Approfondissement sur les classes

Le cas d'une association de cardinalités 1-1

Faculté des sciences, Université de Montpellier
Module HAI717I - Programmation

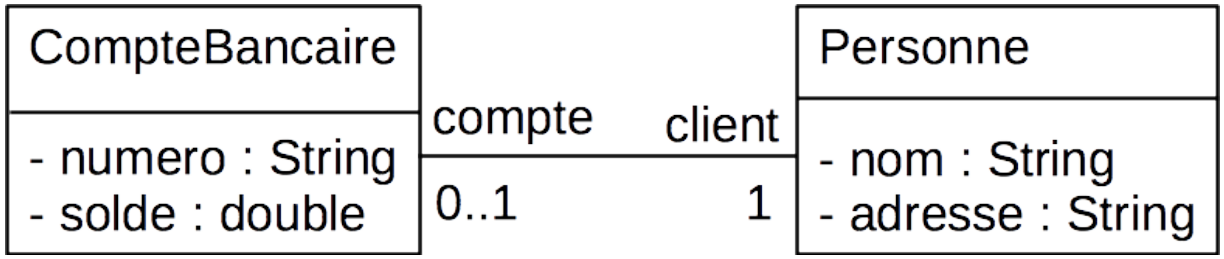
Approfondissement sur les classes

- Un programme à objets est en général composé de plusieurs classes
- Les classes sont associées les unes aux autres
  - En Java, par des attributs
  - En UML, préférentiellement par des associations
- Métaphore : les objets de ces classes communiquent en s'envoyant des messages
- Dans ce cours on étudie une simple association entre 2 classes

Association de cardinalité 1-1

Un compte bancaire est associé à une personne qui joue le rôle de client

Une personne peut avoir un compte bancaire (au plus un pour simplifier)



Dans le langage Java, pas de construction pour les associations

Association de cardinalité 1-1

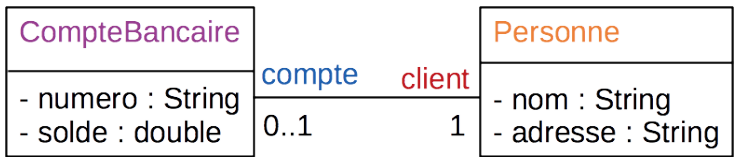
Java : pas de construction pour les associations

$\Longrightarrow$ on les fait apparaître sous forme de classe ou sous forme d'attributs.

- Dans les parties structurelles des classes (les attributs), on choisit ici deux attributs qui représentent les deux rôles de l'association
- Dans les parties comportementales (les méthodes), cela va nous faire réfléchir à la distribution du calcul entre les deux classes

Association de cardinalité 1-1

Rôles : deux attributs opposés




```
public class CompteBancaire {
      // attributs issus du compartiment "attribut" de la classe UML
      private String numero ;
      private double solde ;

     // attribut correspondant au rôle que joue la personne dans l'association
      private Personne client ;
...
}
public class Personne {
      // attributs issus du compartiment "attribut" de la classe UML
      private String nom ;
      private String adresse ;

      // attribut correspondant au rôle que joue le compte dans l'association
      private CompteBancaire compte ;
...
}
```

Contraintes de création et d'évolution des objets

Les règles régissant les objets doivent être énoncées le plus explicitement possible.

Elles sont différentes suivant les classes.

Ici, on choisit :

- une personne est créée sans compte bancaire ; elle peut ne pas en avoir ; elle peut en changer
- un compte bancaire est créé pour une personne à laquelle il est directement rattaché et avec un solde qui doit être positif ou nul

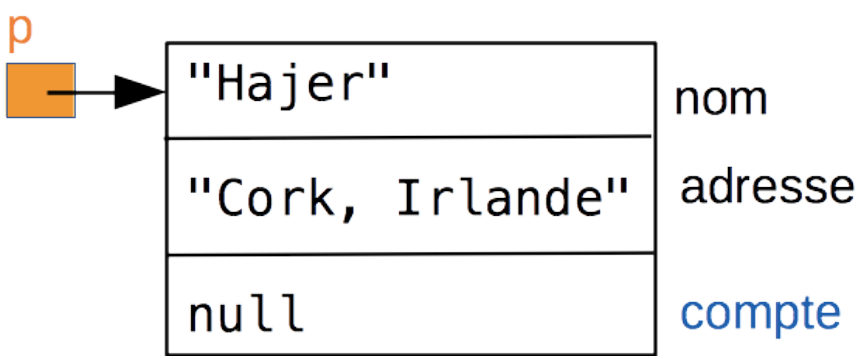
Constructeur de la classe Personne

- une personne est créée sans compte bancaire ; elle peut ne pas en avoir
- quand on crée une personne, on va donc initialiser seulement le nom et l'adresse

```
public Personne(String nom, String adresse) {
     this.nom = nom; this.adresse = adresse;
     // par défaut, this.compte vaut null
}
```

Constructeur de la classe Personne

```
public class Personne{
        public Personne(String nom, String adresse) {
             this.nom = nom; this.adresse = adresse;
             // par défaut, this.compte vaut null
        }
        ...
 }

     // dans un main
     Personne p = new Personne("Hajer", "Cork, Irlande");
```

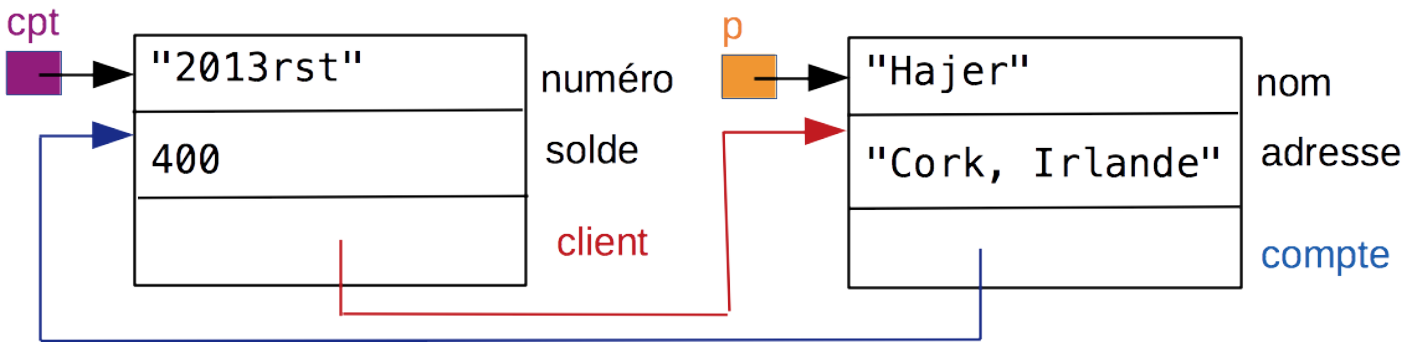


Constructeur de la classe CompteBancaire

un compte bancaire est créé

- pour une personne à laquelle il est directement rattaché
- avec un solde qui doit être positif ou nul

```
// dans le main
Personne p = new Personne("Hajer", "Cork, Irlande");
CompteBancaire cpt = new CompteBancaire("2013rst", p, 400);
```



Il faut mettre en place :

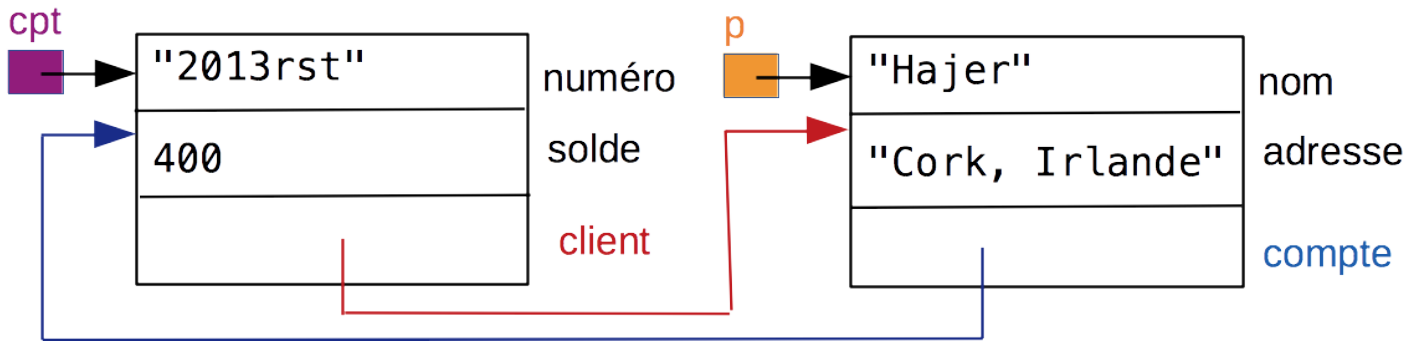
- le lien rouge du compte vers le client
- le lien bleu du client vers le compte

Constructeur de la classe `CompteBancaire`

- un compte bancaire est créé
  - pour une personne à laquelle il est directement rattaché
  - avec un solde qui doit être positif ou nul

```
public CompteBancaire(String numero, Personne client, double soldeInitial) {
   // on initialise le numero
   this.numero = numero;

   // on relie dans les deux sens le client et le compte
   this.client = client; // lien rouge
   client.setCompte(this); // lien bleu

   // on initialise le solde
   if (soldeInitial >= 0)
      this.solde = soldeInitial;
   else
      System.out.println("le solde initial doit être positif ou nul");
}
```

Constructeur de la classe CompteBancaire




```
// dans le main
CompteBancaire cpt = new CompteBancaire("2013rst", p, 400);

// dans la classe
public CompteBancaire(String numero, Personne client, double soldeInitial) {
.....
   // on relie dans les deux sens le client et le compte
   this.client = client; // lien rouge
   client.setCompte(this); // lien bleu
.... }
```

Pendant l'exécution de
CompteBancaire cpt = new CompteBancaire("2013rst", p, 400);
this est cpt
client est p

Accesseurs dans la classe Personne

Une personne peut changer de nom, d'adresse et de compte bancaire

$\Longrightarrow$ on crée une paire d'accesseurs pour chacun de ces attributs

```
public String getNom() {return this.nom;}
public void setNom(String nom) {this.nom = nom;}

public String getAdresse() {return this.adresse;}
public void setAdresse(String adresse) {this.adresse = adresse;}

public CompteBancaire getCompte() {return this.compte;}
public void setCompte(CompteBancaire compte) {this.compte = compte;}
```

Accesseurs dans la classe `CompteBancaire`

Un compte ne peut pas être rattaché plus tard à un autre client et il ne peut pas changer de numéro

$\Longrightarrow$ Donc pas d'accesseur `setNumero`, ni d'accesseur `setClient` ce qui évitera de les modifier (depuis une autre classe)

```
public double getSolde() {return this.solde;}
public void setSolde(double nouveauSolde) {this.solde = nouveauSolde;}

public String getNumero() {return this.numero;}

public Personne getClient() {return this.client;}
```

Exercice : (1) dans `setSolde` il faut vérifier que le paramètre `solde` est positif ou null, et afficher un message d'erreur dans le cas contraire (code semblable à une partie de celui du constructeur) ; puis (2) appeler `setSolde` dans le constructeur.

Répartition du calcul

Une règle est assez commune !

Responsabilité des classes

Une classe est responsable de ses propres attributs

Nous l'étudions dans le cas de l'écriture de la méthode `toString`

Sans répartition du calcul (mauvaise solution)

La classe `Personne` s'occupe de tout :
de ses attributs et de ceux de la classe `CompteBancaire`

```
public String toString() {
   String res = "nom "+this.nom+" adresse "+this.adresse;
   if (this.compte != null)
      { res = res+" num compte = "+this.compte.getNumero()
              +" solde = "+this.compte.getSolde();
   }
   else  res += " pas de compte associé";
   return res;
}
```

Inconvénient : les principes de la conception de la classe `CompteBancaire` ne sont pas spécialement considérés, par exemple ici la forme String d'une instance de `Compte` et chaque autre classe traitant de comptes construira cette forme à sa guise

Avec répartition du calcul (bonne solution)

La classe `Personne` délègue à la classe `CompteBancaire` la construction de la forme String de ses instances

```
public class Personne {....
   public String toString() {
      String res = "nom "+this.nom+" adresse "+this.adresse;
      if (this.compte != null){
         // délégation à la classe CompteBancaire
         res = res + this.compte.toString();
      }
      else { res += " pas de compte associé"; }
   return res;
   }
}

public class CompteBancaire{.....
   public String toString() {
      return " num compte = "+this.getNumero()
         +" solde = "+this.getSolde();
   }
}
```

Comprendre le passage de paramètres en Java

Passage de paramètre en Java

Un paramètre est toujours passé par valeur
On ne peut pas modifier une variable passée en paramètre

Deux exemples :

- avec un paramètre de type primitif
- avec un paramètre de type construit (classe)

Paramètre de type primitif

On ne peut pas modifier une variable passée en paramètre

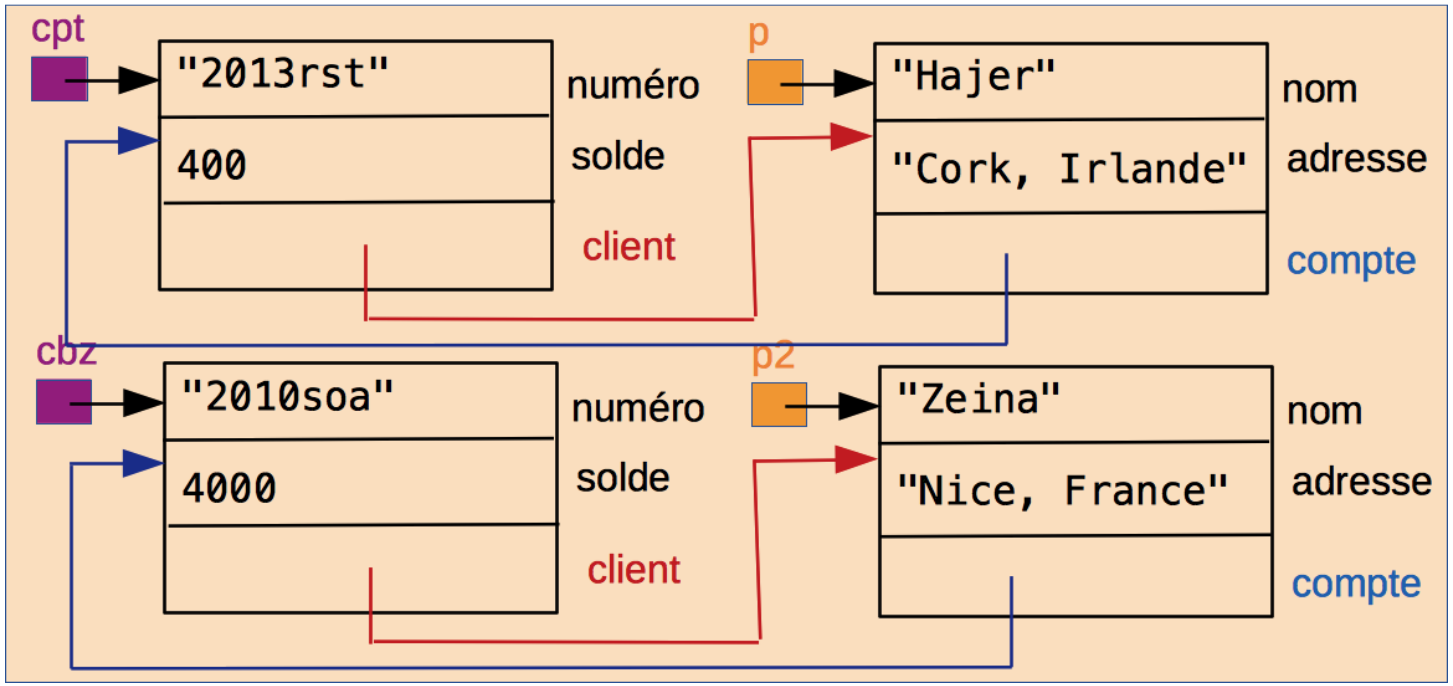
```
public class CompteBancaire{....
   public void augmenteDe10pourCents(double valeur){
         valeur = 1.1 * valeur; // ne change que le paramètre
   }
}

   // Dans un main
   double v = 100;
   System.out.println(v);
   cpt.augmenteDe10pourCents(v);
   System.out.println(v);    // v est inchangé ici
```

Paramètre de type référence d'objet

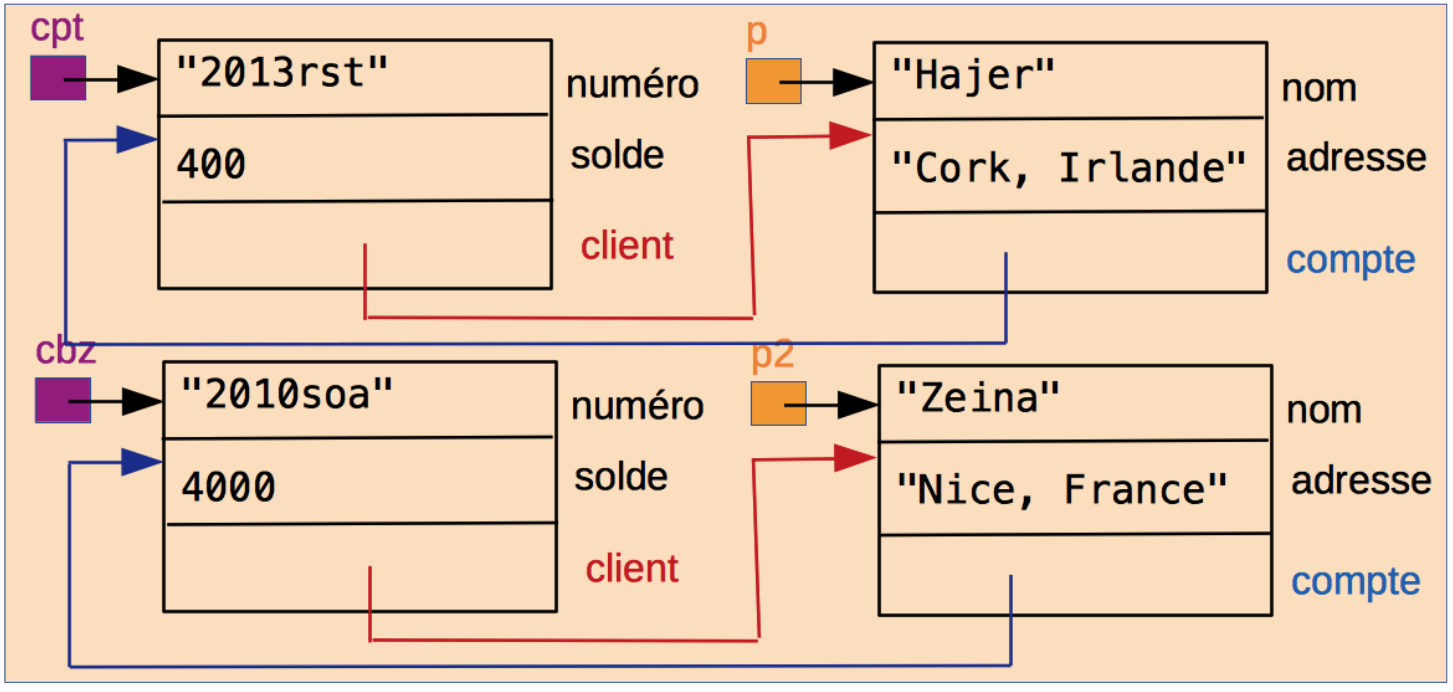
L'adresse/la référence de l'objet est passée par valeur.
On ne pourra pas modifier la variable adresse, mais on peut modifier l'objet au travers de l'adresse.

```
public class CompteBancaire{....
    public void virement(double v, CompteBancaire autreCompte) {
        if (v>=0 && this.getSolde()>=v) {
            this.solde = this.solde - v;
            autreCompte.solde = autreCompte.solde + v;
        }
        else {
            System.out.println("virement impossible");
        }
    }
}
// Dans un main
    Personne p2 = new Personne("Zeina","Nice, France");
    CompteBancaire cbz = new CompteBancaire("2010soa", p2, 4000);
    p2.getCompte().virement(1000, p.getCompte());
    // les soldes des comptes sont changés
```

Passage (de référence) d'objet en paramètre

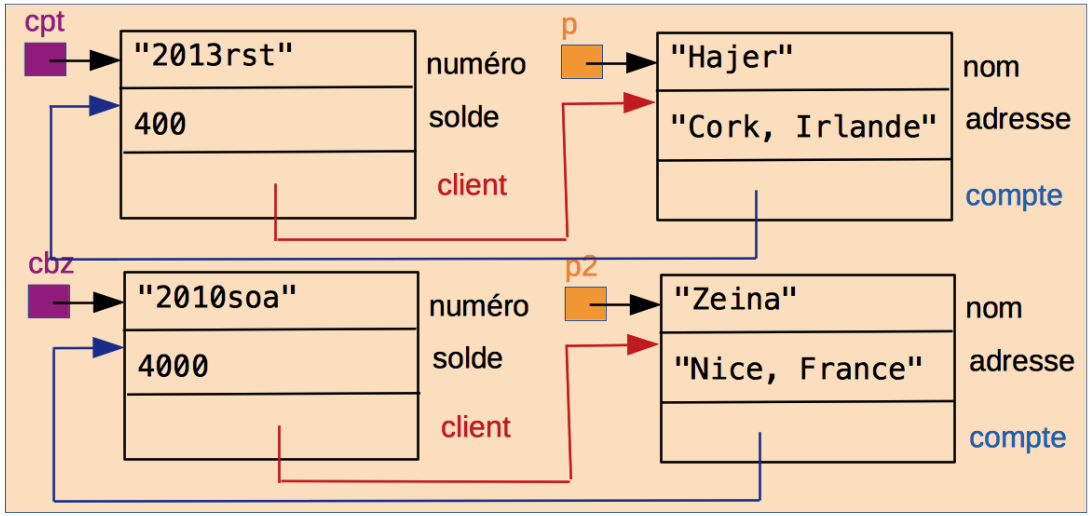


Passage (de référence) d'objet en paramètre

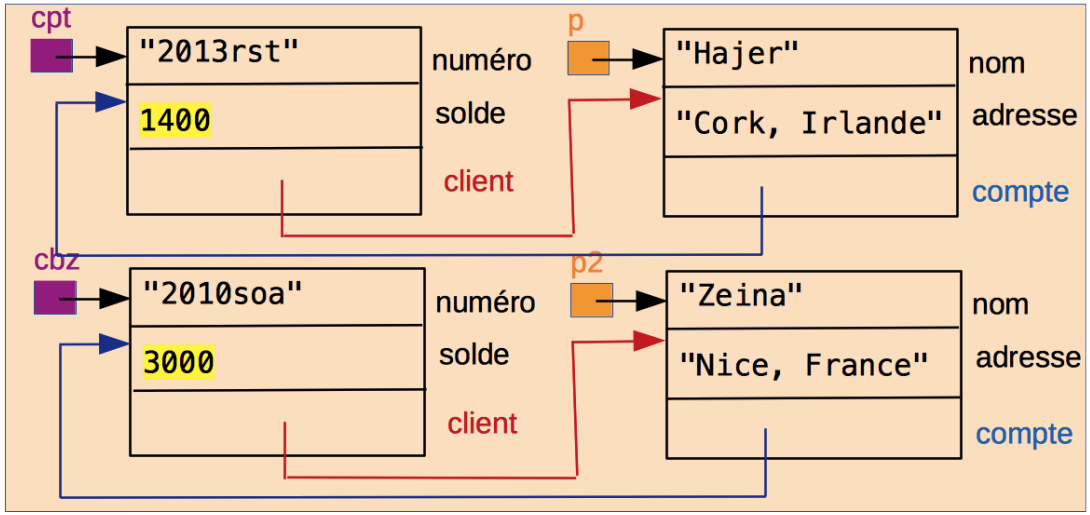



```
p2.getCompte().virement(1000, p.getCompte());
```

Passage (de référence) d'objet en paramètre

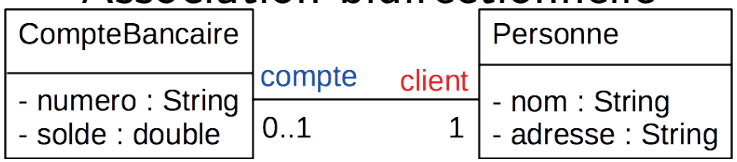


p2.getCompte().virement(1000, p.getCompte());

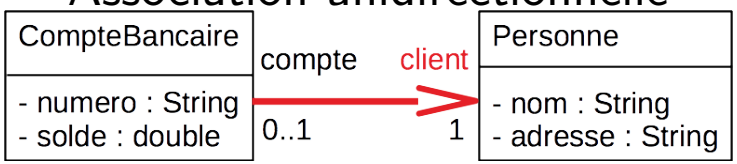


Différents types d'associations

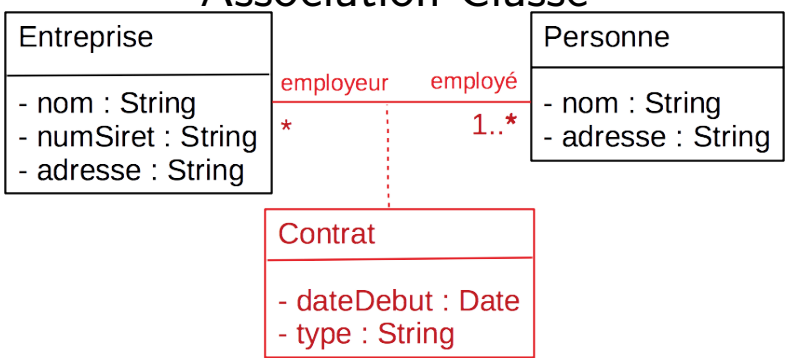
Association bidirectionnelle



Association unidirectionnelle



Association-Classe



Et bien d'autres, avec des mises en place variées en programmation !

Synthèse

- Mise en place de classes associées
- Explicitation des règles de fonctionnement
- Principe de distribution du calcul
- Passage de paramètres par valeur en Java