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

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

Héritage

A. Beugnard, F. Dagnat & J. Mallet

Cours – UE-IR-S6 – C1– 1er semestre 2026

Plan

Avancement

Le défi de la réutilisation

Comment ajouter des propriétés à une classe ?

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

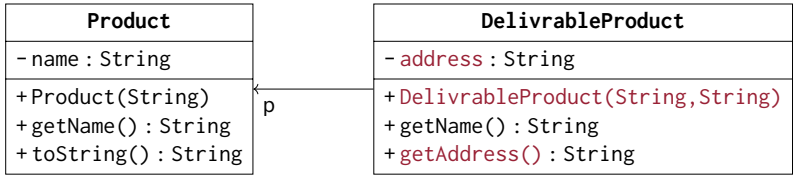
Par exemple, on dispose de la classe `Product` et on souhaite ajouter une adresse de livraison.
Comment faire ?

Trois solutions

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*.

Principe de la délégation

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

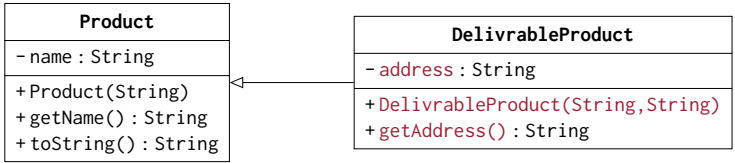


Déléguer, c'est rediriger vers. . .p.

```
public DelivrableProduct(String n,String a) {
  p = new Product(n); // le délégué
  address = a;
}
public String getName() {
  return p.getName();
}
```

Principe de l'héritage

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.

```
public DelivrableProduct(String n,String a) {
  // pas de délégué
  ...
  address = a;
}
/** redirections  inutiles */
```

Comparaison

- 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

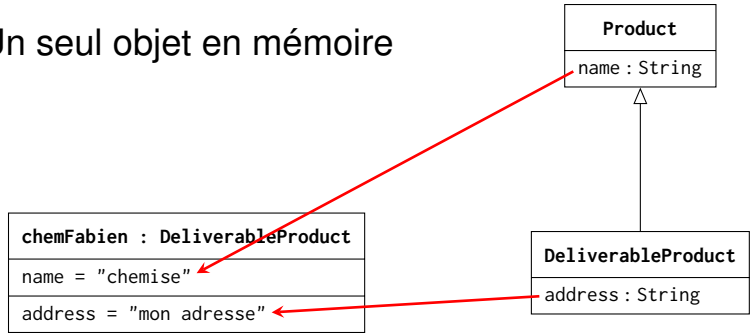
Avancement

L'héritage en Java

```java
public class DeliverableProduct extends Product {
  private 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;
  }
}
```

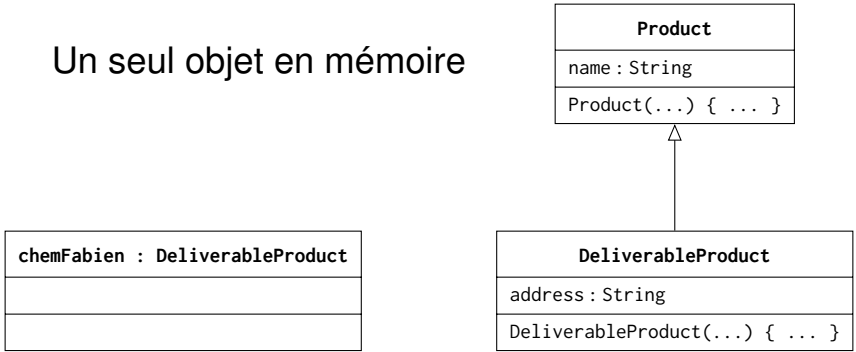
Objet et héritage

Un seul objet en mémoire



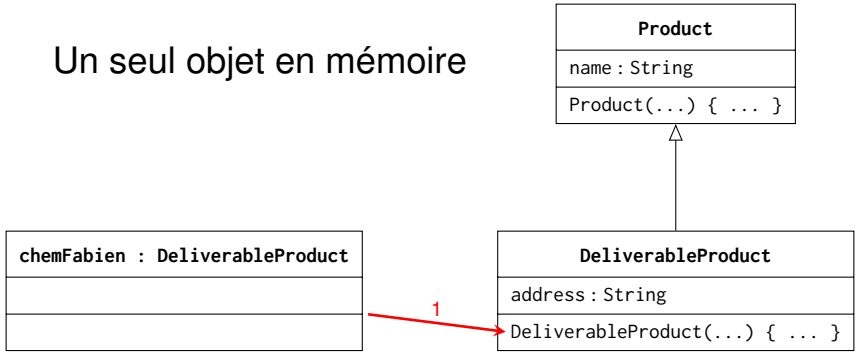
Constructeur et héritage

Un seul objet en mémoire



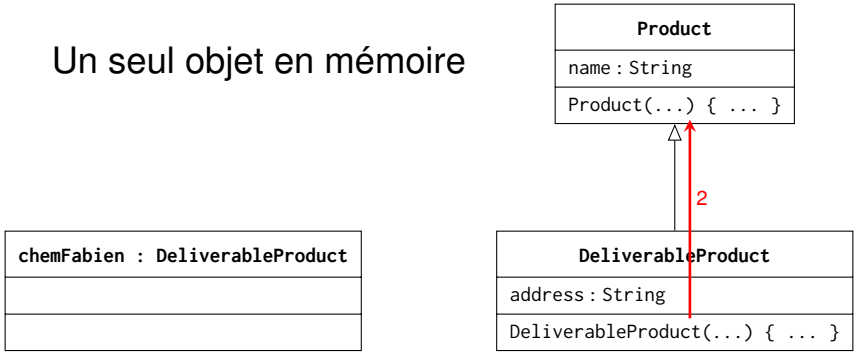
Constructeur et héritage

Un seul objet en mémoire



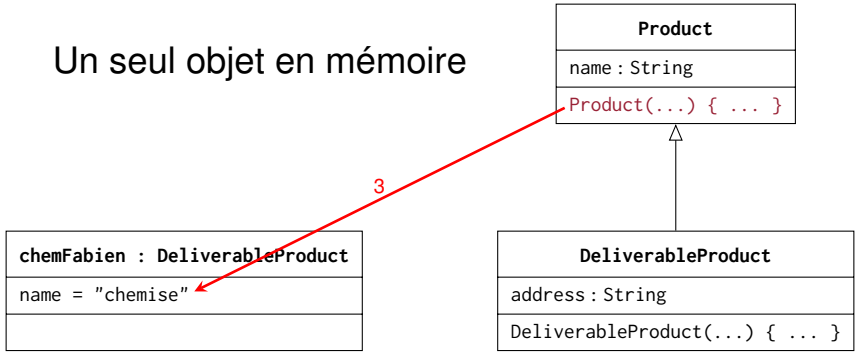
Constructeur et héritage

Un seul objet en mémoire



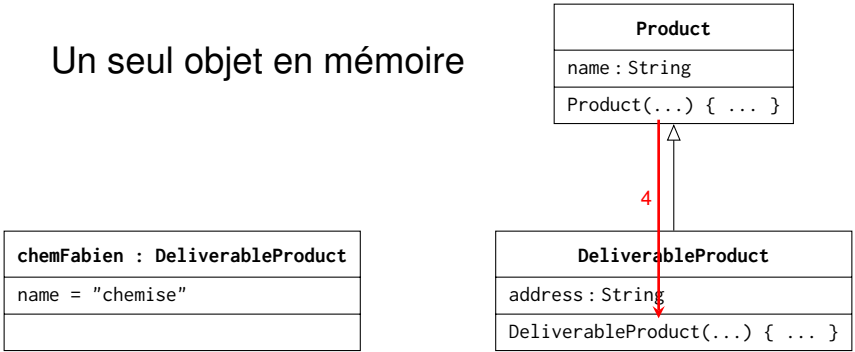
Constructeur et héritage

Un seul objet en mémoire



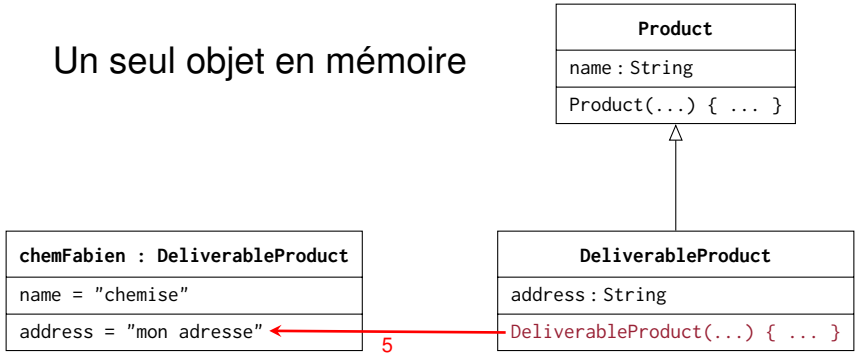
Constructeur et héritage

Un seul objet en mémoire



Constructeur et héritage

Un seul objet en mémoire



Un constructeur commence par un appel

- À un autre constructeur de la même classe

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

- Explicite à un constructeur de la classe mère

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

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

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

Dernier retour sur la visibilité

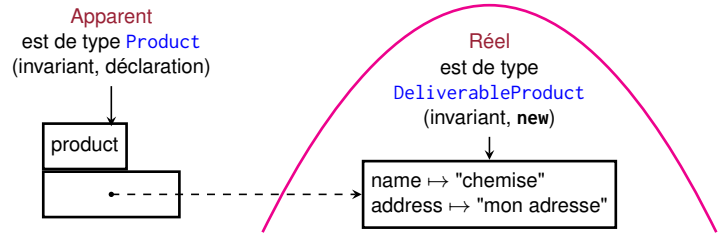
- Ajout du modifieur de visibilité **`protected`**

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

Avancement

Les deux types. . .

- `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

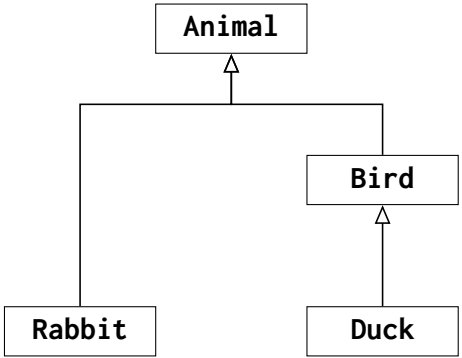
On utilise aussi le vocabulaire type statique (apparent) et dynamique (réel)

Transtypage – *casting*

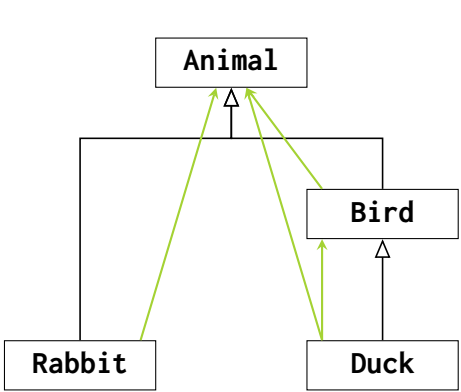
- Comment utiliser `product` comme un `DeliverableProduct` ?
- On transtype (*cast*) vers le type `DeliverableProduct`
  - `(DeliverableProduct) product`
- `ClassCastException` levée si l'objet pas du type demandé
- On peut tester si un objet est d'un type

```
DeliverableProduct pl;
if (product instanceof DeliverableProduct)
  pl = (DeliverableProduct) product;
```

Hiérarchie et Transtypage

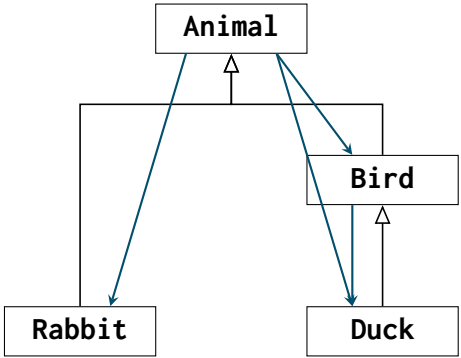


Hiérarchie et Transtypage




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

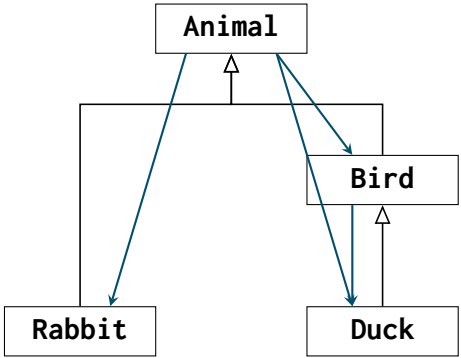
Hiérarchie et Transtypage



Conversion nécessaire (*cast*)

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

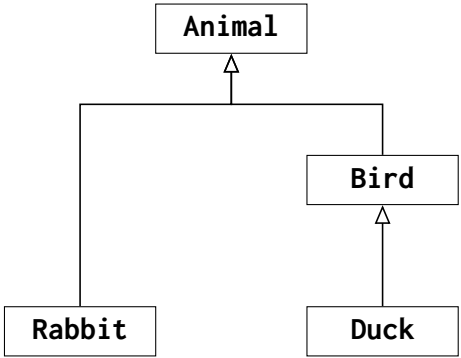
Hiérarchie et Transtypage



Conversion nécessaire (*cast*)

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

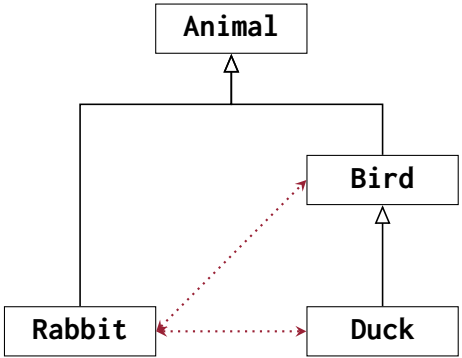
Hiérarchie et Transtypage




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

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

Hiérarchie et Transtypage



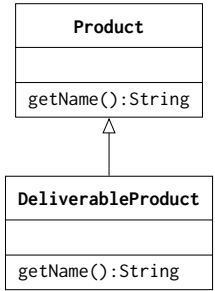
..... Erreur de compilation

```
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"

Avancement

Exemple I

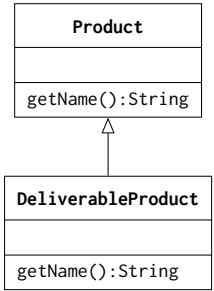



```
public class Product{
...
  public String getName(){return name;}
}

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

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

Exemple II




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

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

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

Recherche de la méthode

- 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

Redéfinition

- 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`** $\rightarrow$ **`public`**) mais pas la réduire (sous-typage).

Avancement

La classe `Object`

- 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

Un extrait du code de la classe `Object`

```
package java.lang;

public class Object {
  public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
  }
  public boolean equals(Object obj) { return (this == obj); }

  protected native Object clone() throws CloneNotSupportedException;

  public final native Class getClass();
  public native int hashCode();
  ...
}
```

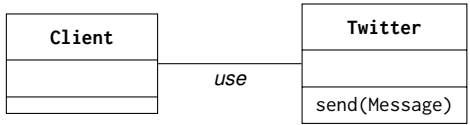
L'égalité le retour

- 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

Avancement

La problématique

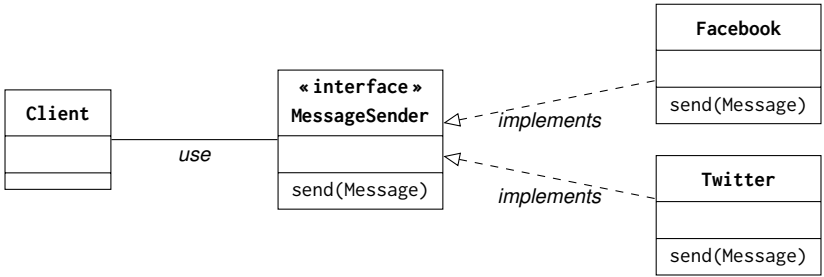
- `Client` utilise `Twitter` pour envoyer des messages (`send`) :



- 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`)

Une solution : l'interface

- 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

Classe Twitter et une interface MessageSender

```
public class Twitter implements MessageSender{
  ...
  public Twitter(String server) {
    ...
  }
  public void send(Message m) {
    ...
  }
  ...
}

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

Qu'est-ce qu'une interface ?

- 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

Quand utiliser une 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.

Type apparent ?
Type réel ?

Quand utiliser une 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.

Type apparent ? C'est le quoi !
Type réel ? C'est le comment !

Avancement

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)