M1 info

GINF41B2 (Conception et Programmation Orientée Objet)

Cours #3
Classes, délégation et héritage

(approfondissements techniques)

Pierre Tchounikine

Plan

- Lien modélisation / implantation (UML / Java)
- Héritage et typage
- Quelques détails techniques

Modélisation / Implantation
UML / Java

UML et les générateurs de code

- UML
  - ✗ pour penser, avant de rentrer dans le code
  - ✗ pour dénoter différentes choses à différents niveaux compréhensibles par différentes personnes
  - ✗ pour programmer (génération de code) via des représentations graphiques de plus en plus précises (modèles dynamiques)

- Les générateurs de code
  - ✗ permet un gain de temps : génération des squelettes
  - ✗ aide au maintien de la cohérence
  - ✗ va dans le sens d'une homogénéisation des codes et de la documentation (→ réutilisation, échanges, inter-opérabilité, etc.)

c'est le sens de l'histoire de la programmation
c'est poussé par le mouvement MDA (Model Driven Architecture)

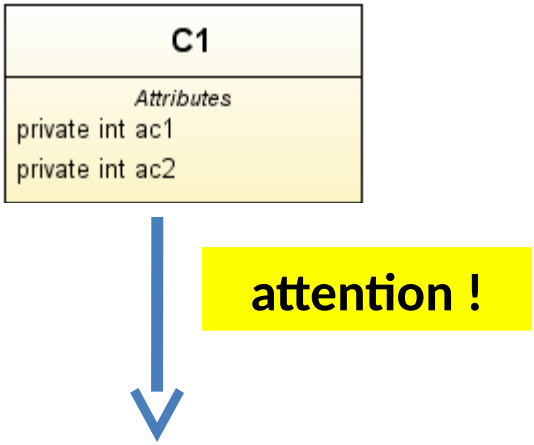
Éléments pris en revue

- Classe
- Association
- Dépendance
- Agrégation / composition
- Héritage

sans entrer dans un cours UML
→ idées générales, à approfondir !

(variations selon logiciels et/ou paramétrage)

Classe

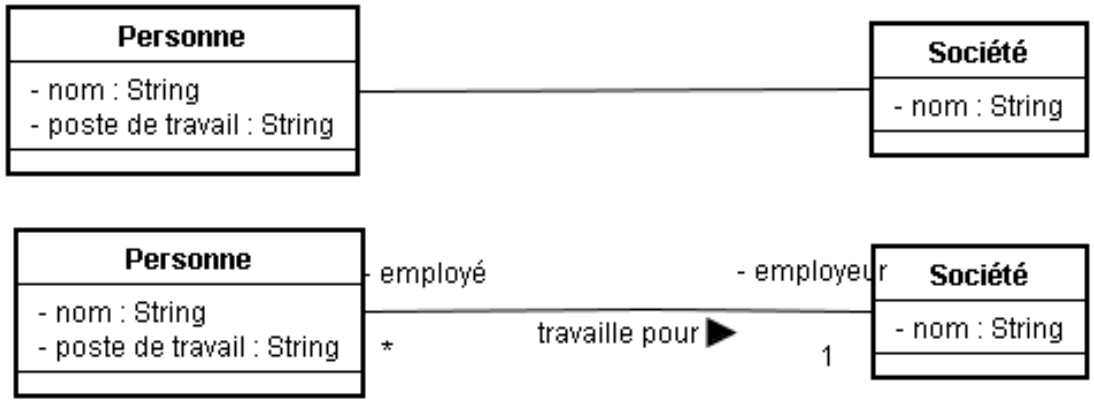



```
public class C1 {
    private int ac1;
    private int ac2;

    public C1 () {  }

    public int getAc1 () {    }
    public void setAc1 (int val) {    }
    public int getAc2 () {    }
    public void setAc2 (int val) {    }
}
```

Association entre 2 classes

cas des relations binaires (on passe ici sur les relations ternaires et les « classes-association »)

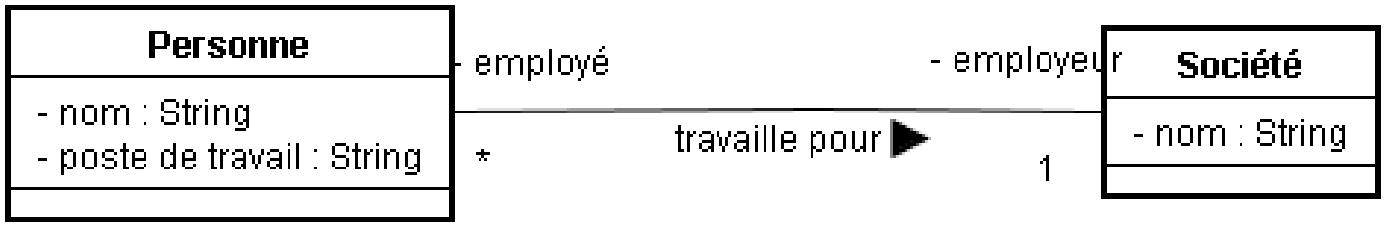
- En termes de modélisation
  - ✗ il existe une relation structurelle entre les 2 classes (entre les instances des 2 classes), elles sont liées (de façon « permanente »)



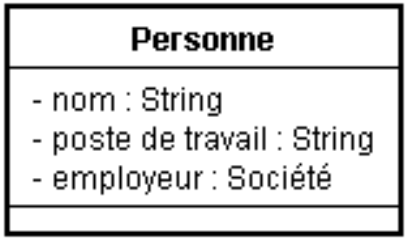
- En termes d'implantation
  - ✗ une des classes sert de **type** à un **attribut** de l'autre
  - ✗ il peut y avoir un/des **envoi(s) de message(s)** de l'une vers l'autre

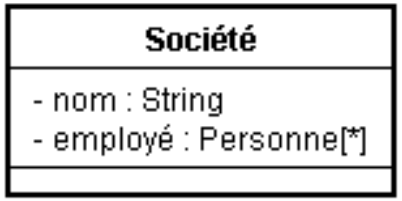
Association entre 2 classes

- Représentation UML (association = entité)



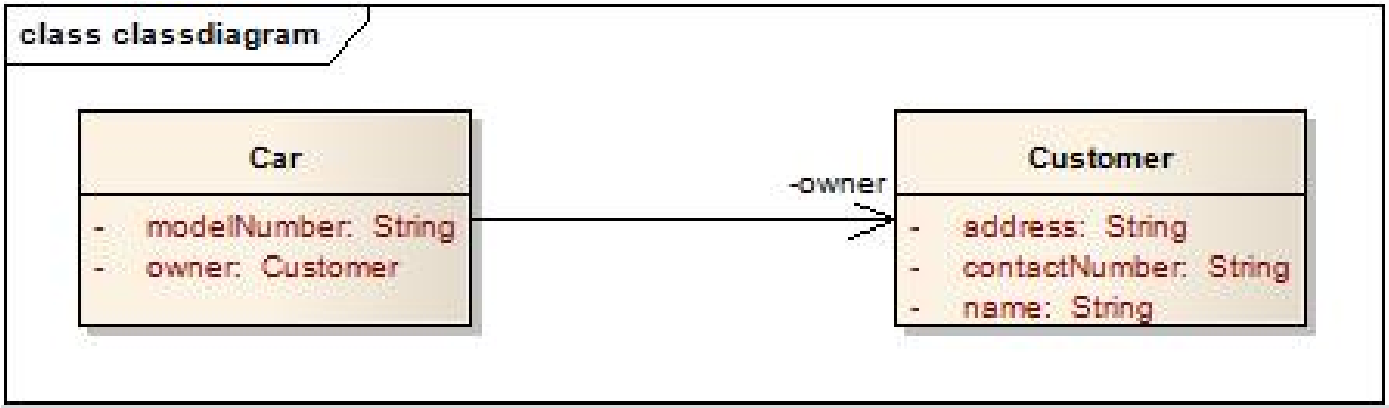
- Représentation via les attributs (en termes d'implantation)





Association entre 2 classes

association unidirectionnelle




```java
public class Customer {
private String name;
private String address;
private String contactNumber;
}

public class Car {
    private String modelNumber;
    private Customer owner;
}
```

Association entre 2 classes

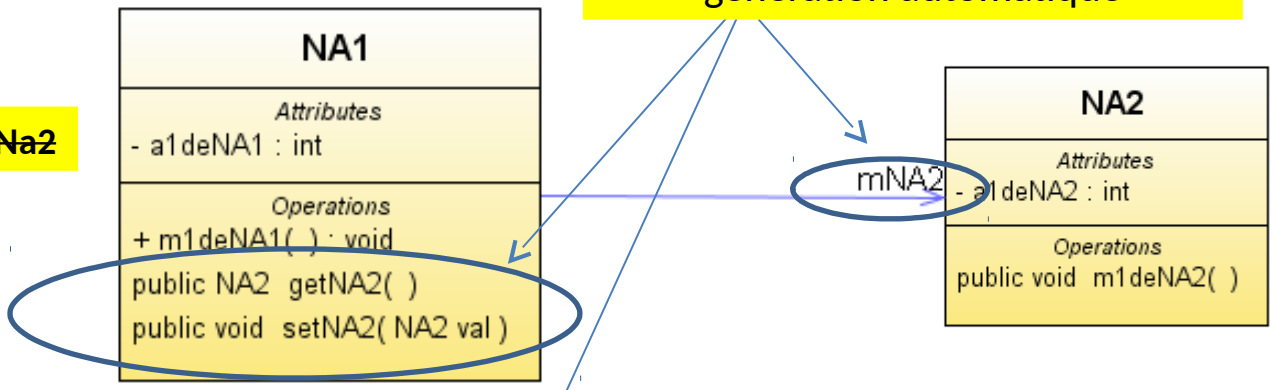
association bidirectionnelle


```java
public class Customer {
private String name;
private String address;
private String contactNumber;
private Car car;
}

public class Car {
    private String modelNumber;
    private Customer owner;
}
```

Association entre 2 classes

génération automatique

~~mNa2~~

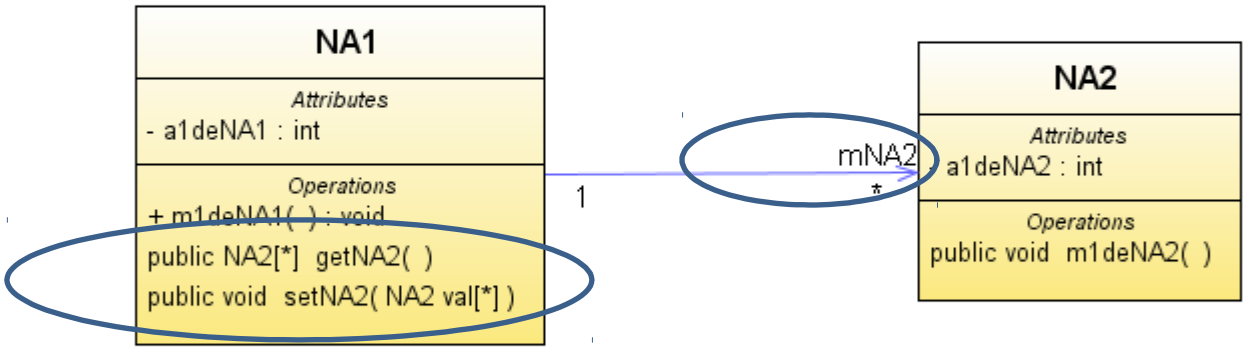



```
public class NA1 {
private NA2 mNA2;
private int a1deNA1;
public void m1deNA1 () {}
public NA2 getNA2 () {
        return mNA2;
    }
public void setNA2 (NA2 val) {
        this.mNA2 = val;
    }
}
```

```
public class NA2 {
private int a1deNA2;
public void m1deNA2 () { }
}
```

association unidirectionnelle de NA1 vers NA2 : NA2 ne connait pas l'existence de NA1 (notion de **navigabilité**)

Association avec multiplicité




```
import java.util.ArrayList;
public class NA1 {
private ArrayList<NA2> mNA2;
private int a1deNA1;
public void m1deNA1 () {}
public ArrayList<NA2> getNA2 () {
        return mNA2;     }
public void setNA2 (ArrayList<NA2>
val) {
        this.mNA2 = val;     }
}
```

(liste extensible)

```
public class NA2 {
private int a1deNA2;
public void m1deNA2 () { }
}
```

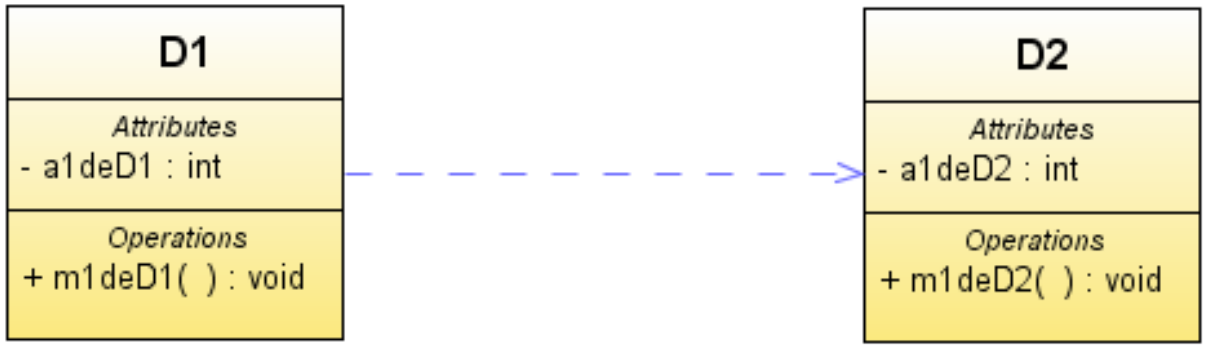
Dépendance entre 2 classes

- En termes de modélisation
  - ✗ association affaiblie : une modification de l'un peut entrainer une modification de celui qui en dépend ; elles sont liées mais de façon ponctuelle

- En termes d'implantation
  - ✗ la classe qui utilise peut recevoir comme paramètre un objet de la classe utilisée pour une méthode donnée
  - ✗ la classe qui utilise peut créer localement, le temps d'une méthode, un objet de la classe utilisée (qui va disparaître à la fin de la méthode)
  - ✗ si l'interface de la classe utilisée change, il y aura des répercussions sur la classe utilisatrice

le lien n'existe que le temps de l'exécution de la méthode, ce n'est pas une relation structurelle

Dépendance



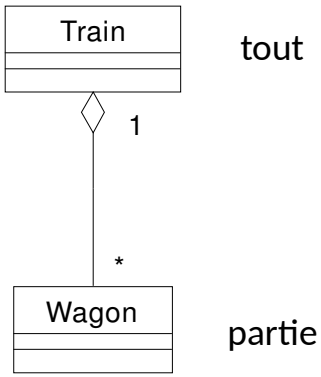

```
public class D1 {
private int a1deD1;
public void m1deD1 () {
}
```

```
public class D2 {
private int a1deD2;
public void m1deD2 () { }
}
```

Agrégation et composition

- En termes de modélisation
  - ✗ association particulière de « tout » à « partie »

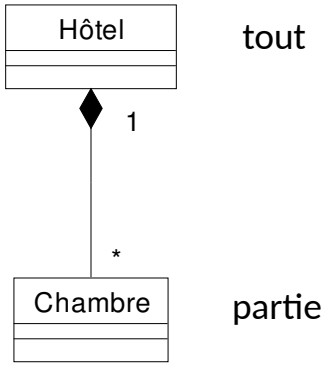
agrégation



relation, purement conceptuelle

les durées de vies des objets ne sont pas liées : on peut « détruire » un train, les wagons existeront toujours par ailleurs (la « partie » survit à la destruction du « tout »)

composition



agrégation « forte » (« par valeur »)

les durées de vies des objets sont liées : si je détruis l'hôtel je détruis les chambres avec

Agrégation et composition

- Idée clé et claire de l'agrégation et de la composition
  - ✗ dénoter la relation de tout à partie (/ association de base où tous les éléments sont au même niveau)

- Problèmes et difficultés
  - ✗ la différence est un point de vue de modélisation → lié à l'intention
  - ✗ la différence peut être « peu claire » (…)

→ pas de consensus sur l'utilisation de ces 2 relations

- ✓ les wagons peuvent exister indépendamment du train
- ✓ une chambre d'hôtel n'existe pas s'il n'y a pas d'hôtel
- ✓ une case d'échiquier n'existe pas sans échiquier

- ✓ relation entre « un téléphone portable » et « une coque de téléphone portable » ?

CRITERE
« les durées de vies des objets sont liées »

Agrégation et composition

- En termes de modélisation

  (/ association de base où tous les éléments sont au même niveau)

  - ✗ association particulière de « tout » à « partie »

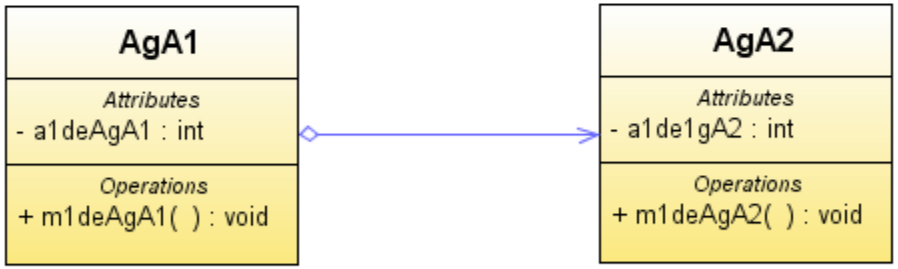
- En termes d'implantation

  - ✗ différentes approches selon l'accent mis sur cette notion de « durée de vie liée »

✔ NetBeans

✔ Code

Agrégation

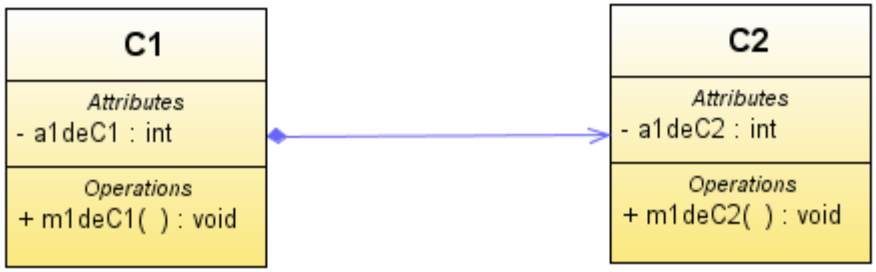


même chose
qu'une association

```
public class AgA1 {
private int a1deAgA1;
private AgA2 mAgA2;
public void m1deAgA1 () {}
}
```

```
public class AgA2 {
private int a1de1gA2;
public void m1deAgA2 () {}
}
```

Composition

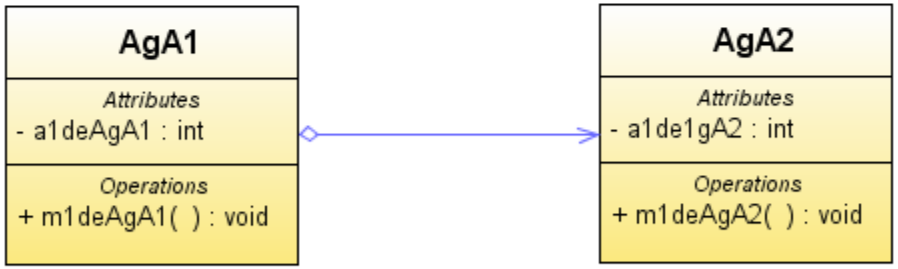


même chose
qu'une association

```
public class C1 {
private int a1deC1;
private C2 mC2;
public void m1deC1 () {}
}
```

```
public class C2 {
private int a1deC2;
public void m1deC2 () {}
}
```

Agrégation avec multiplicité



même chose
qu'une association

```
import java.util.ArrayList;
public class AgA1 {
private int a1deAgA1;
private ArrayList<AgA2> mAgA2;
public void m1deAgA1 () {}
}
```

```
public class AgA2 {
private int a1de1gA2;
public void m1deAgA2 () {}
}
```

Agrégation / Composition

différenciation « une référence » / « mon objet »

- Agrégation : un objet O1 fait référence à un autre objet O2

→ O2 existe par ailleurs : les durées de vie de O1 et O2 ne sont pas liées

- Composition : un objet O1 contient un autre objet O2

→ O2 existe comme un champ de O1 (et disparaît avec lui)

Agrégation / Composition

différenciation « une référence » / « mon objet »

```
public class O {
   private String nom;
   private int val;

   public O(String nom, int val){
       this.nom=nom;
       this.val=val;    }

   public void afficheToi(){}

   public void incrémenteToi(){}
}
```

un O a un nom et une val

un O sait s'afficher

un O sait s'incrémenter (val++)

Agrégation / Composition

différenciation « une référence » / « mon objet »

```
public class C1 {
   private String nom;
   private O monO;

   C1(String nom, O o)  {
       this.nom=nom;
       this.monO=o;
    }

   public void incrémenteTonO(){
       monO.incrémenteToi();
    }
   public void afficheToi(){
       System.out.println ("Nom"+nom);
       monO.afficheToi();
   }
}
```

un C1 a un nom et un 0

on construit un C1 en lui passant un nom et un O

un C1 sait incrémenter son O

un C1 sait s'afficher, i.e., afficher son nom et son O

Agrégation / Composition

différenciation « une référence » / « mon objet »

```
public class C2 {
   private String nom;
   private O monO;

   C2(String nom, String nomO, int val) {
       O oLocal;
       oLocal = new O(nomO,val);
       this.nom=nom;
       this.monO=oLocal;
   }
   public void incrémenteTonO(){
       monO.incrémenteToi();
   }
   public void afficheToi(){
       System.out.println ("Nom : " + nom ");
       monO.afficheToi();
   }
}
```

un C2 a un nom et un O

on construit un C2 lui passant un nom, un nom et une valeur pour son O et en créant un O local

un C2 sait incrémenter son O

un C2 sait s'afficher, i.e., afficher son nom et son O

Agrégation / Composition

différenciation « une référence » / « mon objet »

```
O obj1 = new O("obj1",10);
obj1.afficheToi();
C1 obj1C1 = new C1("obj1C1", obj1);
obj1C1.afficheToi();
C1 obj2C1 = new C1("obj2C1", obj1);
obj2C1.afficheToi();
C2 obj1C2 = new C2 ("obj1C2","Obj-local-a-obj1C2",20);
obj1C2.afficheToi();
obj1C1.incrémenteTonO();
obj1C1.afficheToi();
obj2C1.incrémenteTonO();
obj2C1.afficheToi();
obj1C1.afficheToi();
obj1C2.incrémenteTonO();
obj1C2.afficheToi();
```

obj1 (valeur = 10)

obj1C1 dont le O est obj1 (valeur = 10)

obj1C1 et obj2C1 ont le même O

obj2C1 dont le O est obj1 (valeur = 10)

obj1C2 dont le O est Obj-local-a-obj1C2 (valeur = 20)

obj1C1 dont le O est obj1 (valeur = 11)

2 accès (donc, 2 incrémentations) du même objet

obj2C1 dont le O est obj1 (valeur = 12)

obj1C1 dont le O est obj1 (valeur = 12)

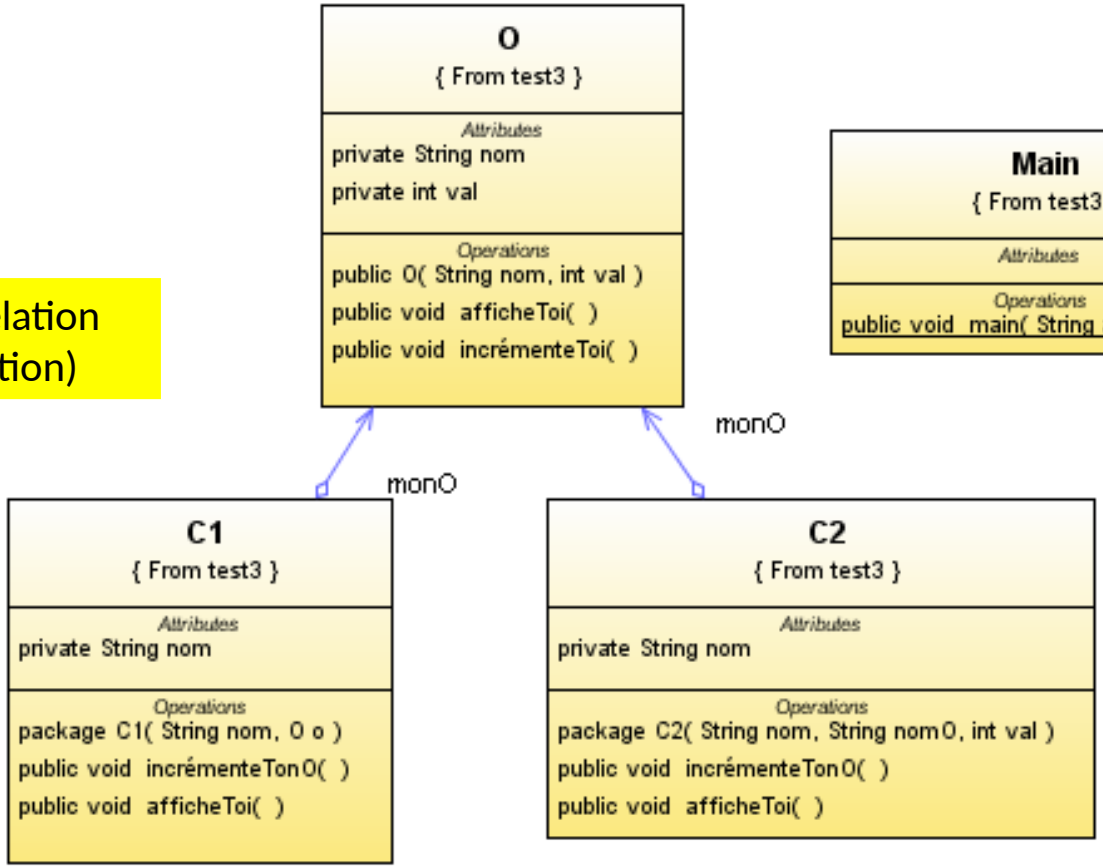
obj1C2 dont le O est Obj-local-a-obj1C2 (valeur = 21)

Agrégation / Composition

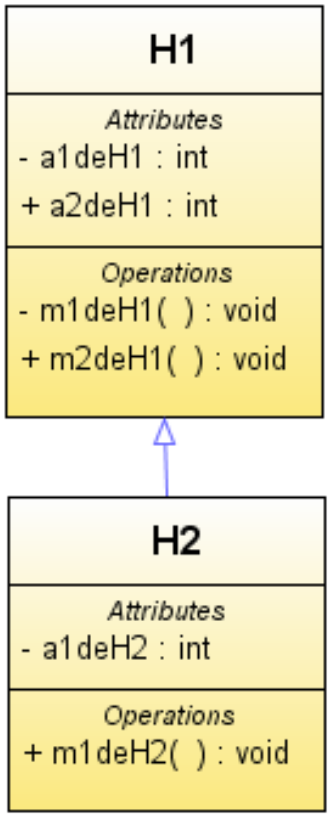
différenciation « une référence » / « mon objet »

reverse engineering

même relation (agrégation)



Héritage

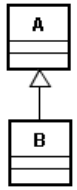



```
public class H1 {
public int a2deH1;
private int a1deH1;
public void m2deH1 () {     }
private void m1deH1 () {     }
}
```

```
public class H2 extends H1 {
    private int a1deH2;
    public void m1deH2 () { }
}
```

Héritage et typage

Héritage : différents points de vue



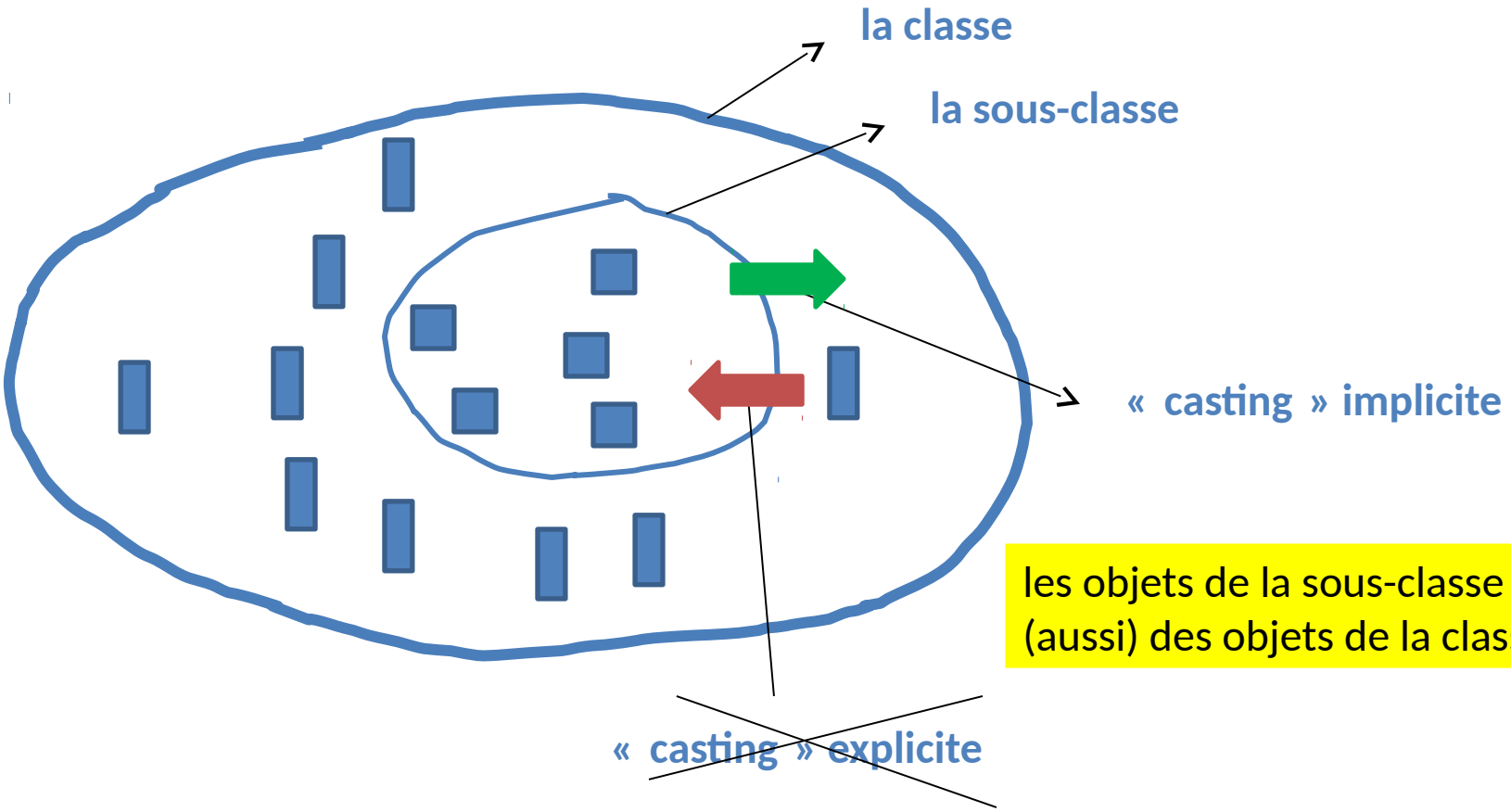
B hérite de A

taxonomie de B. Meyer, approfondi plus tard

Point de vue	Interprétation	Notion sous jacente
Conceptuel	B est plus spécifique que A, B est un A plus spécifique	Classification
Ensembliste	**Les éléments de B sont aussi des A, $B \subset A$**	**Type**
Logique	Si i est un B, i est un A	Polymorphisme
Comportemental	Les méthodes applicables aux A le sont aussi aux B, les B savent faire ce que savent faire les A	Classification
Structurel	Les attributs de A font partie des attributs de B, les instances de B ont les attributs de A	Classification

Héritage : vision ensembliste

- Les instances de la sous-classe forment un sous-ensemble des instances de la super-classe



les objets de la sous-classe sont (aussi) des objets de la classe

Héritage : type et structure

- Une classe est un type → une sous-classe est un type
- Héritage = ajout
  - ✗ une classe plus spécifique ne peut que rajouter des propriétés à sa super-classe
  - ✗ si une classe a (attributs) ou fait (méthodes), ses sous-classes ont et font

une sous-classe fait la même chose ou plus, mais pas moins !

(mais elle peut faire la même chose … « différemment »)

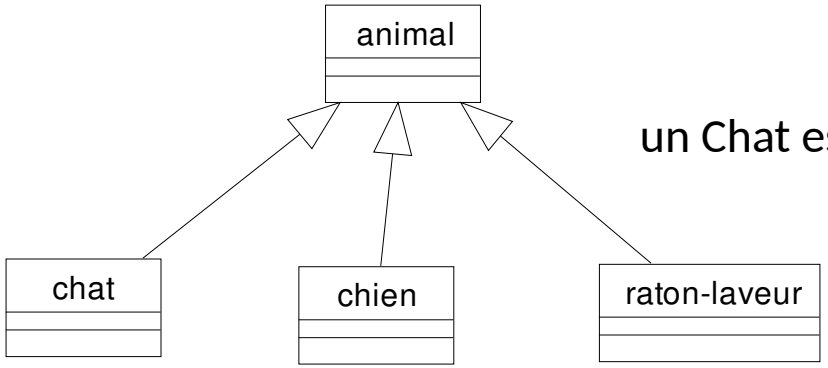
L'héritage est à la fois

B. Meyer

- ✗ une spécialisation du point de vue du type — **relation « est-un »**
- ✗ une extension du point de vue du module — **le nombre de services est augmenté**

Principe de substitution

- Principe de substitution : on peut « remplacer » un objet de la classe mère par un objet de la classe fille



un Chat est un Animal + des éléments supplémentaires
(et/ou qui a des particularités)

ce que sait faire un animal, un chat sait le faire
(en revanche, un animal ne sait pas faire tout ce que sait faire un chat)

la relation « est-un » n'est pas bijective

Héritage et typage

```
class Projet {
String nom;
Personne responsable;
void afficheToi(){System.out.println(nom);}   }

class Personne {
String nom;
void afficheToi(){System.out.println(nom);}   }

class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "   " + salaire);}    }
```

Principe de substitution : on peut « remplacer » un objet de la classe mère par un objet de la classe fille

Personne p;
Employe e;
p=e; // légal ? → **oui, un employé est une personne**
e=p; // légal ? → **non, une personne n'est pas un employé !**

Héritage et typage

```
class Projet {
String nom;
Personne responsable;
void afficheToi(){System.out.println(nom);}   }

class Personne {
String nom;
void afficheToi(){System.out.println(nom);}   }

class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "   " + salaire);}    }
```

Principe de substitution : on peut « remplacer » un objet de la classe mère par un objet de la classe fille

Personne a=new Employe(« titi », 3); // légal ? ⟶ **oui, un employé est une personne**
Employe b=new Personne(); // légal ? ⟶ **non, une personne n'est pas un employé !**

(instance « a » créée comme un Employé)

➡ schéma syntaxique <classe> <idf> = New <classe>

Héritage et typage

```
class Projet {
String nom;
Personne responsable;
void afficheToi(){System.out.println(nom);}   }

class Personne {
String nom;
void afficheToi(){System.out.println(nom);}   }

class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "   " + salaire);}    }
```

Personne p=new Personne(); p.nom="toto";
Projet proj=new Projet(); proj.responsable=p; //vilain, il faudrait un set
proj.responsable.afficheToi(); ⟶ **toto**
p=new Employe("titi",3);
proj.responsable.afficheToi(); ⟶ **toto** (on a créé un nouvel objet, proj n'est pas concerné)
p.afficheToi(); ⟶ **titi 3** (p est un Employé)

Héritage et typage

```
class Projet {
String nom;
Personne responsable;
void afficheToi(){System.out.println(nom);}   }

class Personne {
String nom;
void afficheToi(){System.out.println(nom);}   }

class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "   " + salaire);}    }
```

Personne p = new Employe("toto",4);
p.salaire=3; // légal ? ⟶ **illégal**
p.afficheToi(); ⟶ **toto 4** (déclenche le afficheToi de Employé = type réel)
((Employe)p).salaire=8; // légal ? ⟶ **légal**
p.afficheToi(); ⟶ **toto 8**
((Personne)p).afficheToi(); ⟶ **toto 8** (pas d'accès au afficheToi de la surclasse)

Héritage et typage

```
class Projet {
String nom;
Personne responsable;
void afficheToi(){System.out.println(nom);}   }

class Personne {
String nom;
void afficheToi(){System.out.println(nom);}   }

class Employe extends Personne {
int salaire;
Employe(String nom, int salaire) {this.nom=nom;this.salaire=salaire;}
void afficheToi(){System.out.println(nom + "   " + salaire);}    }
```

```
Personne p=new Personne();
p.nom="tutu";
Employe e=new Employe("titi",3);
//p=e;  licite
//e=p;  illicite
e=(Employe)p;   // légal?
```

Compilation : OK

Exécution : Exception in thread "main"
java.lang.ClassCastException:
Personne cannot be cast to
Employe

(ça aurait pu être un Employé … mais c'est une Personne)

Héritage et typage

- Un objet peut être de plusieurs types
- Le typage a une dimension dynamique

il y a 2 types !

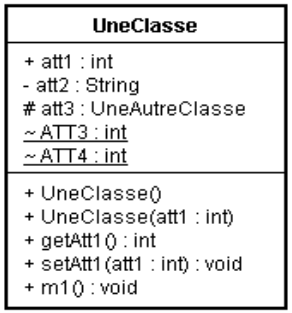
  - ✗ le type à travers lequel on voit l'objet à un moment donné, associé à **la référence** à l'objet qui est utilisée
  - ✗ le type réel de l'objet, celui avec lequel il a été **créé**

- Difficulté
  - ✗ faire attention à la compatibilité des types que définit l'héritage
- Avantage
  - ✗ possibilité de manipuler des objets « différents » comme des objets de même type (à un certain niveau d'abstraction)

polymorphisme et utilisation du principe de « liaison dynamique »

Quelques éléments techniques

Terminologie



- membres
- champs
- variable d'instance
- variable de classe
- message
- méthode
- paramètre
- constante
- argument
- signature
- receveur
- état

```
public class UneClasse {

1.    public int att1;
2.    private String att2;
3.    protected UneAutreClasse att3;
4.    static int ATT3;
5.    static final int ATT4;
6.
7.    public UneClasse() {
8.      }

9.    public UneClasse(int att1) {
10.     this.att1 = att1;
11.     this.att2 = "default";
12.   }

13.   public int getAtt1() {
14.     return att1;
15.   }

16.   public void setAtt1(int att1) {
17.     this.att1 = att1;
18.   }
19.   …
20.   public void m1() {
21.     int i=5;
22.     this.att3.m2(att1,i);
23.   }
24.}
```

- altérateur
- spécificateur d'accès
- constructeur
- portée
- accesseur
- accessibilité
- mutateur
- variable locale
- littéral
- attribut
- comportement
- getter/setter

Conventions

- Classes
  - ✗ première lettre = majuscule

```
UneClasse
UneAutreClasse
```

- Méthodes
  - ✗ reflète une action → verbe
  - ✗ première lettre = minuscule

```
afficher();
getValeur();
setValeur();
```

Visibilité et passage de paramètres

```
class A {
private int attribut;

A (int i){attribut=i;}

boolean methode(A a){
return this.attribut==a.attribut;}
}
```

la méthode regarde si l'objet passé en paramètre a le même attribut

```
A a1=new A(3);
A a2=new A(4);
System.out.println(a1.methode(a2));
System.out.println(a2.methode(a1));
```

structure symétrique : on peut appeler sur a1 ou sur a2 ; on pourrait en faire une méthode de classe à 2 paramètres

la méthode appelée sur un objet a accès aux champs privés d'un autre objet de la même classe

en Java, l'unité d'encapsulation est la classe (et pas l'objet)

Cas particuliers de désignation

- Il est parfois nécessaire de se faire référence

  `this, self`

  exemples :

  - ✔ une méthode qui transmet l'objet (dans sa globalité) à une autre méthode
  - ✔ référence explicite à mon attribut x (this.x=x)

- Il est parfois nécessaire de faire référence à maman

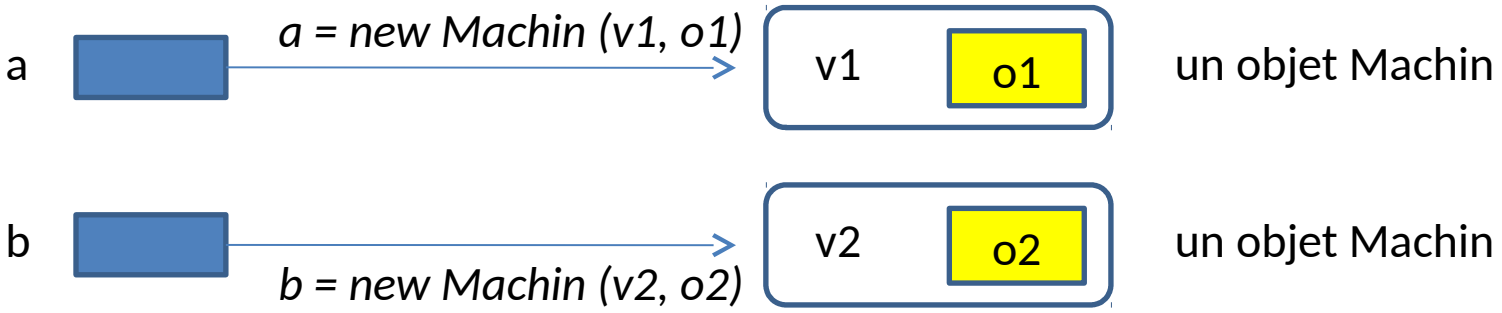
  `super`

  exemple : constructeur qui utilise le constructeur de la sur-classe

le constructeur de la classe fille appelle le constructeur de sa mère (super) pour initialiser les attributs hérités de la superclasse, et complète ensuite pour ses attributs propres

précepte : chaque classe initialise ses attributs

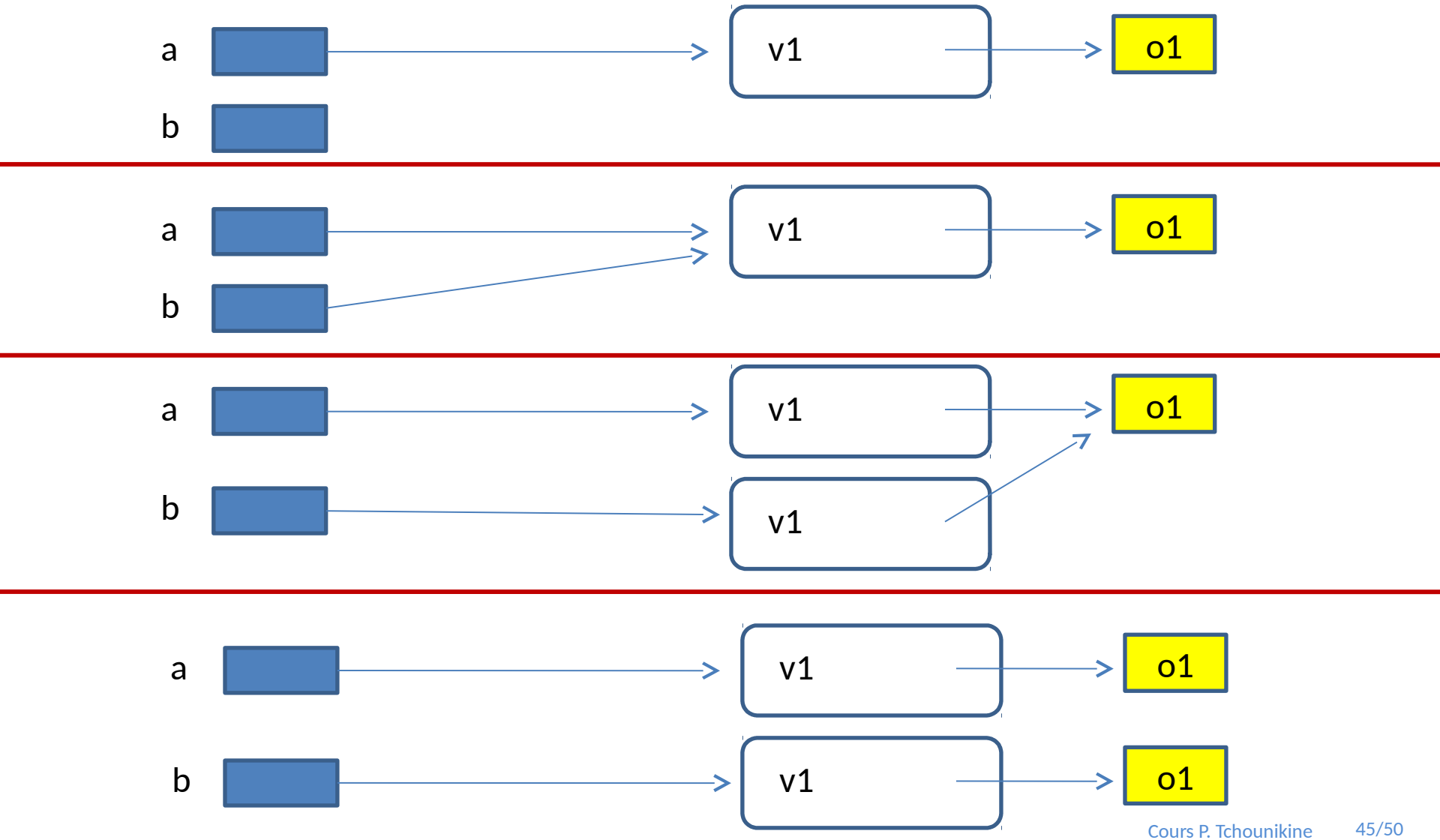
Référence et objet



- Affectation
- Copie superficielle / profonde (récursive)
- Comparaison

copie des valeurs / clonage (…)

Affectation, copie et clonage



Création d'un objet

appel de l'un des constructeurs

en entier !

Classe A

1. allocation mémoire d'un A
2. initialisation implicite (par défaut) des champs (0, false, « null », etc.)
3. initialisation explicite des champs (déclarations, s'il y a)
4. exécution des instructions du constructeur

Classe B extends A

1. allocation mémoire d'un B
2. initialisation implicite des champs (0, false, « null », etc.)
3. initialisation explicite des champs hérités de A (déclarations, s'il y a)
4. exécution des instructions du constructeur de A
5. initialisation explicite des champs de B (déclarations, s'il y a)
6. exécution des instructions du constructeur de B

il y a des cas (tordus) dans lequel cela peut avoir de l'importance

éviter les constructions « tordues » !

GL(APL)= Rip

Mort d'un objet

- Un objet non utilisé est candidat au « Garbage Collector »

GC : récupération de la mémoire **si/quand** utile

toute cette place / cette énergie là est gagnée

mécanique interne ∉ COO !

si pour des raisons très très particulières il y a besoin de faire disparaître explicitement des objets, c'est généralement possible (`finalize`)

Persistance des objets

c'est un problème important et compliqué

- Sauvegarde des valeurs des attributs (etc.)
  - ✗ simple … mais ne sauvegarde par l'organisation (problème des références !)

- Sérialisation : construction d'une « image mémoire » de l'organisation des objets
  - ✗ préserve la nature des objets et de leurs relations
  - ✗ possible / différents algorithmes (gestion récursivité notamment)
  - ✗ service proposé par différents langages

- Sauvegarde dans une base de donnée
  - ✗ base de donnée relationnelle (objet → enregistrement dans une table/classe)
  - ✗ base de donnée objet (dans l'ex-futur ?) (plusieurs si agrégation)

Rappels / structuration

- 1 classe = 1 type = 1 module = 1 fichier

- 1 programme = 1 ensemble de classes en interactions = 1 ensemble de fichiers liés

Rappels / Javadoc

- La documentation est un élément essentiel du GL
  - @author : auteur de l'élément décrit
  - @version : numéro de version
  - @param : nom du paramètre et descriptif
  - @see : crée un lien vers une autre documentation de classe
  - @since : numéro de la version initiale
  - @exception : liste les exceptions levées
  - @return : décrit ce que renvoie la méthode
  - … etc.