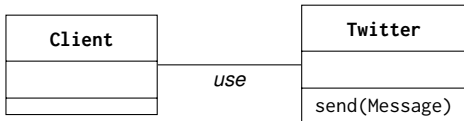
- ▶ En Java, une classe qui n'étend personne étend la classe `Object`
- ▶ Cette classe est la racine de l'arbre d'héritage
- ▶ En fait : toutes les références sont de type `Object` (même tableaux)
- ▶ Conséquence : toutes les méthodes de `Object` peuvent être utilisées sur les références
- ▶ De plus,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

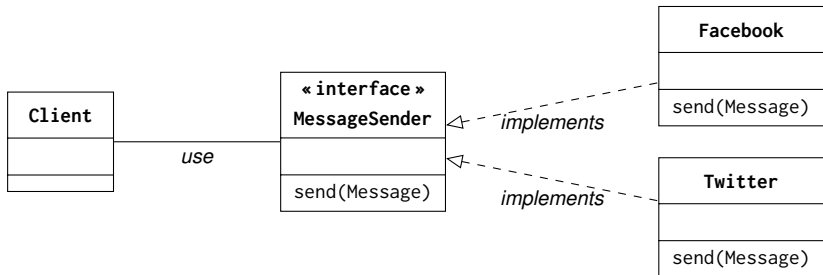
- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface**
- 7 Conclusion

- ▶ **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

- ▶ 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 (le contrat à respecter)
- ▶ 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 (le contrat à respecter)
- ▶ des implémentations (les classes) qui décrivent le comment.

TypeApparent ? C'est le quoi !

TypeRéel ? C'est le comment !

- 1 Héritage : principes
- 2 Héritage en java
- 3 Les types
- 4 La liaison dynamique
- 5 La classe Object
- 6 Interface
- 7 Conclusion**

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