# Associations UML et leur implémentation en Java

LIRMM / Université de Montpellier

Février 2017

# Sommaire

# Définition

- association : relation entre 2 ou plusieurs classes qui décrit les connexions structurelles entre leurs instances
- exemple : entre Pays et Ville : *a pour capitale* (ou dans l'autre sens : *est la capitale de*).
- au niveau instance : lien

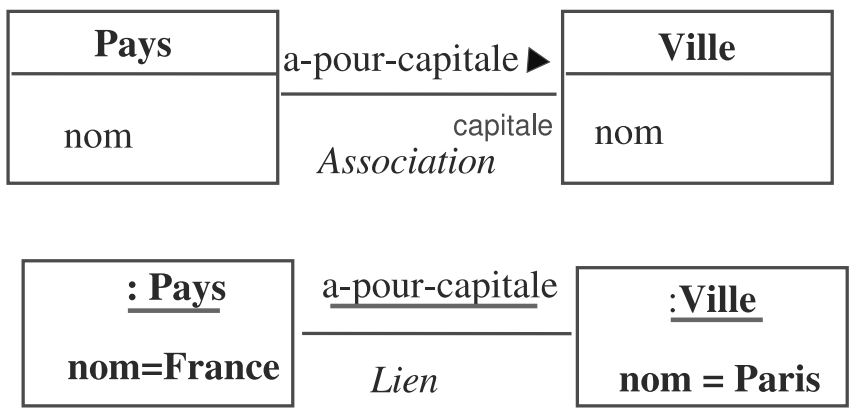



Figure – Associations et liens

# Représentation des associations

Association binaire : un trait entre 2 classes avec possiblement :

- le nom de l'association,
- le nom des rôles aux extrémités de l'association,
- la multiplicité des extrémités,
- la navigabilité.


Figure – Association binaire

# Associations et attributs

- Lisibilité : on voit bien mieux les liens entre classes avec une association qu'avec des attributs ;
- Liens dépendants : avec une association, on définit 2 liens dépendants : les 2 extrémités de l'association. Deux attributs de part et d'autre ne sont pas équivalents à une association ;
- Associations complexes : on peut définir des associations complexes.

## Convention utilisée dans la suite de ce module

- Convention : les attributs sont uniquement de type simple (entiers, flottants, booléens, ...).
- Conséquence logique : on ne mettra jamais un attribut de type complexe, on préférera alors une association.

# Sommaire

# Comment traduire les associations en Java ?

- On ne traduit que les extrémités navigables.
- Si extrémité de cardinalité $\leq 1 \rightarrow$ attribut.
- Si extrémité de cardinalité $> 1 \rightarrow$ types de bibliothèque représentant des collections : listes, tableaux, ensembles, ...
- Si association bidirectionnelle $\rightarrow$attention à bien à faire les mises à jour des 2 côtés. Ex : voiture et propriétaire.
- Si composition $\rightarrow$attention à bien respecter la contrainte de non partage de composants.

# Tableaux

- Déclaration
- Construction effective (avec une taille donnée)
- Initialisation

```
typ[] tab;
tab = new typ[10];
```

# Exemple de tableau d'int

```
int[] tab = new int[10];
for(int i = 0; i < 10; i++)
    tab[i] = i;
// autre solution
int[] tab2={0,1,2,3,4,5,6,7,8,9};
```

# Tableaux à plusieurs dimensions

```
typ[][] tab;
tab = new typ[N][M]; // N lignes, M colonnes
int[][] tab = {{1,2,3},{4,5,6},{7,8,9}};
```

# Qu'est-ce qu'une collection en Java ?

- Une classe qui définit une structure destinée à regrouper plusieurs objets
- Exemple : Pile, File, Liste chaînée, ...
- Une instance de Collection permet de manipuler les éléments de la structure (de pile, de file, de liste, ...) grâce à des méthodes bien pratiques : ajout d'élément, recherche, etc.

# Les collections Java

- Beaucoup de types complexes définis dans la bibliothèque Java pour les collections :
  - ensembles
  - listes
  - tableaux associatifs
  - ...
- Collections génériques
  - collections d'éléments de type E (où E est de type non primitif).
  - déclaration du type des éléments qui seront rangés dans la collection : collection d'entiers (en utilisant le type `Integer` et non `int`), de voitures, de personnes, etc.
- Opérations définies pour toutes les collections : l'ajout et la suppression d'éléments, l'obtention de la taille de la collection, etc.

# Qu'est-ce que la généricité ?

Le paramétrage d'une classe par un type.

# Qu'est-ce que la généricité ?

## Exemple d'une classe Pile

- Pile de quoi ?
- Dans quoi stocker les éléments de la pile ?
- Peut-on mettre n'importe quoi dans une pile ? Des assiettes et des feuilles ?

# L'exemple de la pile

- Côté programmation de la pile, on n'a pas précisément besoin de savoir ce qu'on mettra dedans : les piles d'assiettes et les piles de feuilles se ressemblent. Mais il nous faut quand même avoir la connaissance du type des objets manipulés ...
- Côté utilisation, on voudrait pouvoir dire :
  - "je veux une pile d'assiettes", et ne pouvoir y ranger que des assiettes
  - "je veux une pile de feuilles", et ne pouvoir y ranger que des feuilles
- Bilan : on veut fabriquer des piles contenant des éléments d'un certain type T, que l'utilisateur définira quand il déclarera sa pile.
  - Le programmeur de pile programme des piles de T
  - L'utilisateur précise ce qu'il veut pour T lors de l'utilisation d'une pile : T+assiette, ou T=Feuille, etc.

# Création de classes génériques

Rendez-vous l'année prochaine ...

# Utilisation de classes génériques

- Lors de la déclaration d'une variable dont le type est une classe générique
  - On précise entre $<$ $>$ le type des éléments que l'on veut manipuler
  - `Pile<Assiette> maPileAssiettes; // une pile d'assiette`
  - `Pile<Feuille> maPileFeuilles; // une pile de feuilles`
- Utilisation du constructeur
  - `maPileAssiettes=new Pile<Assiette>()`
- Ensuite on utilise normalement les méthodes prévues dans la classe générique
  - Si dans la classe `Pile<T>`, il y a une méthode `push(T elem)`
  - Alors
    - `maPileAssiettes.push(new Assiette());`
    - et surtout pas : `mapileAssiettes.push(new Feuille());`

# Collections Java : c'est facile !

## Un peu de méthode Coué ne peut pas nuire

Les collections Java sont très simples à manipuler, beaucoup plus que des tableaux classiques.

# Exemple des vecteurs

- Collections qui ressemblent aux tableaux, mais de taille non prédéfinie

```
Vector<MonType> v; // déclaration
v=new Vector<MonType>(); // création
```

Déclaration et création en une seule ligne :

```
Vector<MonType> v=new Vector<MonType>();
```

# Méthodes classiques des vecteurs

- `void addElement(E obj)` ou plus court `void add(E obj)`. Ajoute l'objet `obj` à la fin du vecteur, et augmente sa taille de 1.
- `E get(int index)`. Retourne l'objet placé en position `index` dans le vecteur.
- `boolean isEmpty()`. Teste si le vecteur n'a aucun élément.
- `int size()`. Retourne la taille du vecteur.
- `E remove(int index)`. Supprime l'objet en position `index` et le retourne.
- `boolean remove(Object o)`. Supprime la première occurrence de `o` rencontrée (laisse le vecteur inchangé si l'objet `o` n'est rencontré, et retourne alors faux).

# Exemple de vecteur de chaînes

```
Vector<String> prenoms = new Vector<String> ();
prenoms.addElement("Thomas");
prenoms.addElement("Sophie");
// prenoms.addElement(new Voiture()); -> interdit

//affichage des prénoms
for (int i = 0; i < prenoms.size(); i++)
{
   System.out.println(prenoms.get(i));
}
// autre affichage des prénoms avec la boucle
// for existant depuis Java 1.5
for (String prenomCourant:prenoms){
  System.out.println(prenomCourant);
}
```

# Tableau et Vector : un exemple

On veut manipuler des familles de mots. Comparons l'usage d'un tableau par rapport à un vector.

```
1   package tableauxVsVectors;
2
3   public class Mot {
4     private String mot;
5     private String traduction;
6
7     public Mot(String mot){
8         this.mot=mot;
9     }
10
11    public Mot(String mot, String traduction){
12        this.mot=mot;
13        this.traduction=traduction;
14    }
15
16    public String toString(){
17        String result=mot;
18        if (traduction!=null) result=result+"("+traduction+")";
19        return result;
20    }
21  }
```

# Programme manipulant les familles de mots

```
1   public static void main(String[] args){
2           ManipMotsTableaux mmt=new ManipMotsTableaux();
3
4           Mot ga=new Mot("ga");
5           Mot bu=new Mot("bu");
6           Mot zo=new Mot("zo");
7           Mot meu=new Mot("meu");
8           Mot zobuga=new Mot("ZoBuGa","pomper avec une petite pompe");
9
10          mmt.ajoutMot(ga);
11          mmt.ajoutMot(bu);
12          mmt.ajoutMot(zo);
13          mmt.ajoutMot(meu);
14          mmt.ajoutMot(zobuga);
15          System.out.println("il y a "+mmt.nbMots()+" mots");
16          System.out.print(ga+" existe ? ");
17          System.out.println(mmt.existeMot(ga));
18          System.out.print(zobuga+" existe ? ");
19          System.out.println(mmt.existeMot(zobuga));
20      }
```

# Manipulation d'une famille de mots stockée dans un tableau

```
1  package tableauxVsVectors;
2
3  public class ManipMotsTableaux {
4
5      // on va manipuler un tableau de mots
6      private Mot[] tabMots;
7
8      public ManipMotsTableaux(){
9          // Il faut creer le tableau, donc en fixer la taille
10         tabMots=new Mot[4];
11     }
12 ...
```

# Manipulation d'une famille de mots stockée dans un tableau

```
1   ...
2       // on ajoute le mot m
3       public void ajoutMot(Mot m){
4           int i=0; // indice auquel on peut placer m
5           boolean placeTrouvee=false; // on a trouve une place pour m
6           // on cherche une place vide
7           while (i<tabMots.length&&!placeTrouvee){
8               if (tabMots[i]==null){placeTrouvee=true;} // on a trouve
9               else {i++;} // on avance
10          }
11          if (placeTrouvee){
12              tabMots[i]=m;
13              System.out.println("ajout du mot "+m);
14          } else{
15              System.out.println("plus de place !");
16          }
17      }
18  ...
```

# Manipulation d'une famille de mots stockée dans un tableau

```
1   ...
2       public boolean existeMot(Mot mot){
3           boolean trouve=false;
4           int i=0;
5           while (i<tabMots.length&&!trouve){
6               if (tabMots[i]==mot){trouve=true;} // on a trouve
7               else {i++;} // on avance
8           }
9           return trouve;
10      }
11
12      public int nbMots(){
13          int result=0;
14          for (int i=0;i<tabMots.length;i++){
15              if (tabMots[i]!=null){result++;}
16          }
17          return result;
18      }
19  ...
```

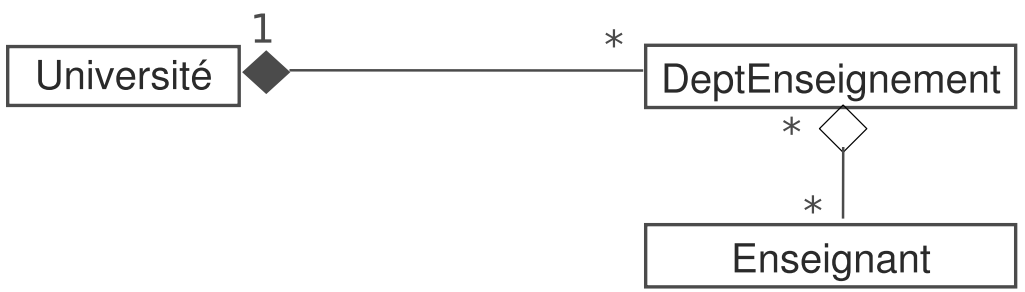
# Manipulation d'une famille de mots stockée dans un Vector

```
1  package tableauxVsVectors;
2  import java.util.Vector;
3  public class ManipMotsVector {
4      // on va manipuler un vecteur de mots
5      private Vector<Mot> vMots;
6
7      public ManipMotsVector(){
8          // Il faut creer le vector mais pas necessairement en fixer la
               taille
9          vMots=new Vector<Mot>();
10     }
11
12     // on ajoute le mot m
13     public void ajoutMot(Mot m){
14         vMots.add(m);
15         System.out.println("ajout du mot "+m);
16     }
17
18     public boolean existeMot(Mot mot){
19         return vMots.contains(mot);
20     }
21
22     public int nbMots(){
23         return vMots.size();
24     }
25 }
```

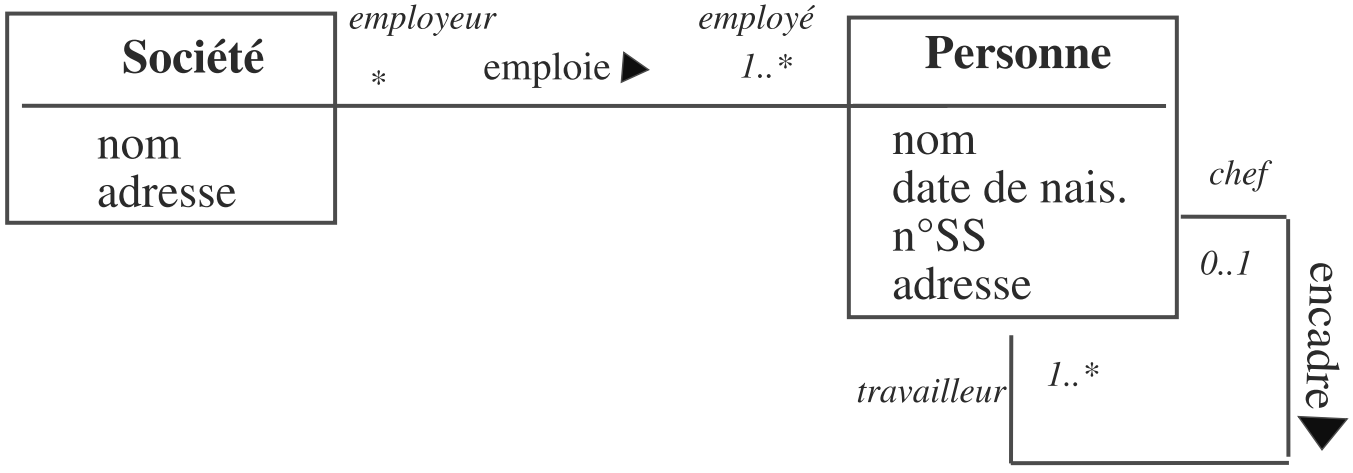
# Sommaire

# Agrégation et composition

- Agrégation : relation *ensemble-élément*, dénotée par un losange non rempli du côté de l'ensemble.
- Composition : relation de composition, on la note avec un losange plein du côté du composite. Notion d'exclusivité : un composant ne peut pas être partagé par plusieurs composites.
- "Despite the semantic linked to aggregation everyone think (for different reasons) that it is necessary. Consider it as a placebo" – James Rumbaugh.

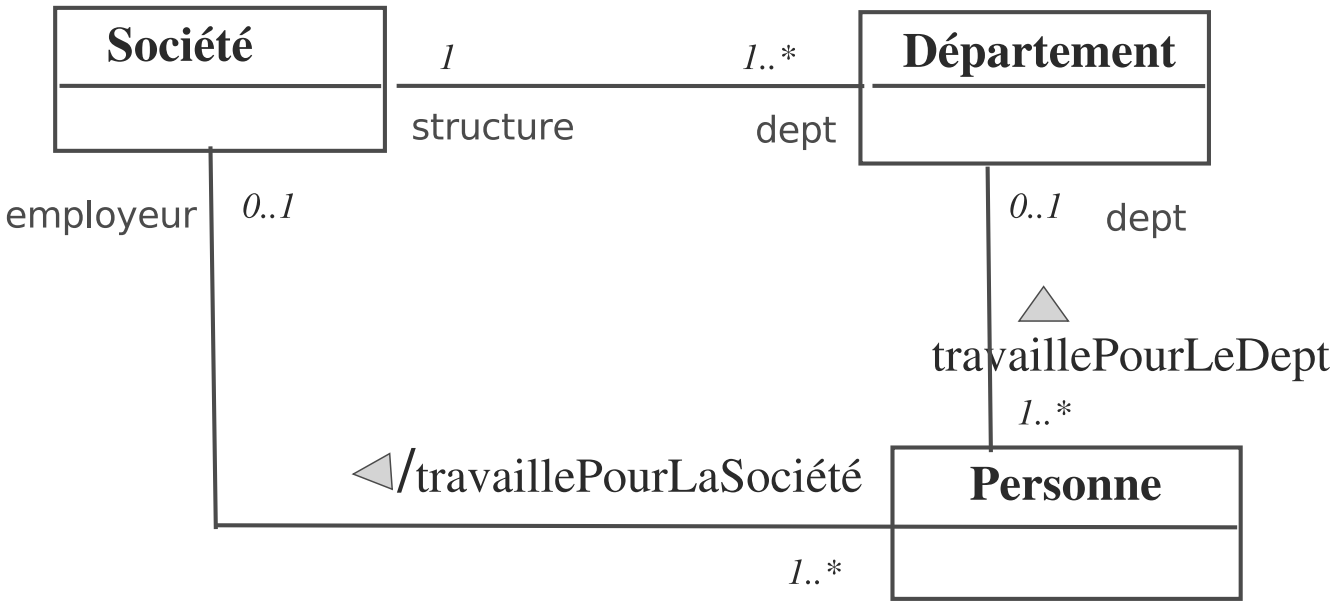


# Association réflexive



# Associations dérivées

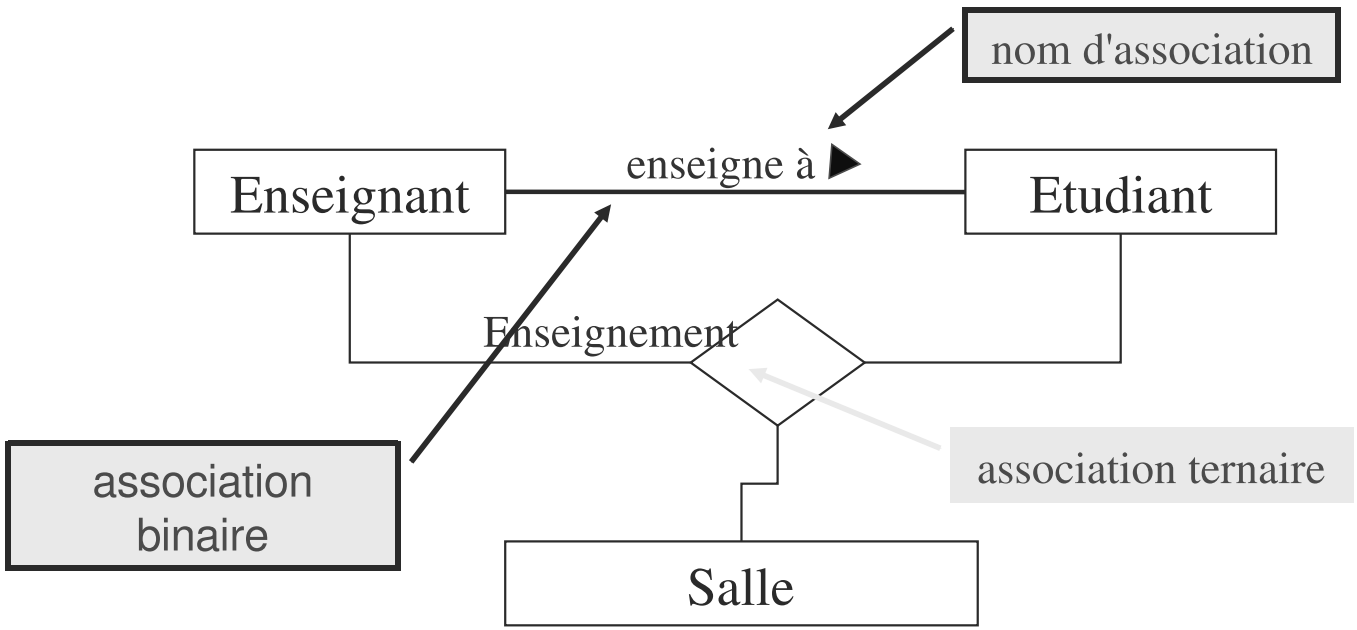
Association dérivée d'autres associations, marquée par : /.




{Personne.employeur=Personne.dept.structure}

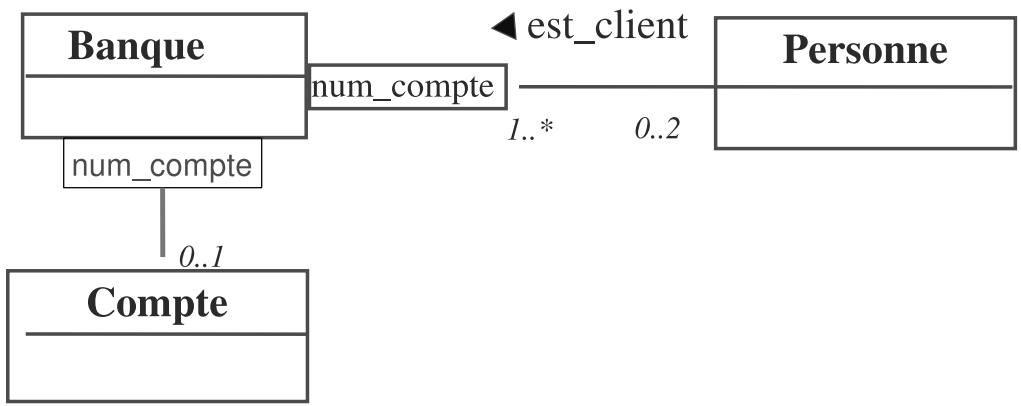
# Associations n-aires

Les associations peuvent être d'arité 3, 4, ...



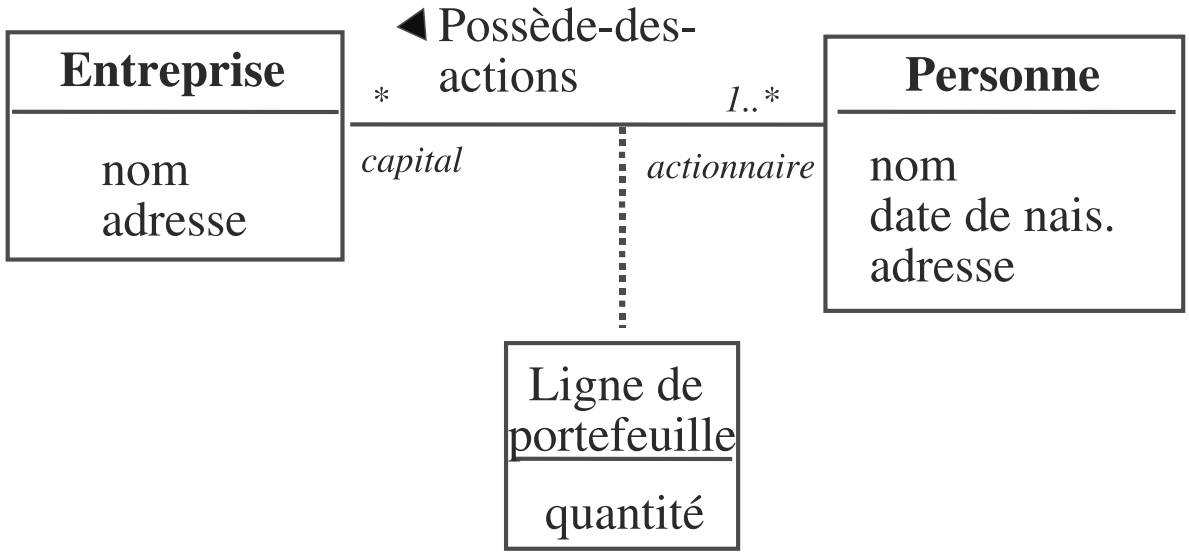
# Associations qualifiées

- Un qualifieur sur une association permet de sélectionner un sous-ensemble dans l'association
- Un qualifieur peut être typé (comme un attribut)

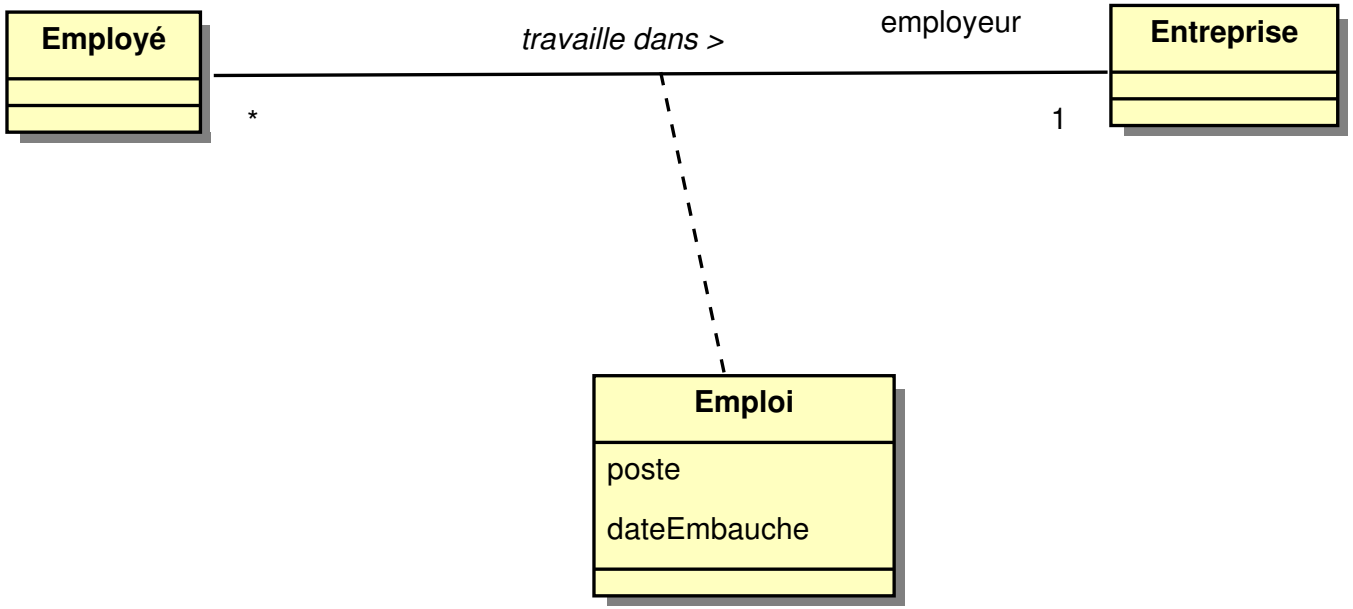


# Classes d'association

- Permet de rajouter des informations sur une association complexe (de réifier l'association)
- C'est une classe à part entière ⇒ peut être liée à d'autres classes



# Classes d'association, autre exemple

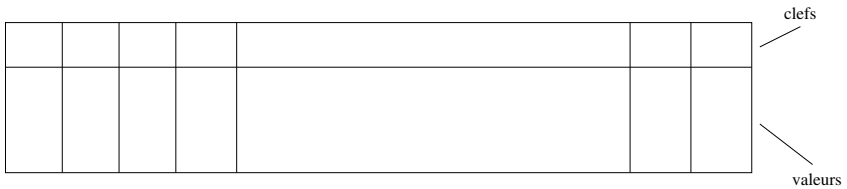


# Sommaire

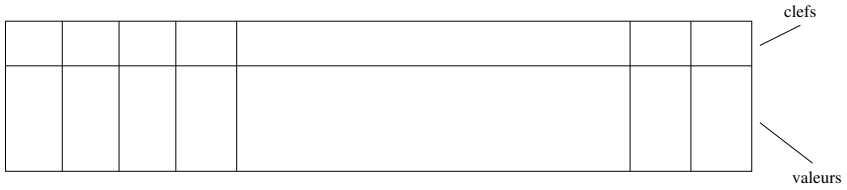
# Les tables de hachage : principe d'utilisation

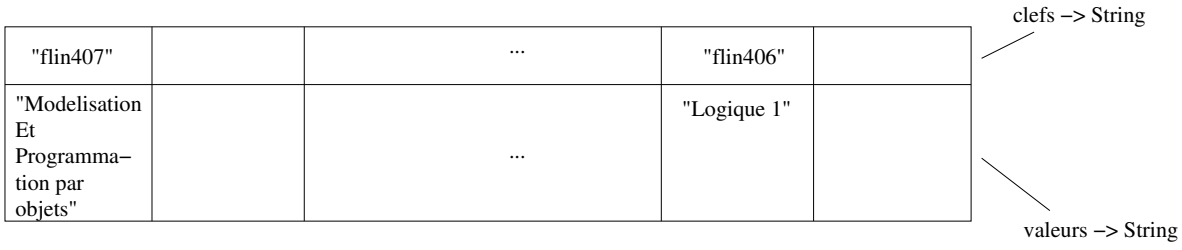
- Une table de hachage est une table indexée par une clef = elle implémente une table d'association entre une clef (un index) et une valeur.
- Par ex. utilisée pour traduire une association qualifiée.
- Exemple. Une table de hachage de comptes bancaires, indexés par leur numéro de compte, que l'on sait unique.
  - clef = numéro de compte, de type entier.
  - La table contient des comptes, accès direct grâce à la clef
- Intérêt des tables de hachage : accès très rapide à une valeur à partir de la clef
- Nous n'étudions pas ici le mécanisme sous-jacent mais juste l'utilisation des tables de hachage

# Tables de hachage : illustration

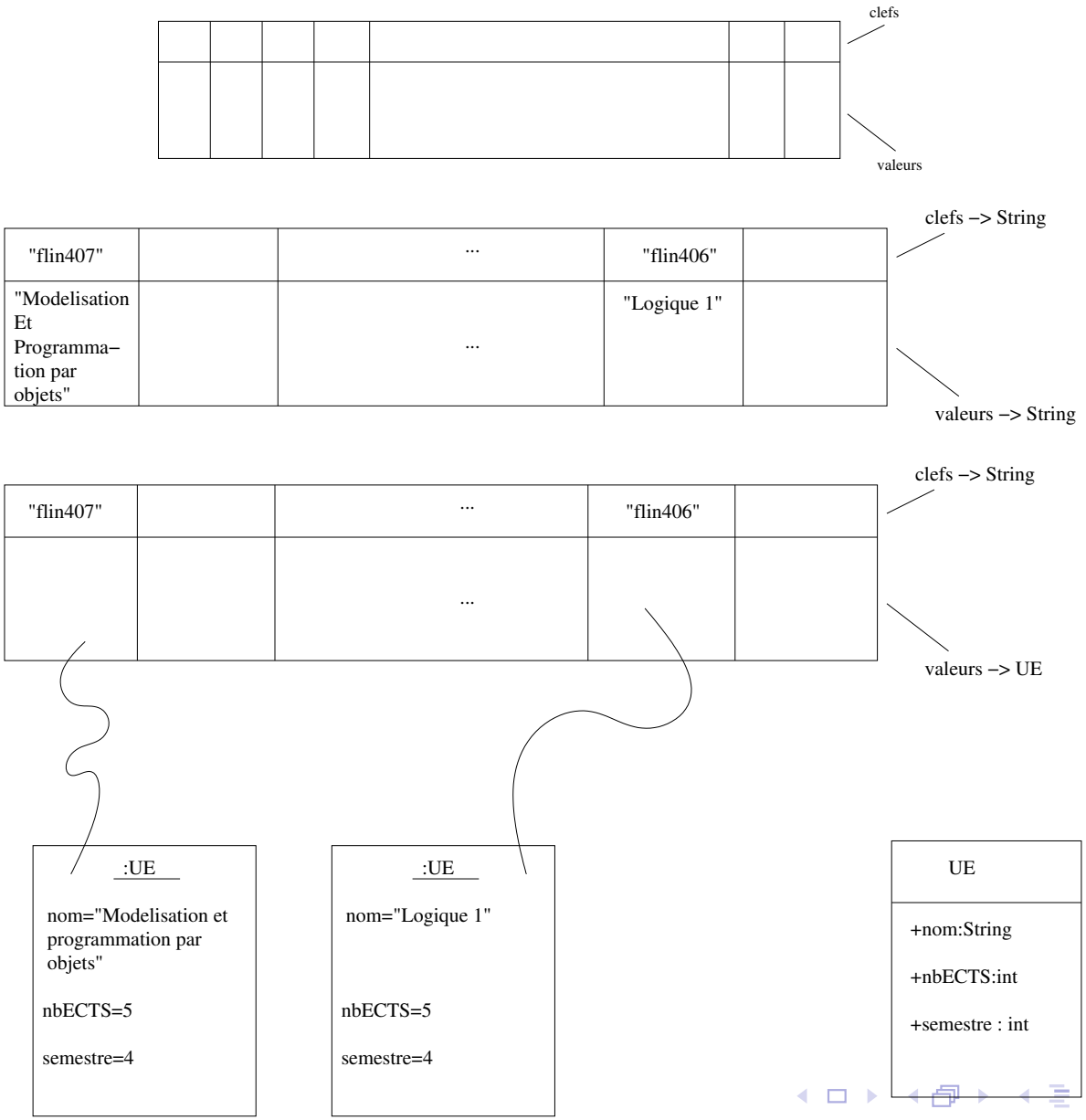


# Tables de hachage : illustration



# Tables de hachage : illustration



# Les tables de hachage : Hashtable

Une classe pour gérér les tables de hachage

```
Hashtable<TClef, TValeur> table=new Hashtable<TClef, TValeur>();
```

Exemples de méthodes pour une table d'objets de type V et de clefs de type K :

- `Object get(K key)` Retourne l'objet de clef `key` ou `null` s'il n'y en a pas.
- `V put(K key, V value)` Ajoute l'élément `value` avec comme clef `key`. S'il existait déjà un élément de même clef, cet élément est retourné (et écrasé dans la table par `value`).
- `V remove(Object key)` Retire l'élément de clef `key` de la table.
- `Collection<V> values()` Retourne une collection des valeurs contenues dans la table

# Les tables de hachage : exemple Java

```
Hashtable<String, Integer> numbers
  = new Hashtable<String, Integer>();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

Integer n = numbers.get("two");
if (n != null) {
   System.out.println("two = " + n);
}
```

# Autre exemple en Java

```
Hashtable<String, UE> ues_licence
  = new Hashtable<String, UE>();
ues_licence.put("flin407", new UE("Modelisation et
  programmation orientée objects",5,4));
ues_licence.put("flin406", new UE("Logique 1",5,4));

UE mpoo = ues_licence.get("flin407");
```

# UML vers Java : ce qu'on ne peut pas traduire directement

- Les classes d'association : on les transforme en général en classes "normales" fortement associées aux autres classes
- Les associations ternaires ou n-aires, avec n>2 : on réifie en général l'association (elle est implémentée comme une classe)