Associations UML et leur implémentation en Java

1ère partie

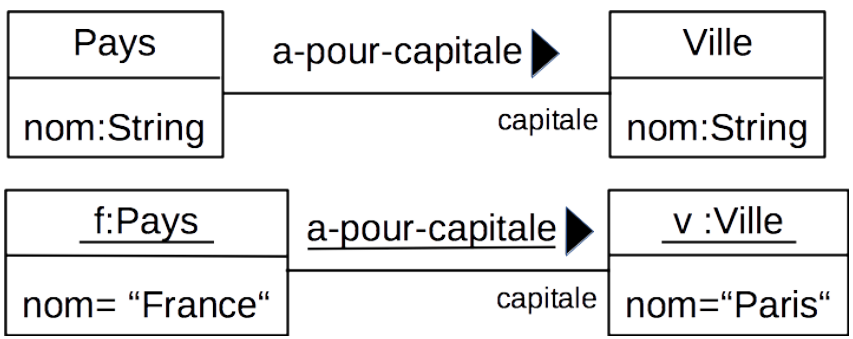
LIRMM / Université de Montpellier

3 mars 2022

Sommaire

Définition

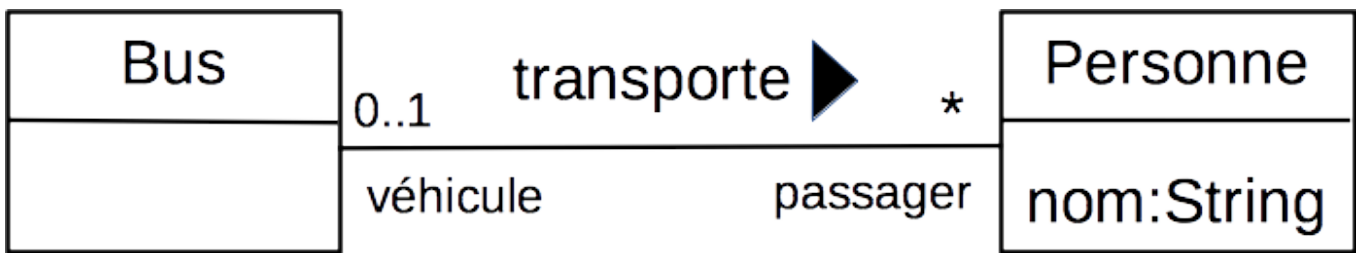
- une association est une relation entre 2 ou plusieurs classes qui décrit les connexions structurelles entre leurs instances
- une classe est un ensemble d'objets, une association est un ensemble de tuples
- *a-pour-capitale* $\subseteq$ *Pays* $\times$ *Ville*
- un lien relie des instances : $(f, v) \in$ *a-pour-capitale*



Représentation des associations

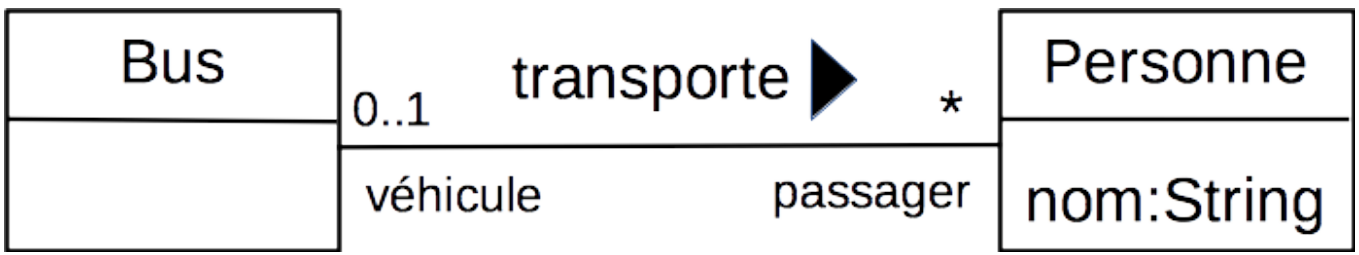
Association binaire = une ligne entre 2 classes éventuellement annotée par :

- le nom de l'association (orientation de lecture par le triangle noir)
- le nom des rôles aux extrémités de l'association, un rôle décrit la fonction de l'objet dans l'association
- la multiplicité des extrémités
- la navigabilité (flèche au bout de la ligne)



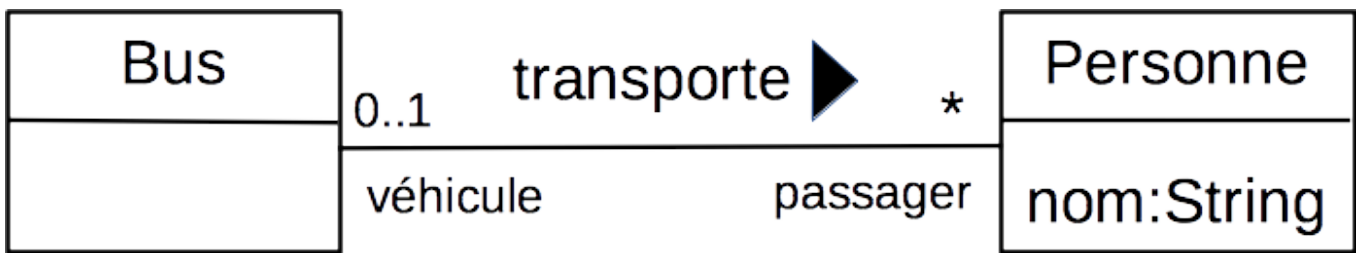
Nom d'association

- le nom de l'association est suivi par un triangle noir
- l'association s'appelle `transporte`
- on lit "un bus transporte des personnes" et non l'inverse
- $transporte \subseteq Bus \times Personne$



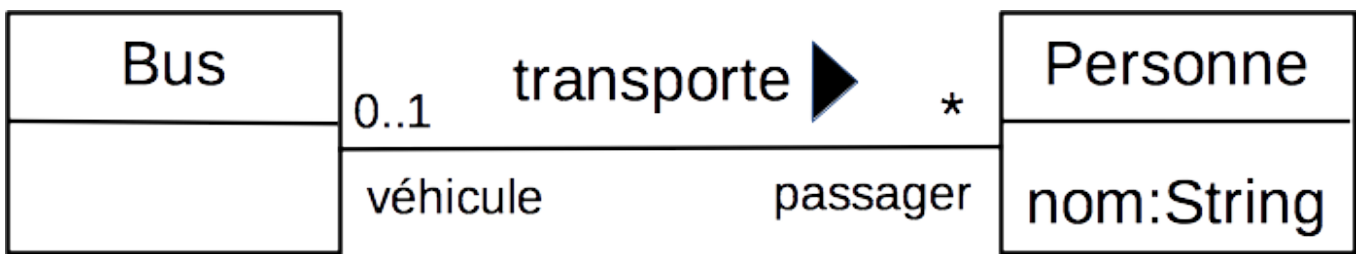
Nom de rôle

- le nom de rôle décrit la fonction d'un objet dans une association
- le bus joue le rôle de `vehicule` dans l'association
- la personne joue le rôle de `passager` dans l'association
- dans une autre association, la personne pourrait jouer un autre rôle : conducteur, contrôleur, etc.
- noter la position du nom de rôle, contre la classe qu'il décrit : `passager` contre `Personne`, `vehicule` contre `Bus`



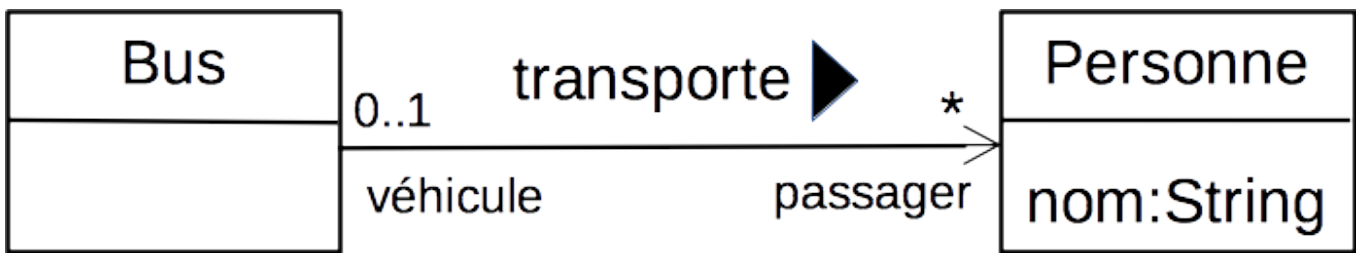
Multiplicité

- La multiplicité indique le nombre d'instances d'une extrémité qui peuvent être reliées à une instance de l'autre extrémité
- Technique : on fixe une instance d'un côté pour décider, en considérant les rôles
- Etant donné 1 bus, combien de personnes transporte-t-il ? **aucune ou plusieurs**, d'où *,
- Etant donné 1 personne, par combien de bus peut-elle être transportée (à un moment donné) ? **zéro ou un seul**, d'où **0..1**

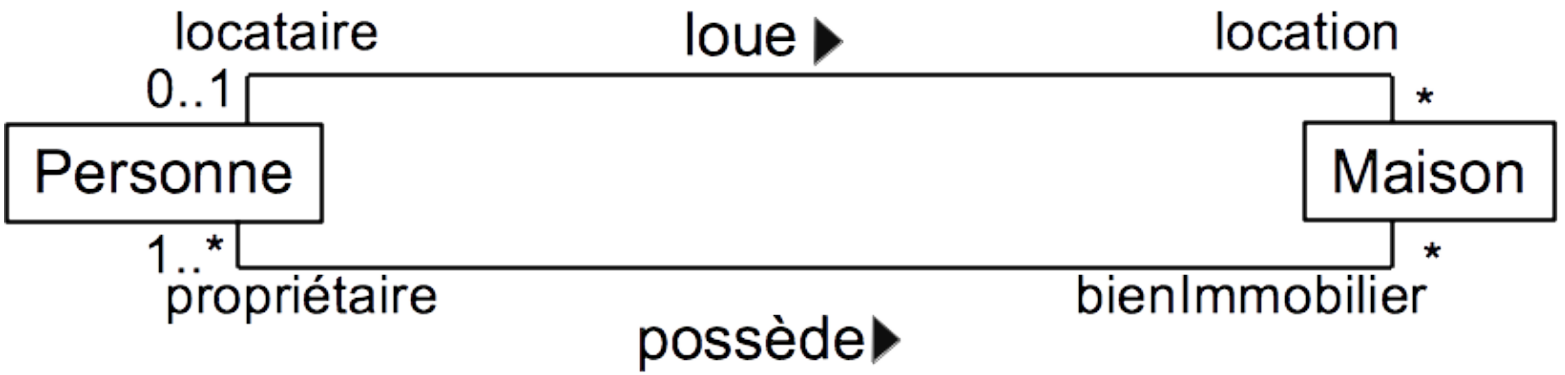


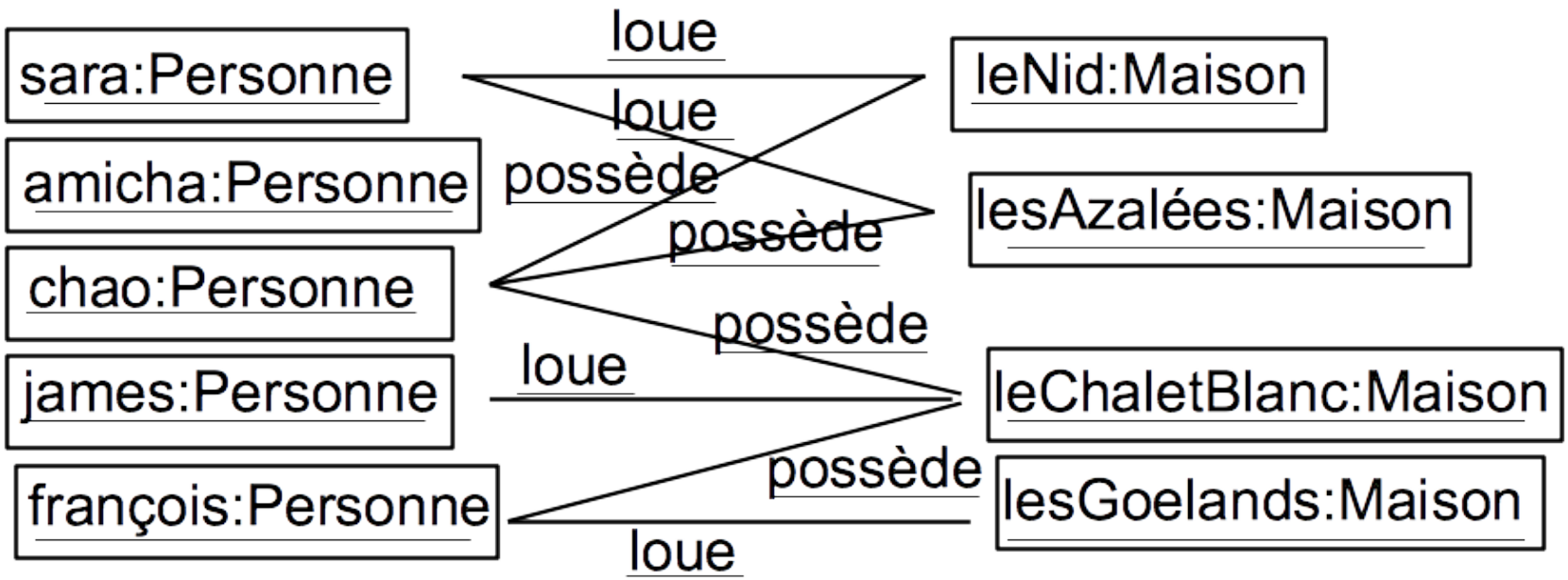
Navigabilité

- La navigabilité indique si un objet à une extrémité "connaît" les objets auxquels il est rattaché
- Cela sert à prendre une décision au moment de la traduction en attributs dans le langage de programmation (qui n'a en général pas de structure pour traduire directement les associations)
- La représentation (classe) de Bus contient la liste des passagers
- La représentation (classe) de Personne ne contient pas le véhicule dans lequel elle se déplace
- Quand rien n'est précisé ou s'il y a une flèche des deux côtés : bidirectionnelle

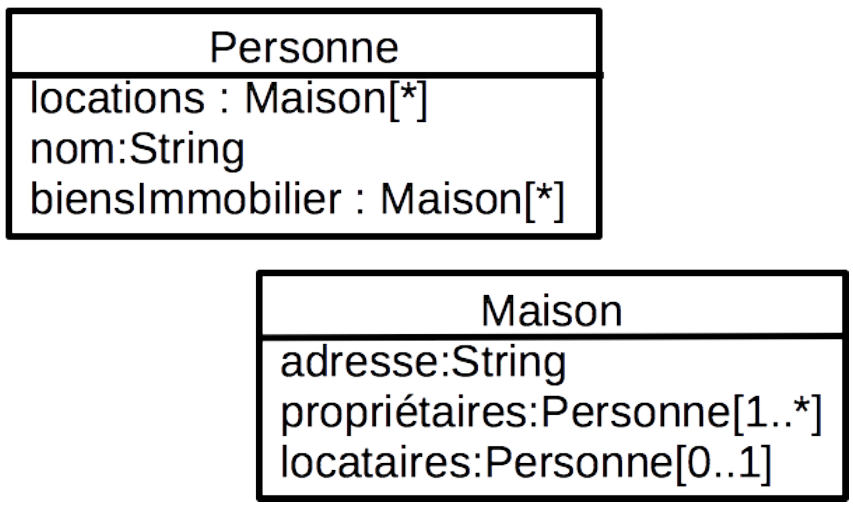


Classes connectées par plusieurs associations





Alternative avec des attributs



La représentation est ambiguë sur les paires d'attributs :
Quel attribut est l'opposé de quel attribut ?
`locations` est-il l'opposé de `locataires` ou de `propriétaires`

Associations et attributs

Intérêt des associations

- Lisibilité : on voit bien mieux les liens entre classes avec une association qu'avec des attributs
- Rôles dépendants : avec une association, on définit 2 rôles dépendants : les 2 extrémités de l'association
- Associations complexes : on pourra définir des associations complexes, par exemple pour attacher des attributs aux liens

Convention utilisée dans la suite de ce module

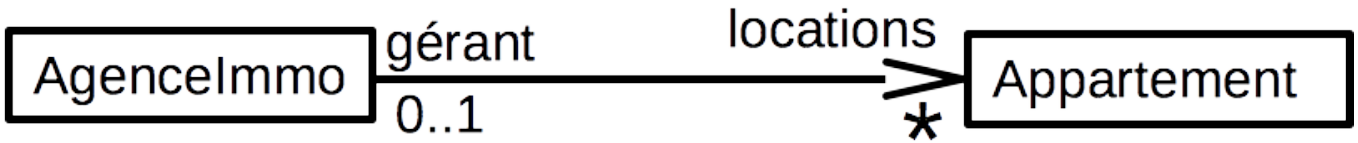
- Convention : les attributs seront uniquement de type simple (entiers, flottants, booléens, ...) ou des chaînes de caractères (String)
- Pas d'attribut de type complexe (classe), on préférera dans ce cas une association

Sommaire

Comment traduire les associations en Java ?

- Par une classe, ou par 1 ou 2 attributs
- On ne traduit par un attribut que les extrémités navigables
- Si extrémité de cardinalité $\leq 1 \rightarrow$ attribut du type de l'extrémité
- Si extrémité de cardinalité $> 1 \rightarrow$ attribut de type collection : liste, tableau, ensemble, etc.
- Si association bidirectionnelle $\rightarrow$ attention à bien à faire les mises à jour des 2 côtés

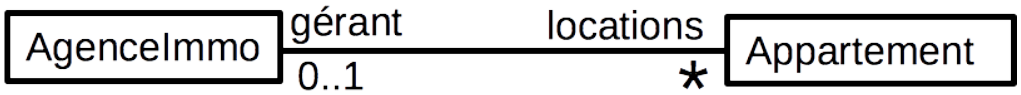
Traduction par 1 attribut (association unidirectionnelle)




```
public class AgenceImmo{
   private ArrayList<Appartement> locations;
   // collection d'appartements (voir plus loin)
}

public class Appartement{
   // rien concernant l'association
}
```

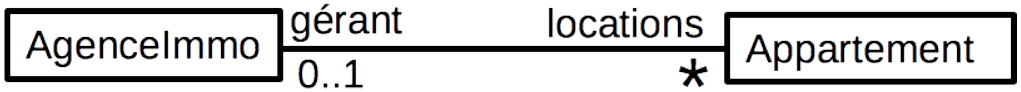
Traduction par 2 attributs (association bidirectionnelle)




```
public class AgenceImmo{
   private ArrayList<Appartement> locations;
   // collection d'appartements
}

public class Appartement{
   private AgenceImmo gerant;
}
```

Traduction par 1 classe (réification)

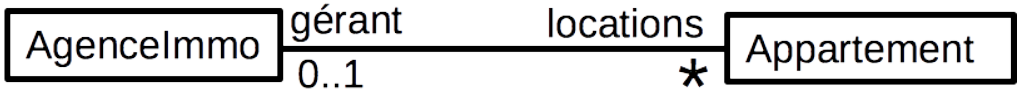



```
public class Location{
   private AgenceImmo gerant;
   private Appartement location;
   private Date dateDebut, dateFin;
}

public class AgenceImmo{
   // rien concernant Location
}

public class Appartement{
   // rien concernant Location
}
```

Traduction par 1 classe et liens inverses (réification)




```
public class Location{
   private AgenceImmo gerant;
   private Appartement location;
   ....
}

public class AgenceImmo{
   private ArrayList<Location> locations;
}

public class Appartement{
   private Location contratEnCours;
}
```

Sommaire

Collections

Groupe de valeurs d'un même type

- Tableaux primitifs (accès par un indice, taille fixe)
- Listes (accès par un indice, extensible)
- Ensembles (sans doublons)
- Dictionnaires associatifs (accès par une clef, extensible)

Le terme de `Collection` dans l'API Java est réservé pour certaines de ces structures.
Même type peut signifier instances d'une classe ou de ses sous-classes.

Classes collections

- Classes qui définissent des structures de données regroupant plusieurs objets ; certaines sont des classes abstraites ; certaines sont même des (types-)interfaces
- Exemples : Pile (`Stack`), Liste chaînée (`LinkedList`), Liste tabulaire (`ArrayList`), etc.
- Des méthodes de manipulation les rendent plus faciles d'usage que les tableaux primitifs

Les collections Java

Les collections sont génériques à plusieurs sens :

- Elles sont paramétrées par le type E des éléments stockés (type non primitif) : `ArrayList<E>`
- Des opérations de même nom et de même signification sont définies pour toutes les collections

La liste tabulaire (ArrayList)

- Collection qui stocke ses éléments dans un tableau et qui est extensible

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

Déclaration et création en une seule ligne :

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

Liste des appartements

Cas d'étude

Une agence immobilière est connectée à plusieurs appartements qu'elle gère

Intérêt de la liste pour notre cas d'étude

- Une liste est une succession de valeurs :
  - toutes de même type (des appartements)
  - ordonnées (par ex. chronologie d'arrivée)
  - indexées (par leur numéro de rang dans la succession)
  - extensible (utile si on ne connaît pas à l'avance le nombre de valeurs)
- Le type des valeurs doit être classe ou (type-)interface

  Si on désire une liste de valeurs d'un type primitif (int, double, ...), on devra utiliser une classe enveloppe (Integer, Double, ...)

Classe AgenceImmobiliere

```
import java.util.ArrayList;
public class AgenceImmobiliere {
    // Attribut représentant la liste des appartements gérés
    // On indique le type ArrayList
    // et le type des éléments stockés dans la liste,

    private ArrayList<Appartement> locations;  ..... }
```

Syntaxe

```
private ArrayList<Appartement> locations;
```

ArrayList : Type de liste utilisé
Appartement : Type des éléments stockés dans la liste
< > : Parenthèsage du type des éléments
locations : Nom de la variable

Créer la liste : solution 1

Création lors de la déclaration

```
import java.util.ArrayList;

public class AgenceImmobiliere {

    // Attribut représentant la liste des appartements gérés
    private ArrayList<Appartement> locations = new ArrayList<>();

    public AgenceImmobiliere() {
    }

    public AgenceImmobiliere(String nom) {
        this.nom = nom;
    }
    ...
}
```

Créer la liste : solution 2

Création dans un ou plusieurs constructeurs

```
import java.util.ArrayList;

public class AgenceImmobiliere {

    // Attribut représentant la liste des appartements gérés
    private ArrayList<Appartement> locations;

    public AgenceImmobiliere() {
       this.locations = new ArrayList<>();
    }

    public AgenceImmobiliere(String nom) {
        this.nom = nom;
        this.locations = new ArrayList<>();
   }
    ...
}
```

Méthodes importantes des listes tabulaires

Pour une ArrayList<E> :

- `void add(E obj)`. Ajoute l'objet `obj` à la fin de la liste.
- `boolean contains(E obj)`. Retourne vrai ssi `obj` est dans la liste.
- `E get(int index)`. Retourne l'objet placé en position `index` dans la liste.
- `boolean isEmpty()`. Retourne vrai ssi la liste n'a aucun élément.
- `int size()`. Retourne la taille de la liste.
- `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 la liste inchangée si l'objet `o` n'est rencontré, et retourne alors faux).

Connaître le nombre d'éléments

Méthode de la classe `ArrayList<T>` int size()
Pour la liste `maListe` maListe.size()

Exemple d'utilisation

```
/* Méthode de AgenceImmobiliere
 * Connaître le nombre d'appartements gérés
 * Illustration de la méthode size
 */

public int nbAppartGeres() {
   return this.locations.size();
}
```

Connaître l'élément de rang i

Méthode de la classe `ArrayList<T>` T get(int i)
Pour la liste `maListe` maListe.get(i)
Les indices vont de 0 à maListe.size()-1

Exemple d'utilisation

```
/* Méthode de AgenceImmobiliere
 * Connaître l'appartement de rang i
 * Illustration de la méthode get
 */

public Appartement apptRang(int i) {
   if (i>=0 && i <this.locations.size())
      return this.locations.get(i);
   else
      return null;
}
```

Appartenance d'un élément

Méthode de la classe `ArrayList<T>` boolean contains(T elem)
Pour la liste `maListe` maListe.contains(elem)

Exemple d'utilisation

```
/* Méthode de AgenceImmobiliere
 * Tester le fait que l'agence gère un certain appartement
 * Illustration de la méthode contains
 */

public boolean gere(Appartement appt) {
   return this.locations.contains(appt);
}
```

Ajout d'un élément

Méthode de la classe `ArrayList` boolean add(T nouvelElement)
Pour la liste `maListe` maListe.add(t)

Exemple d'utilisation

```
/* Méthode de AgenceImmobiliere
 * Ajouter un nouvel appartement
 * s'il y est déjà : afficher un message d'erreur
 * sinon : l'ajouter effectivement
 * Illustration des méthodes contains et add
 */

public void ajoute(Appartement appt) {
    if (this.locations.contains(appt))
        System.out.println("appartement déjà présent");
    else
        this.locations.add(appt);
}
```

Accesseur en lecture aux ArrayList

On évitera d'écrire un accesseur en écriture (`setLocations`).
On peut rendre les listes immuables et écrire un accesseur en lecture sans danger.

```
public class AgenceImmobiliere {
   private ArrayList<Appartement> locations = new ArrayList<>();

   // A éviter car on ne contrôlera rien de ce qui sera mis dedans
   // En effet on donne accès à la référence de la liste
   public ArrayList<Appartement> getLocations() {
      return this.locations;
   }

   // A préférer
   public List<Appartement> getLocations() {
      return (List<Appartement>)
                  Collections.unmodifiableList(this.locations);
   }
}
```

Sommaire

Effectuer un traitement sur les éléments

4 formes d'itérations sont disponibles en Java

- for avec un itérateur, pour tout parcourir
- for avec un indice, pour parcourir tout ou partie en connaissant le rang de l'élément
- while avec une condition
- do ... while avec une condition

for avec un itérateur

Pour tout parcourir

L'itérateur est une variable qui prend successivement comme valeur chaque élément de la liste

Exemple d'utilisation, l'itérateur est la variable `appart`

```
public void afficheAdresses() {
   for (Appartement appart : this.locations) {
         System.out.println("adresse :  "+appart.getAdresse());
   }
}
```

`appart` va être successivement le 1er, le 2e, le 3e ... appartement de la liste `locations`
`appart.getAdresse()` correspond à l'application de la méthode `getAdresse()` successivement à chaque appartement

for avec un indice

Pour parcourir jusqu'à un certain rang
L'indice est une variable de type entier qui prend successivement comme valeur des indices en restant entre 0 et `size()-1`

Exemple d'utilisation, l'indice est la variable `i`, qui varie de 0 à `this.nbAppartGeres()-1`.

`this.nbAppartGeres()` est égal à `size()`

```
public void afficheAdresses() {
   for (int i = 0; i < this.nbAppartGeres(); i++) {
      System.out.println("adresse appt. de rang = "+i+" "
         +this.locations.get(i).getAdresse());
   }
}
```

`this.locations.get(i)` va être successivement le 1er, le 2e, le 3e ... appartement de la liste `locations`

while avec une condition d'arrêt

Exemple d'utilisation, la condition d'arrêt est que la variable `i`, qui a été initialisée à 0, atteint `this.nbAppartGeres()`.
`this.nbAppartGeres()` est égal à `size()-1`

```
public void afficheAdresses() {
      int i = 0;
      while(i < this.nbAppartGeres()){
            System.out.println("adresse appt. de rang = "+i+" "
                  +this.locations.get(i).getAdresse());
            i++;
      }
}
```

`i < this.nbAppartGeres()` est la condition d'arrêt
`this.locations.get(i)` va être successivement le 1er, le 2e, le 3e ... appartement de la liste `locations`

do while avec une condition d'arrêt

Exemple d'utilisation, la condition d'arrêt est que la variable `i`, qui a été initialisée à 0, atteint `this.nbAppartGeres()`.
Le do-while effectue au moins une fois le corps de l'itération avant d'effectuer le test "i < this.nbAppartGeres()".
Il faut donc se prémunir du cas où la liste est vide.

```
public void afficheAdresses() {
    if (this.locations.isEmpty())
        System.out.println("pas d'adresse à afficher");
    else { int i = 0;
        do {
            System.out.println("adresse appt. de rang = "+i+" "
                    +this.locations.get(i).getAdresse());
            i++;
        }
        while (i < this.nbAppartGeres());
    }
}
```

Sommaire

Tableaux primitifs

```
// Déclaration
 Appartement[] tab;
 int[] tabEntiers;

 // Construction effective (avec une taille donnée)
  tab = new Appartement[10];
  tabEntiers = new int[4];

  // apres construction, si la taille doit évoluer ...
  // on recrée un tableau
  tab = new Appartement[15];
```

Tableaux primitifs

```
 // Construction avec des valeurs initiales
 tabEntiers = new int[]{11,13,17,19,23,29};

 // Initialisation par une itération
int[] tab = new int[10];
for(int i = 0; i < tab.length; i++)
      tab[i] = i;

// Création avec des valeurs initiales
int[] tab2={0,1,2,3,4,5,6,7,8,9};
```

Tableaux à plusieurs dimensions

```
int[][] tab;
tab = new int[N][M]; // N lignes, M colonnes

int[][] tab = {{1,2,3},{4,5,6},{7,8,9}};
```

Sommaire

Synthèse

- Une forme simple des associations UML (nom, rôles, multiplicités/cardinalités, navigabilité)
- Leur traduction en Java par des attributs et/ou une classe
- Les tableaux primitifs (que nous ne développerons pas beaucoup dans ce module) et les listes tabulaires (`ArrayList`)
- Les répétitives `for/itérateur`, `for/indice`, `while`, `do while`

À suivre ...

- Associations complexes
- Dictionnaires associatifs