

Relations entre classes et objets

Christelle CAILLOUET
(christelle.caillouet@unice.fr)

Retour sur le TD2

Prendre vos clickers

Et brancher le récepteur...

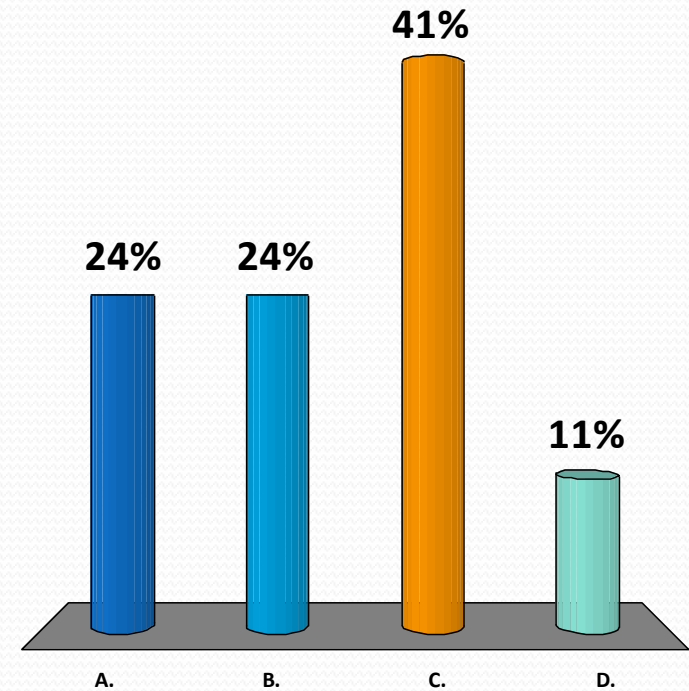
Comment déclarer correctement un tableau de 10 booléens ?

A. `boolean tab[10];`

B. `boolean tab = new tab[10];`

✓ C. `boolean tab = new boolean[10];`

D. `boolean[] tab.length = 10;`

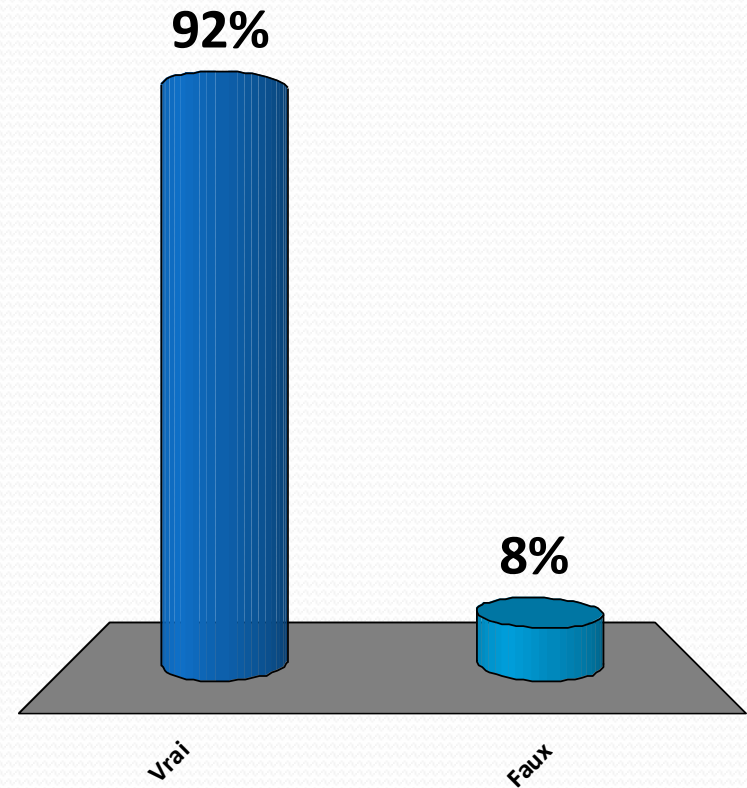


Est-ce correct ?

```
for (int j=2*i; j<= n; j+=i)
```

...

- ✓ A. Vrai
- B. Faux



Boucle for

- Possibilité de déclarer la variable du compteur directement à l'intérieur :

```
for (int i=0; i<=10; i++)
```

- Le dernier élément de la boucle est l'évolution du compteur → ce peut être n'importe quelle formule mathématique (comme les autres champs)

```
for (int i=100; i>0; i%=2)
```

```
for (int i=2*Math.PI; i>0; i--)
```

```
for (double i=0.0; Math.abs(i-1.0)<EPSILON; i+=0.1)
```

Jusqu'à maintenant...

- On sait :
 - Écrire une classe définissant un objet : constructeurs, accesseurs, fonctions, procédures, *toString()*
 - Créer et remplir une classe de tests contenant le *main*
 - Utiliser les types primitifs Java
 - Utiliser les tableaux en Java (au moins à 1 dimension)
 - Utiliser l'IDE Eclipse



Pour la suite...

- D'ici les vacances :
 - Cours aujourd'hui + Gestion des exceptions et fichiers
- D'ici l'IE du 16/03 :
 - Début + Cours sur l'héritage et polymorphisme (inclus)
- Ensuite :
 - Classes abstraites, interfaces
 - Collections, dictionnaires, objets utiles
 - Tests unitaires

Organisation d'un programme

Les package (*paquetages*)

- Regroupement logique de classes sous un identificateur commun
- Facilite le développement en répartissant les classes dans différents packages
(*même nom de classe autorisé dans des packages différents*)
- Déclaration : mot-clé **package** suivi du nom en première ligne du code source

```
1 package geom;
2
3 public class Square
4 {
5     private final double side;
6
7     public Square (double s)
8     {
9         side = s;
10    }
11 }
```

Utilisation des classes

```
1 package geom;
2
3 public class Square
4 {
5     private final double side;
6
7     public Square (double s)
8     {
9         side = s;
10    }
11 }
```

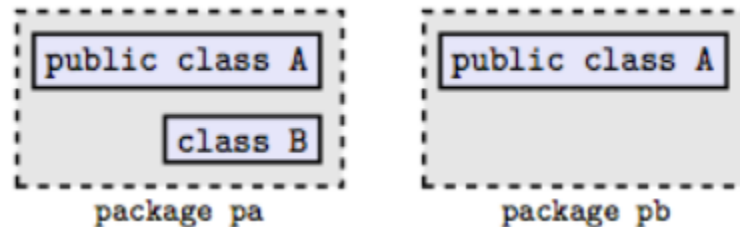
- Utilisation de la classe hors du package:
 - Soit par son nom complet : `geom.Square`
 - Soit en l'important : `import geom.Square`
 - Soit en important son package : `import geom.*`

Visibilité des classes

- Rappel concernant l'encapsulation
- Visibilité par défaut : dans le package.

Modificateur du membre	private	défaut	protected	public
Accès depuis la classe	Oui	Oui	Oui	Oui
Accès depuis une classe du même package	Non	Oui	Oui	Oui
Accès depuis une sous-classe	Non	Non	Oui	Oui
Accès depuis tout autre classe	Non	Non	Non	Oui

Visibilité des classes



- La classe B du package `pa` n'est pas visible depuis la classe A package `pb`.
- Depuis la classe A du package `pa`, les 3 classes sont visibles.

➔ La règle est simple :

- une classe **publique** est visible depuis tout l'univers Java (Appelé Java Universe (JU) en anglais),
- une classe avec visibilité par **défaut** ne l'est que depuis les classes du package dans lequel elle se trouve.

Les packages prédéfinis

- **java.lang** (Object, System, Math, String, ...)
- **java.util** (Date, Calendar, ArrayList, HashMap, ...)
- **java.applet**
- **java.awt**
- **java.awt.event**
- ...

Hiérarchie de classes

- Les classes d'un package sont organisées en hiérarchie
- Dans le package `java.lang`, toutes les classes sont dérivées de la classe `Object`, base de la hiérarchie

<http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html>

Object

Class `Object` is the root of the class hierarchy.

La classe **Object**

- Cette classe contient (sous forme de méthodes), les servitudes de base pour la gestion des objets
- **Transmet implicitement toutes ses méthodes à toute classe Java**
 - Relation d'héritage (cf. cours suivants)
 - Induit la nécessité de **redéfinir** ces méthodes dans toute classe Java (`@override`)
 - **Transtypage implicite** possible de toute référence sur un objet d'une classe quelconque, dans une variable de type `Object` (analogie avec le type `void*` du langage C)

La classe **Object**

- Met à disposition un constructeur par défaut
- Principales méthodes (d'instance)
 - `toString` : retourne un descriptif de l'objet cible
 - `equals` : prédicat d'égalité de 2 objets
 - `clone` : crée et retourne une copie de l'objet cible
 - `getClass` : retourne la classe de l'objet cible
 - ...
- Dans toute classe, on peut redéfinir les méthodes de la classe **Object**

<http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

L'affichage

- Méthode `public String toString()`

```
/*  
 * (non-Javadoc)  
 * @see java.lang.Object#toString()  
 */  
public String toString(){  
    return "("+abscisse+" , "+ordonnee+" )";  
}
```

→ l'objet Point a une représentation connue

Point a = **new** Point(1.0, 0.5);

System.out.println(a); → (1.0, 0.5)

Comparaison d'objets

- **Objets identiques :**
 - S'ils sont de la même classe
 - S'ils possèdent **la même référence**
- Opérateur ==

```
Point a = new Point(0.0, 1.25);  
Point b = a;  
System.out.println(a == b); → true
```

Comparaison d'objets

- **Objets égaux :**
 - S'ils sont de la même classe
 - S'ils possèdent **la même valeur des attributs**

```
Point a = new Point(0.0, 1.25);  
Point b = new Point(0.0, 1.25);  
System.out.println(a == b); → false
```

Comparaison d'objets

- Méthode `equals` de la classe `java.lang.Object` pour comparer les valeurs
- Nécessité de la *redéfinir* dans la classe des objets (comme *toString*)
- Contraintes de la méthode (précisées dans la documentation de la classe `Object`) :
 - Symétrie : pour deux objets `a` et `b`, si `a.equals(b)` alors `b.equals(a)`
 - Réflexivité : pour `a` non null, `a.equals(a)` ➔ `true`
 - Transitivité :
si `a.equals(b)` et `b.equals(c)` alors `a.equals(c)`
 - Pour toute référence non null, `a.equals(null)` ➔ `false`

Redéfinition de equals

- Signature formelle imposée pour cette méthode

`public boolean equals(Object op2)`

```
/*
 * (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals (Object op2){
    double x1= abscisse, x2= ((Point)op2).abscisse;
    double y1= ordonnee, y2= ((Point)op2).ordonnee;
    if (Math.abs(x2-x1) > EPSILON) return false;
    if (Math.abs(y2-y1) > EPSILON) return false;
    return true;
}
```

Comparaison d'objets

- Résultat

```
Point a = new Point(0.0, 1.25);  
Point b = new Point(0.0, 1.25);  
System.out.println(a.equals(b));
```

➔ **true**

Duplication d'objets

- La méthode

`public Object clone()`

- Redéfinition dans la classe en utilisant un constructeur de copie.

```
/**
 * Constructeur de copie
 */
public Point(double x, double y){
    abscisse = x;
    ordonnee = y;
}

/*
 * (non-Javadoc)
 * @see java.lang.Object#clone()
 */
public Object clone(){
    return new Point (abscisse, ordonnee);
}
```

Composition – Agrégation

La relation de composition

- Au cœur de la programmation orientée objet (POO)
- Modélise la relation d'appartenance « *has a* » ou « *is part of* »
- Nature des attributs d'une classe :
 - Jusqu'à présent : types primitifs, tableaux
 - **Maintenant : extension à des types Objet (et tableaux d'objets)**

Exemple

- Description d'une voiture :
 - Une carrosserie **Classe** Carrosserie
 - 4 roues **Classe** Roue
 - ...
- Description d'une roue :
 - Diamètre
 - Largeur
 - ...

Construction par assemblage

- Objectifs :
 - Favoriser la réutilisation des codes sources et la conception
 - Respecter les règles d'encapsulation
- Portée de la relation :
 - Relation exclusivement entre classes
 - Classe composée
 - Composants objets
 - Plusieurs niveaux et récursivité possibles

Les associations

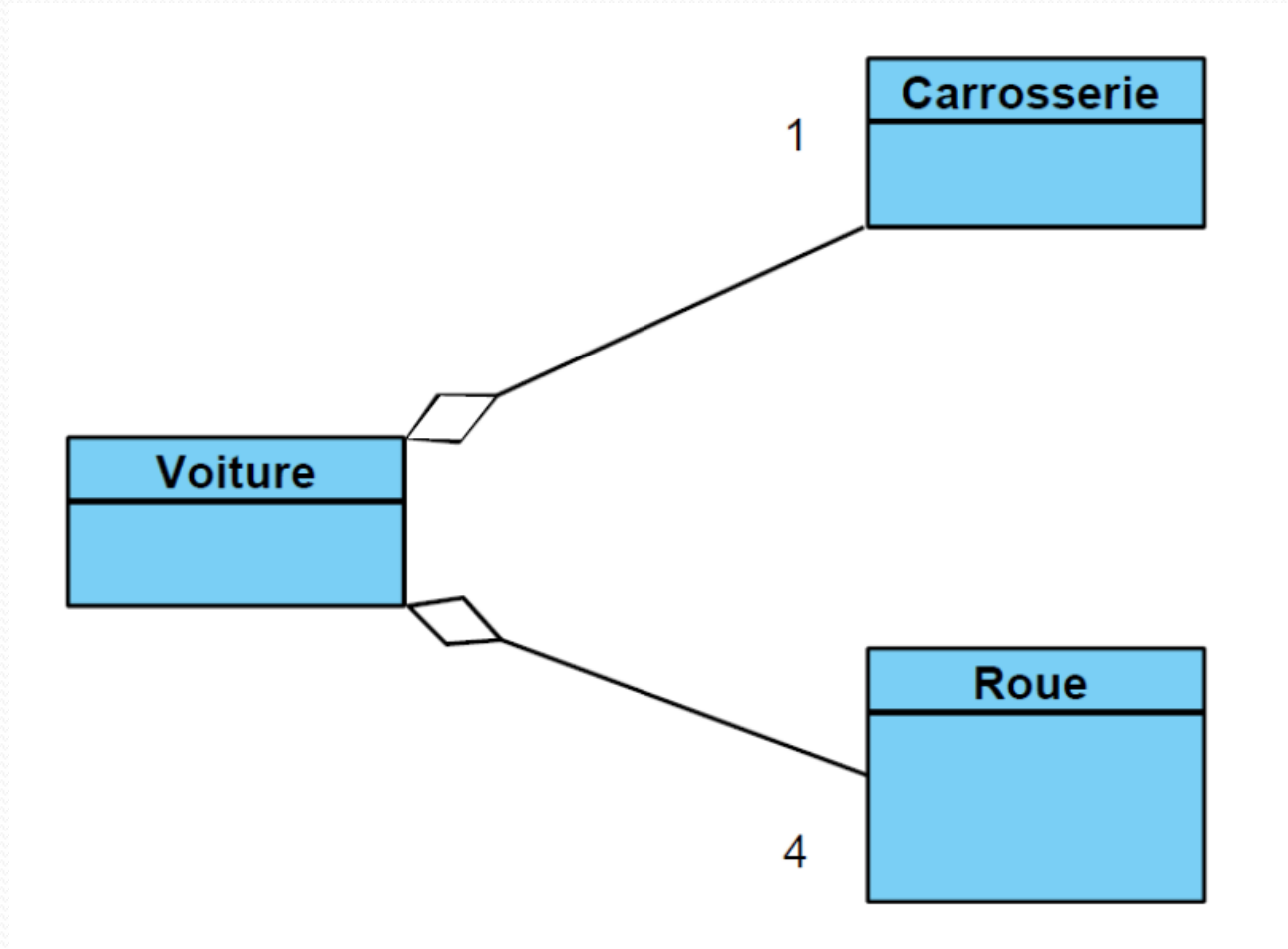
- En conception orientée objet (COO), relation entre les classes
- Chacune des extrémités de l'association définit le nombre d'instances des classes reliées qui sont impliquées dans cette association (**multiplicité**)

1

0..1

1..*

Exemple



Définition d'une classe composée

```
public class Voiture{
    private Carrosserie car;
    private Roue avD, avG, arD, arG;
    // Autres attributs éventuels
    private String immat;

    /*
     * Constructeur
     */
    public Voiture(Carrosserie obj1, Roue obj2,
        Roue obj3, Roue obj4, Roue obj5, String im){
        car = obj1;
        avD = obj2;
        avG = obj3;
        arD = obj4;
        arG = obj5;
        immat = im;
    }
}
```

Création des instances

```
public static void main(String[] args){  
    // Création des composants  
    Carrosserie laCarrosserie = new Carrosserie(...);  
    Roue laRoue1 = new Roue(...);  
    Roue laRoue2 = new Roue(...);  
    Roue laRoue3 = new Roue(...);  
    Roue laRoue4 = new Roue(...);  
  
    // Création du composé  
    Voiture maVoiture = new Voiture(laCarrosserie, laRoue1, laRoue2, laRoue3, laRoue4);  
}
```

Correspondance UML-Java

- **Agrégation**



- Le composant existe en dehors du composé
- Exemple :
 - Les roues et la voiture

Correspondance UML-Java

- Agrégation



```
public class A{
    private B b;

    public A(B x){
        b = x;
    }

    public void montrer(){
        System.out.print("mon agrégé : ");
        b.montrer();
    }
}
```

```
public class B{
    private String nom;

    public B(String x){
        nom = x;
    }

    public void montrer(){
        System.out.println(nom);
    }
}
```

```
public static void main(String[] args){
    B b = new B("unB");
    b.montrer();
    A a = new A(b);
    a.montrer();
}
```

Correspondance UML-Java

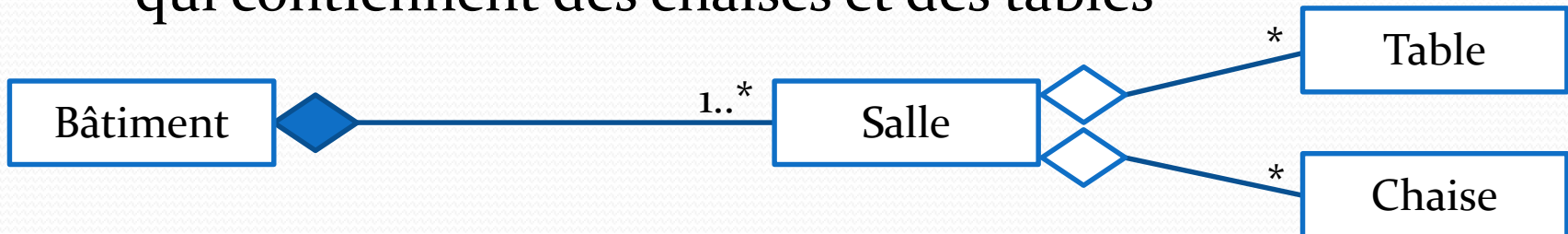
- **Composition**



- « Agrégation forte »
- Tous les composants sont détruits quand on détruit le composé

- **Exemple :**

- Un bâtiment de différents étages comporte des salles, qui contiennent des chaises et des tables



Correspondance UML-Java

- Composition



```
public class A{
    private B b;

    public A(){
        b = new B("mon B");
    }

    public A(String nomB){
        b = new B(nomB);
    }

    public void montrer(){
        System.out.print("mon agrégé : ");
        b.montrer();
    }
}
```

```
public class B{
    private String nom;

    public B(String x){
        nom = x;
    }

    public void montrer(){
        System.out.println(nom);
    }
}
```

```
public static void main(String[] args){
    A a = new A();
    a.montrer();
    A a2 = new A("toto");
    a2.montrer();
}
```

Quelques questions

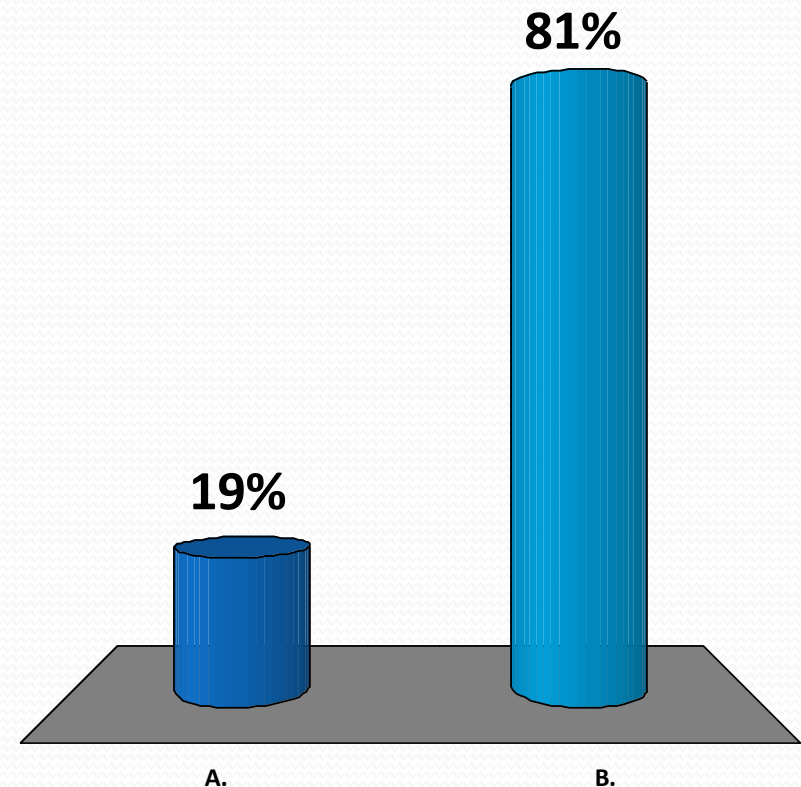
Agrégation ou composition ?

Une fenêtre graphique comporte :

- 2 barres de défilement,
- 1 barre de titre,
- 1 zone client.

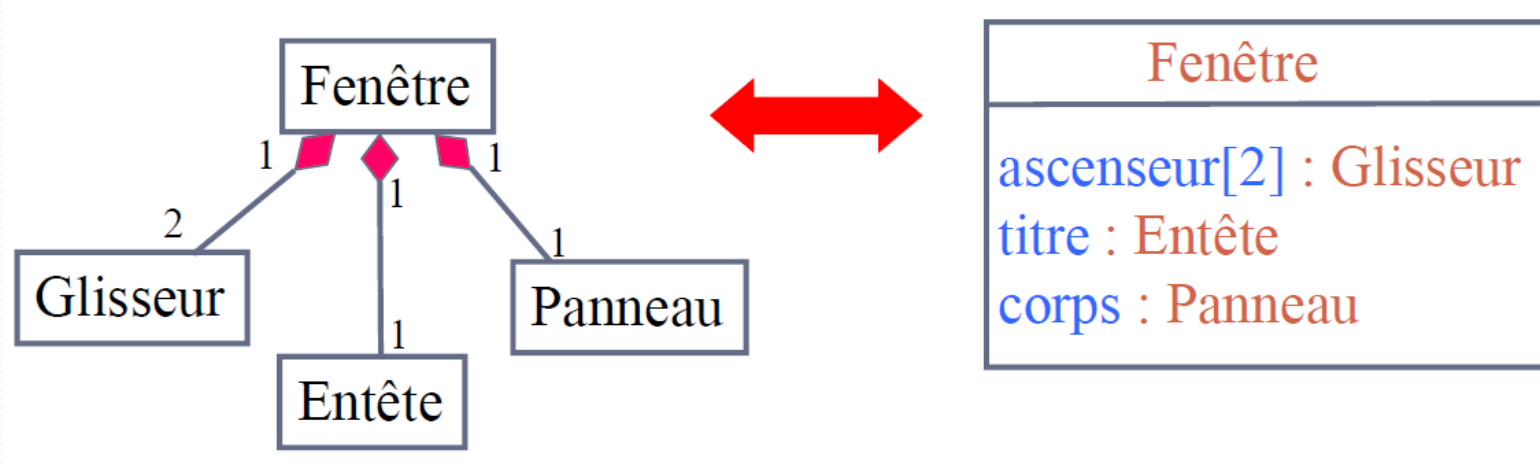
A. Agrégation

✓ B. Composition



Fenêtre graphique

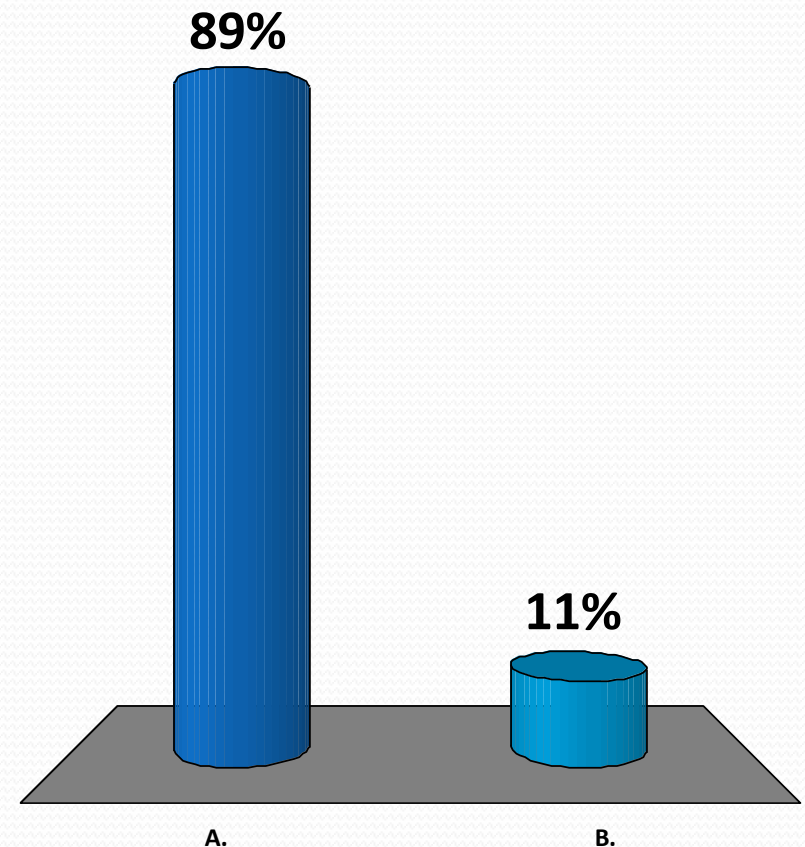
- Une fenêtre graphique comporte :
 - 2 barres de défilement (Objet Glisseur),
 - 1 barre de titre (Objet Entête),
 - 1 zone client (Objet Panneau).



Agrégation ou composition ?

Un segment est composé de deux points distincts.

- ✓ A. Agrégation
- B. Composition



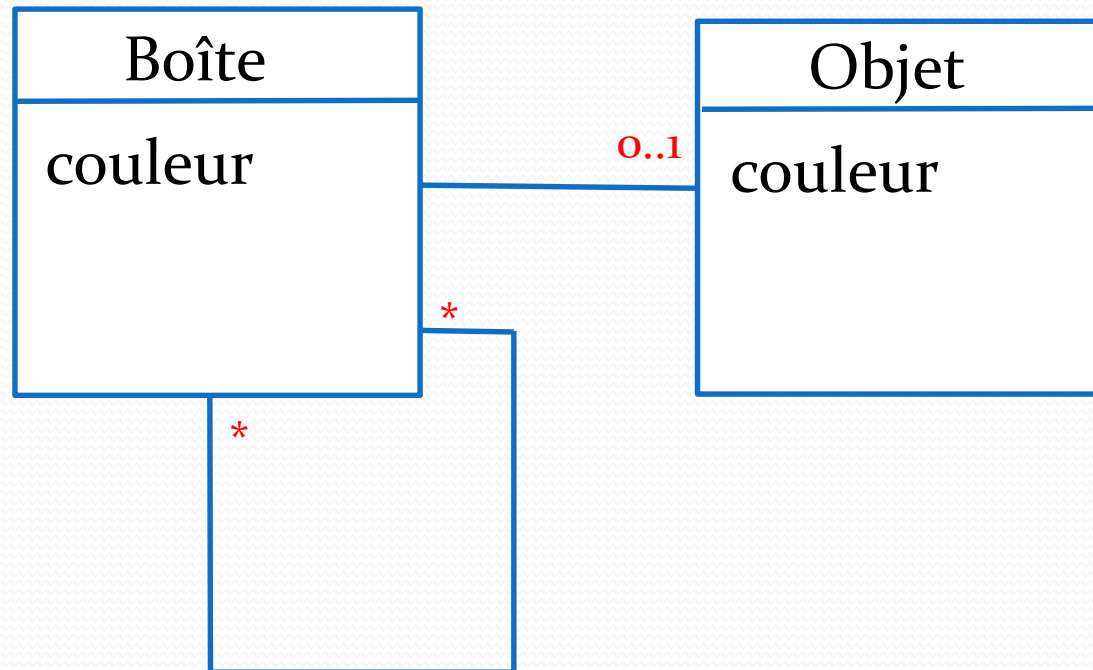
Segment

- Un segment est composé de deux points distincts.



Autres exemples

- Une boîte peut contenir un seul objet et d'autres boîtes.



Autres exemples

- Un livre possède un titre et un auteur
- Un auteur possède un nom et, dans sa carrière, écrit un ou plusieurs livres



Autres exemples

- Dans une bibliothèque, **un livre** peut être emprunté par **un lecteur**.
- De son côté **un lecteur** peut emprunter jusqu'à **3 livres** simultanément.
- **Un emprunt** est donc représenté par **un livre**, **un lecteur**, et **une date de retour** correspondant à 14 jours après la date du jour.

