Héritage et spécialisation/généralisation

Faculté des Sciences / Université de Montpellier
Modélisation et programmation par objets

2017

Héritage et spécialisation/généralisation

Concept

- Extension : ensemble des objets couverts
- Intension : prédicats, caractéristiques des objets couverts
- Classes et interfaces Java, classes C++

Concept Rectangle

- Extension : ensemble des rectangles
- Intension : ensemble des caractéristiques des rectangles, posséder quatre côtés parallèles 2 à 2, deux côtés consécutifs forment un angle droit, notion de largeur, de hauteur, etc.

Héritage et spécialisation/généralisation

Classification par organisation des concepts

- inclusion des extensions
- héritage et raffinement des intensions

`Concept Carré` spécialise `Concept Rectangle`

- Point de vue extensionnel : l'ensemble des carrés est inclus dans l'ensemble des rectangles
- Point de vue intensionnel : les propriétés des rectangles s'appliquent aux carrés et se spécialisent (largeur = hauteur)

Héritage et spécialisation/généralisation

Cas d'étude

- Contexte d'une agence immobilière
- Gestion de la location d'appartement

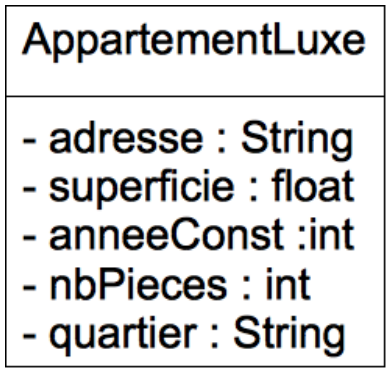
Appartement

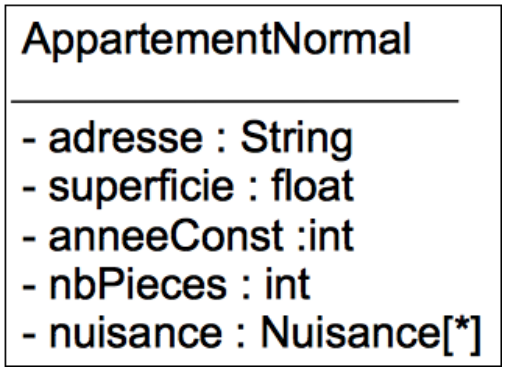
- tous sont décrits par : une adresse, une année de construction, une superficie, un nombre de pièces
- deux sous-catégories nous intéressent
  - les appartements de luxe décrits en plus par leur quartier
  - les appartements normaux décrits en plus par les nuisances de leur environnement

Sans spécialisation/généralisation

Solution 1

Réaliser deux classes





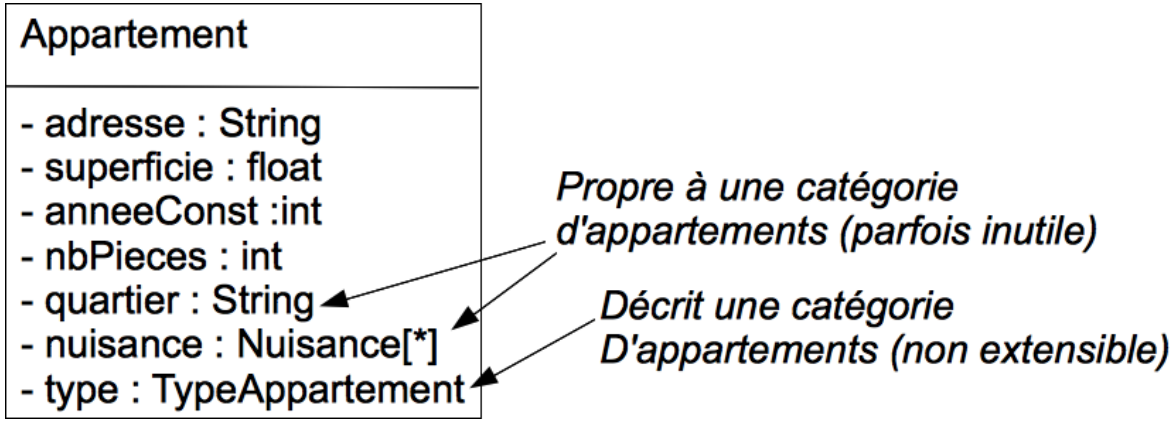
Inconvénients

- répétition des attributs et de parties dans le code des méthodes
- risque d'incohérence (sur les parties communes) entre les différentes sortes d'appartements
- toute modification sur les parties communes demandera d'intervenir à plusieurs endroits (avec de nouveaux risques d'erreurs)

Sans spécialisation/généralisation

Solution 2

Réaliser une seule classe



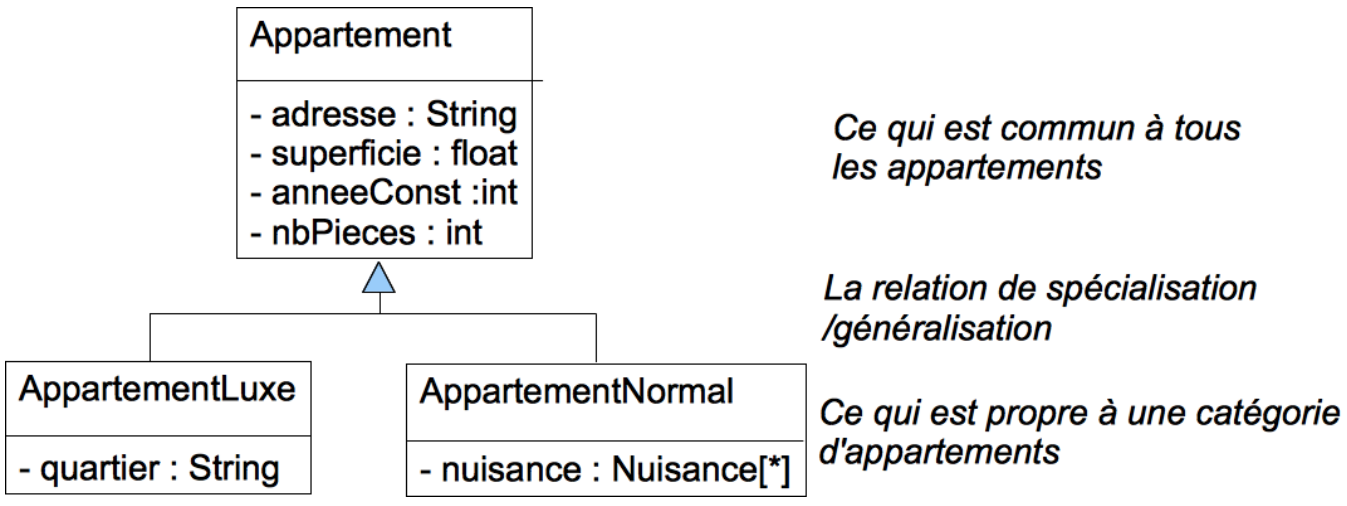
Inconvénients

- certains attributs (quartier, nuisance) ne sont pas toujours utilisés
- on ne peut pas ajouter facilement une sous-catégorie (ex. appartement de fonction)

La solution de la spécialisation/généralisation

Solution spécifique des approches à objets

Réaliser trois classes et les connecter par spécialisation



Avantages

- pas de répétition, pas de risque d'incohérence, pas d'attributs inutiles
- facile à étendre, par une nouvelle sous-classe

Quelques instances

a1 : AppartementLuxe
adresse = « 8, ch. des lilas, Mulhouse »
anneeconst = 1988
superficie = 150
nbPieces = 4
quartier = « La petite rivière »

a2 : AppartementNormal
adresse = « 10, rue H Berlioz, Mulhouse »
anneeconst = 1988
superficie = 40
nbPieces = 2
nuisance = {centreVille}

Précisions sur les relations de généralisation/spécialisation

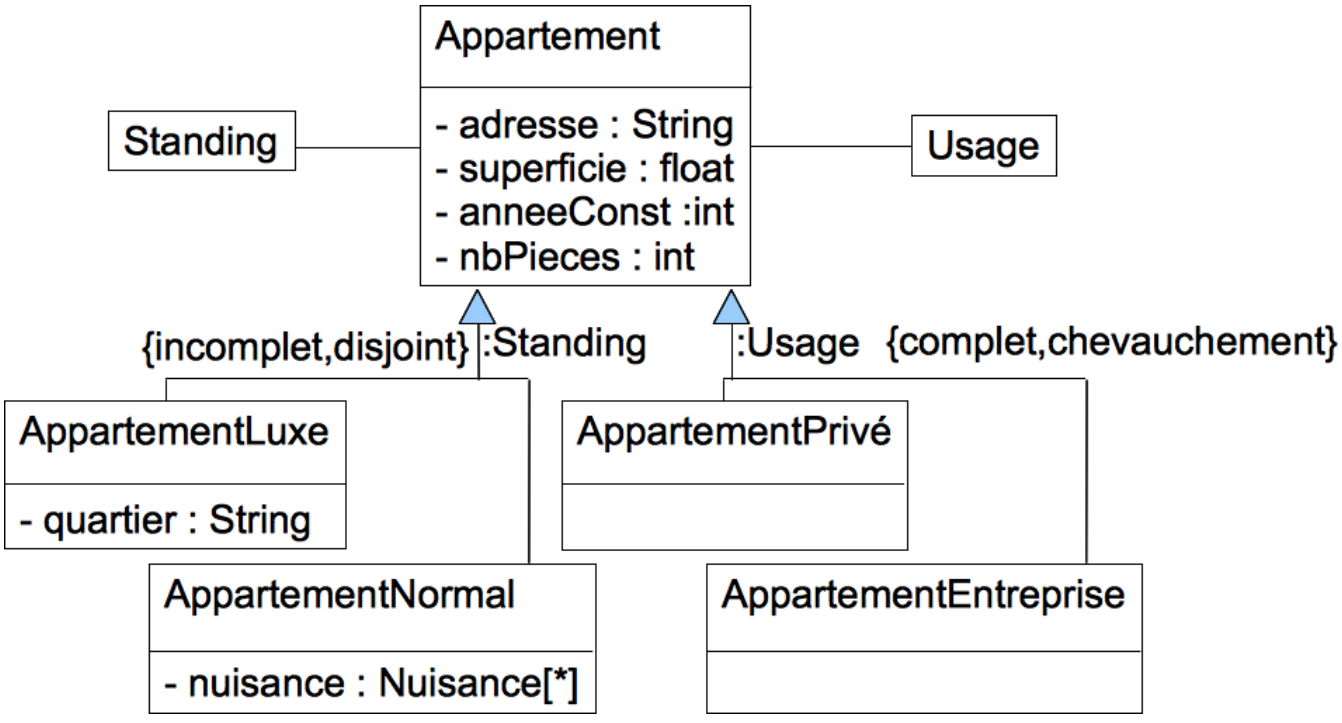
Discriminant

- critère de classification
- est représenté par une classe et par une annotation sur un ensemble de relations de spécialisation/généralisation

Contraintes

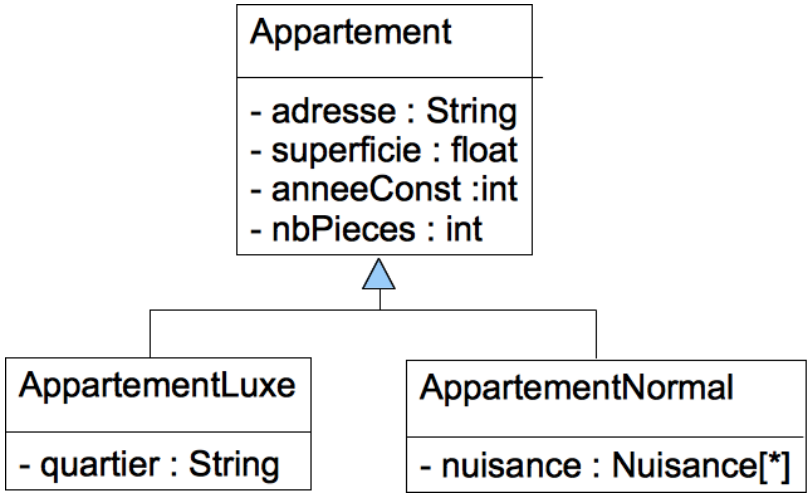
- annotent, entre accolades, un ensemble de relations de spécialisation/généralisation
- il en existe quatre (deux paires) :
  - complet, incomplet : les sous-classes couvrent (ne couvrent pas) l'ensemble des objets de la superclasse
  - disjoint, chevauchement : les sous-classes ne peuvent pas (peuvent) avoir d'instances communes

Discriminants et contraintes



- il existe des appartements autres que de luxe ou normaux
- un appartement ne peut être en même temps de luxe et normal
- un appartement est soit privé, soit pour une entreprise (pas d'autre cas)
- un appartement peut être utilisé à la fois pour un usage privé et pour un usage par une entreprise

Traduction en Java



Ce qui est commun à tous les appartements

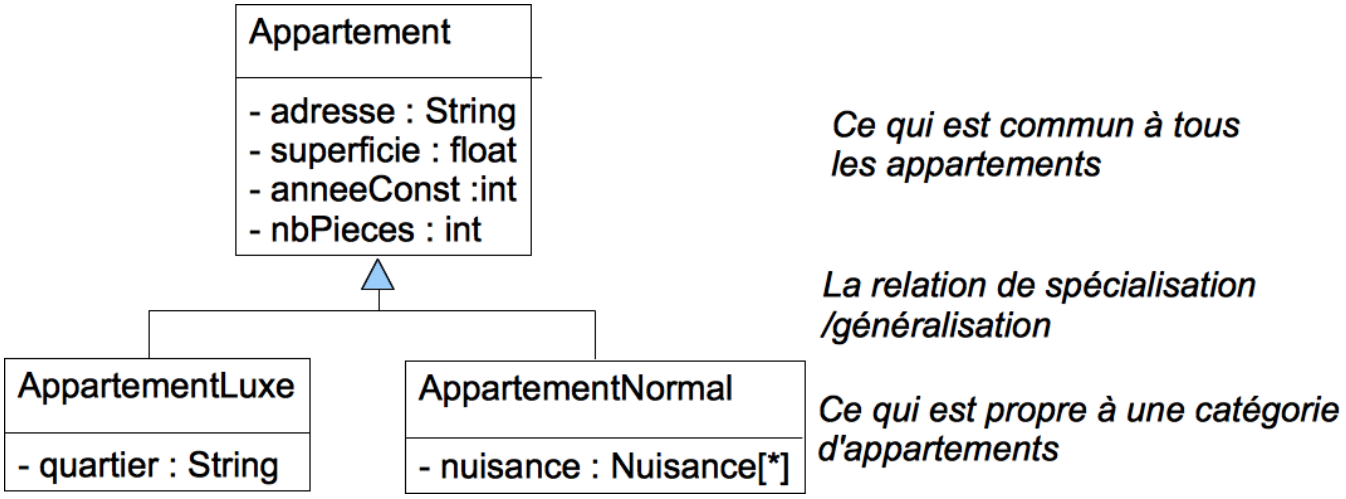
La relation de spécialisation /généralisation

Ce qui est propre à une catégorie d'appartements

Listing 1 – Classe Appartement / attributs

```
1  public class Appartement {
2  private String adresse;
3  private int anneeConst;
4  private float superficie;
5  private int nbPieces;
6  .....
7  }
```

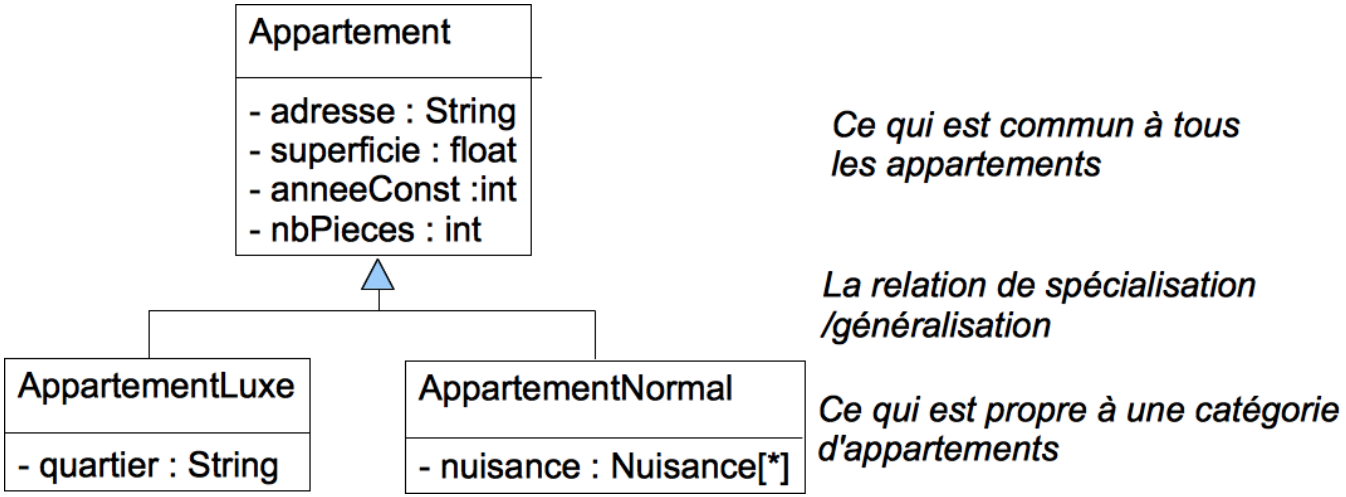
Traduction en Java



Listing 2 – Classe Appartement de luxe/ attributs

```
1  public class AppartementLuxe extends Appartement{
2  private String quartier;
3  .....
4  }
```

Traduction en Java



Listing 3 – Classe Appartement de normal / attributs

```
1  public class AppartementNormal extends Appartement{
2  private Nuisance[] nuisances;
3  .....
4  }
```

Traduction en Java

Listing 4 – enum Nuisance

```
1  public enum Nuisance{
2  autoroute, aeroport, dechetterie, centre_ville,
3  .....
4  }
```

Création d'objets

Listing 5 – Creation

```
1  AppartementLuxe a1 = new AppartementLuxe(...);
2  Appartement a2 = new AppartementLuxe(...);
3  Object a3 = new AppartementLuxe(...);
```

Les lignes 2 et 3 sont des affectations polymorphes :
la variable a un type (Appartement) différent du type de l'instance (AppartementLuxe)

Notez que l'affectation polymorphe se fait "en remontant" :
on affecte une instance de la sous-classe à une variable de la superclasse

`Object` est la super-classe implicite de toutes les classes en Java.

Constructeurs / dans la super-classe

Listing 6 – Constructeurs dans la super-classe

```
1  public class Appartement{
2  public Appartement()
3  {adresse = "inconnue"; anneeConst = 1970; superficie = 15;
       nbPieces = 1;}
4  public Appartement(String ad, int ac, float s, int n)
5  {adresse = ad; anneeConst = ac; superficie = s; nbPieces = n
       ;}
6  }
```

Règle

- Toujours écrire un constructeur sans paramètre
- Le constructeur initialise les attributs propres à la classe

Constructeurs / dans la sous-classe

Listing 7 – Constructeurs dans la sous-classe

```
1  public class AppartementLuxe extends Appartement{
2  public AppartementLuxe(){quartier = "inconnu";}
3  public AppartementLuxe(String ad, int ac, float s, int n,
       String q)
4  {super(ad, ac, s, n); quartier = q;}
5  }
```

Règle

- `super` : transmet les paramètres au constructeur de la super-classe
- L'appel au constructeur de la super-classe, s'il a lieu, est forcément la première instruction exécutable du constructeur.
- La super-classe doit disposer d'un constructeur admettant ces paramètres

Constructeurs / Appel

Listing 8 – Creation

```
1  Appartement a1 = new AppartementLuxe();
2  Appartement a2 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
```

Constructeurs exécutés

- les constructeurs de toutes les super-classes sont appelés du haut vers le bas
- Ligne 1 : Object() ; Appartement() ; AppartementLuxe()
- Ligne 2 : Object() ; Appartement(String ad, int ac, float s, int n) ; AppartementLuxe(String ad, int ac, float s, int n, String q)

Définition des méthodes

Trois configurations

- Héritage : la méthode est écrite dans une classe et accessible dans ses sous-classes
- Masquage : la méthode est écrite dans une classe et entièrement réécrite dans une sous-classe
- Spécialisation : la méthode est écrite dans une classe ; dans la sous-classe on fait référence à la méthode de la super-classe pour modifier légèrement le comportement initial

Héritage de méthode

Listing 9 – Héritage de méthode

```
1  public class Appartement {
2  ...
3  public float valeurLocativeBase()
4     {return superficie * 5 * (1 + nbPieces/n);}
5  }
6  public class AppartementLuxe extends Appartement {
7  ...
8  // rien qui concerne la valeur locative de base
9  }
10 ...
11 AppartementLuxe a1 = new AppartementLuxe("8, ch. Lilas, 
       Mulhouse", 1988, 150, 4, "La petite riviere");
12 System.out.println(a1. valeurLocativeBase());
```

Masquage de méthode

Listing 10 – Masquage de méthode

```
1  public class Appartement {
2  public String toString(){return "appt - adresse ="+ adresse
       + "superficie = "+superficie+" m2";}
3  }
4  public class AppartementNormal extends Appartement {
5  public String toString(){return " annee de construction = "+
       anneeConst;}
6  }
7  ...
8  Appartement a2 = new AppartementNormal("8, ch. Lilas , 
       Mulhouse", 1988, 150, 4, Nuisance.centre_ville);
9  System.out.println(a2.toString());
```

Ligne 11 : annee de construction = 1988

Spécialisation de méthode

Listing 11 – Spécialisation de méthode

```
1  public class Appartement {
2  ...
3  public String toString(){return "appt - adresse ="+ adresse
       + "superficie = "+superficie+" m2";}
4  }
5  public class AppartementLuxe  extends Appartement {
6  ...
7  public String toString(){return super.toString()+" quartier
       = "+quartier;}
8  }
9  ...
10 Appartement a3 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
11 System.out.println(a3.toString());
```

Ligne 11 : appt - adresse = 8, ch. Lilas, Mulhouse superficie = 150 m2 quartier = La petite riviere

Classes et méthodes abstraites

Loyer

Le loyer se calcule comme produit de :

- la valeur locative de base
- le coefficient modérateur

Coefficient modérateur

Le coefficient modérateur vaut :

- 1.1 pour les appartements de luxe
- 1 - 0.1*nombre de nuisances

La méthode calculant le coefficient modérateur est donc nécessaire pour écrire la méthode de calcul de loyer, mais elle ne peut être écrite de manière générale pour les appartements.

Classes et méthodes abstraites

Listing 12 – Classe et méthode abstraite

```
1  public abstract class Appartement {
2  ...
3  public abstract float coeff(); // pas de corps !!!
4  }
5  ...
6  public class AppartementNormal extends Appartement{
7  ...
8  public float coeff()
9  {return 1 - 0.1 * nuisances.lenght;}
10 }
11 ...
12 public class AppartementLuxe extends Appartement{
13 ...
14 public float coeff(){return 1.1;}
15 }
```

Classes et méthodes abstraites

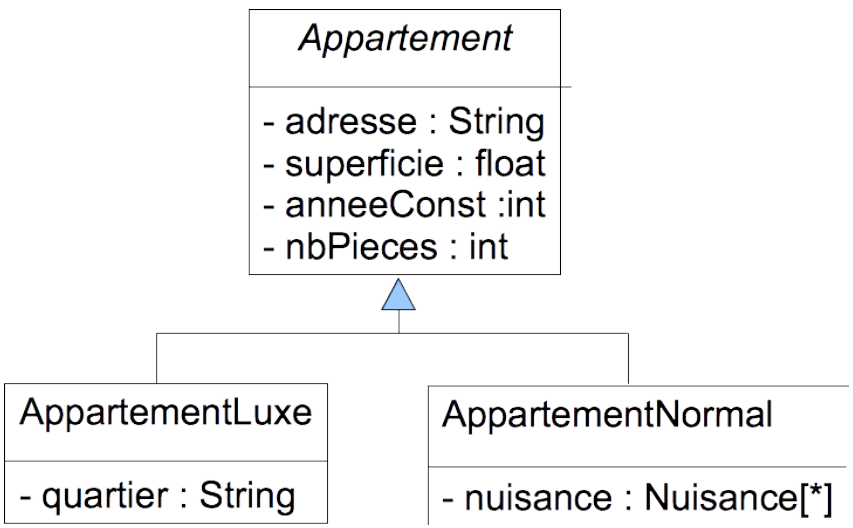
- la classe abstraite peut toujours servir de type pour des variables
- on ne peut plus créer d'instance de la classe abstraite (car sa définition est incomplète)

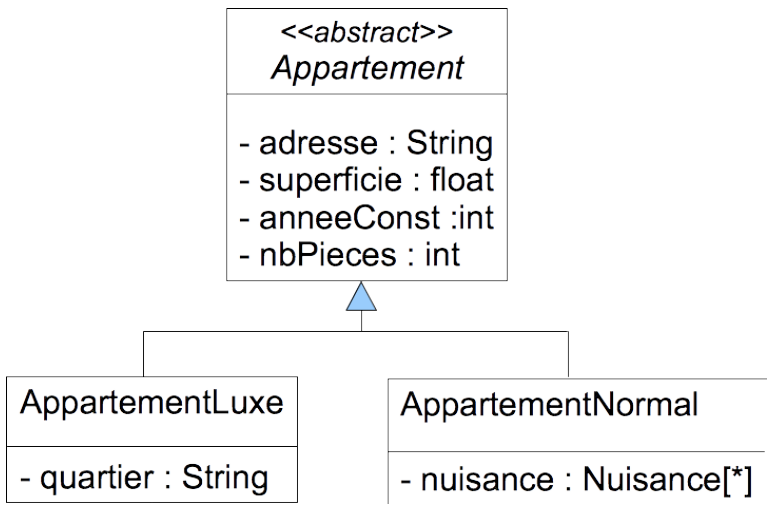
Listing 13 – Classe abstraite

```
1  Appartement a1 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
2  ...
3  // DEVIENT IMPOSSIBLE : Appartement a2 = new Appartement("8,
       ch. Lilas, Mulhouse", 1988, 150, 4);
```

Classes et méthodes abstraites

Représentation en UML (italiques, stéréotype)





Liaison dynamique

Choix d'une méthode à appeler

- Lorsqu'une méthode est appelée, elle est recherchée à partir de la classe de l'objet et en remontant vers ses superclasses jusqu'à la trouver
- On parle de liaison dynamique car le choix est réalisé lors de l'exécution

Listing 14 – Classe abstraite

```
1  Appartement a1 = new AppartementLuxe("8, ch. Lilas, Mulhouse
       ", 1988, 150, 4, "La petite riviere");
2  System.out.println(a1.loyer());
```

Exécute : loyer et valeurLocativeBase de Appartement ; coeff de AppartementLuxe