Spécialisation/généralisation et Héritage

Synthèse et bonnes pratiques

Approfondissement de la modélisation en UML
Affectation polymorphe
La classe Object
Liaison dynamique
Répartition des responsabilités

Faculté des Sciences / Université de Montpellier
Modélisation et programmation par objets 1

Spécialisation/généralisation et Héritage

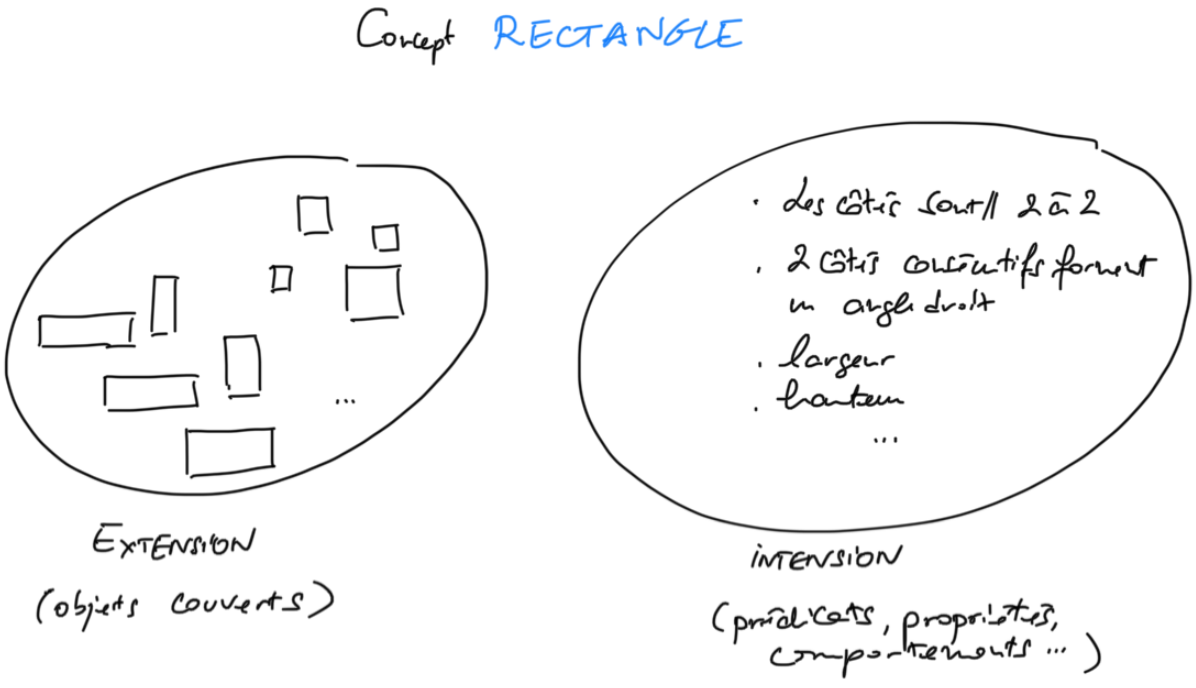
Les classes traduisent la notion de *concept*

- Un concept a une extension : ensemble des objets couverts
- Un concept a une intension : prédicats, caractéristiques des objets couverts

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.

Le concept Rectangle



Spécialisation/généralisation et Héritage

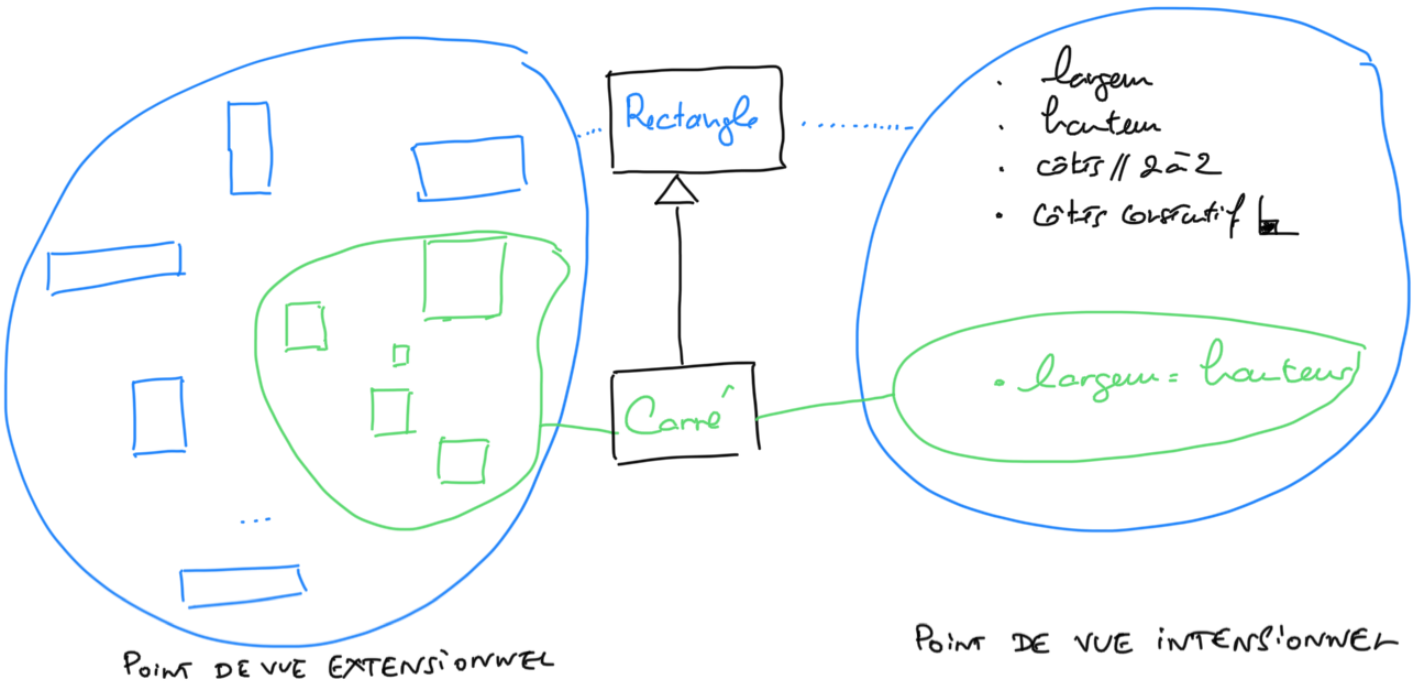
Les concepts s'organisent par spécialisation

- inclusion des extensions (base pour la modélisation, UML)
- héritage et raffinement des intensions (vision programmation, Java)

`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)

Carré et Rectangle



Spécialisation/généralisation et Héritage

Cas d'étude

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

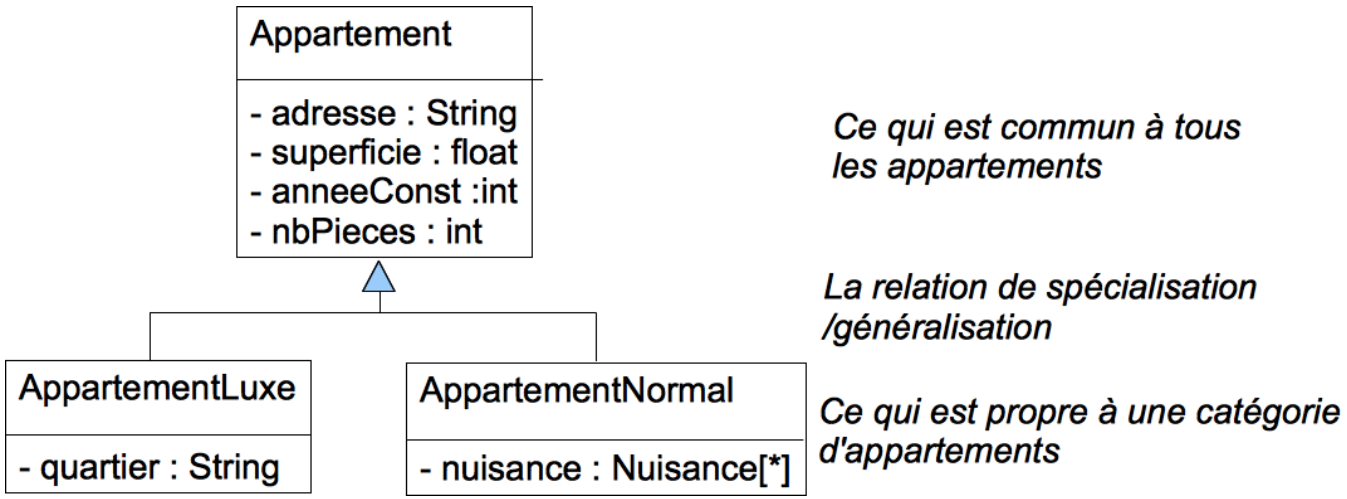
Appartement

- 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

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

Précisions sur les relations de généralisation/spécialisation

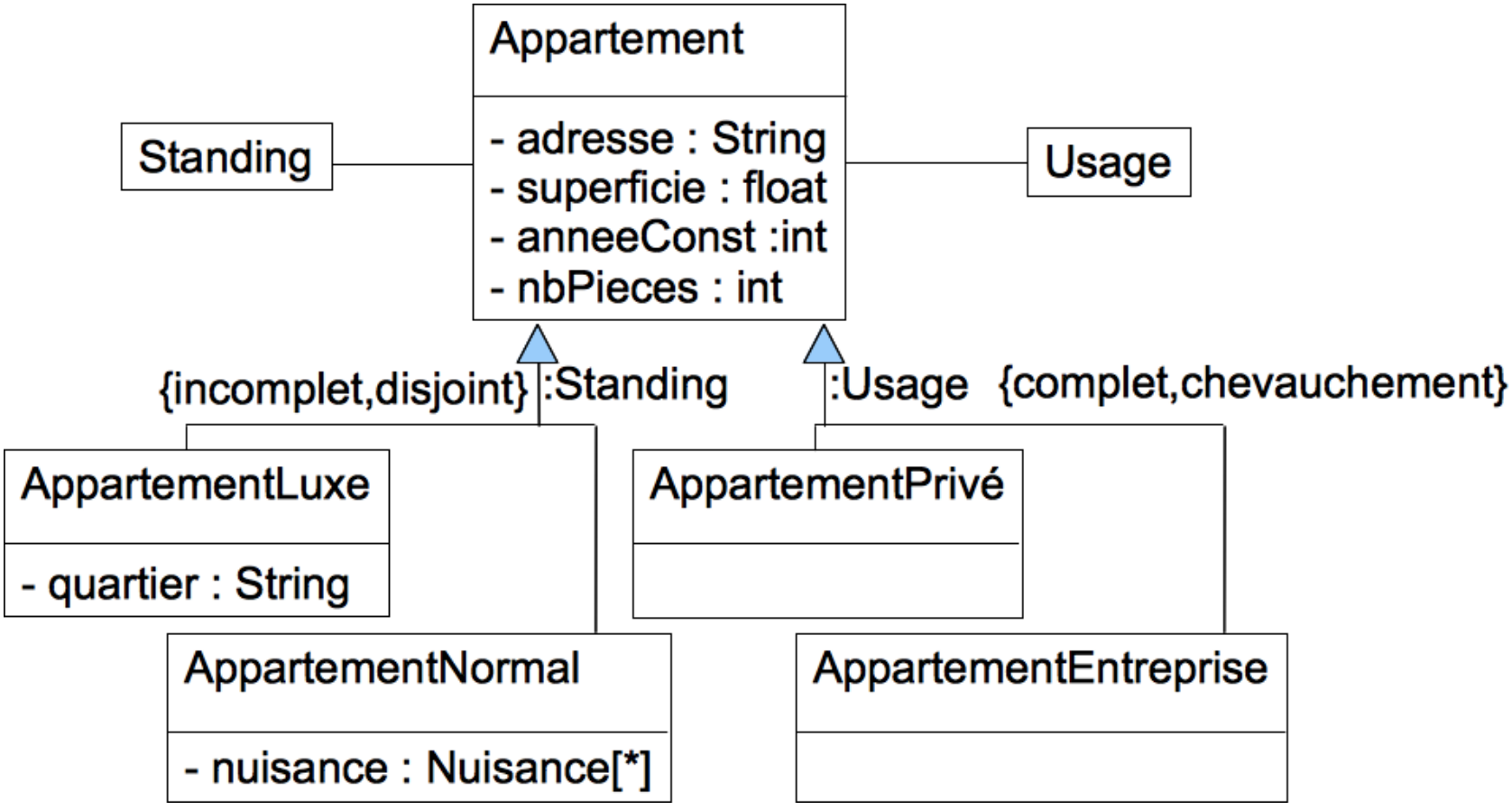
Discriminant

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

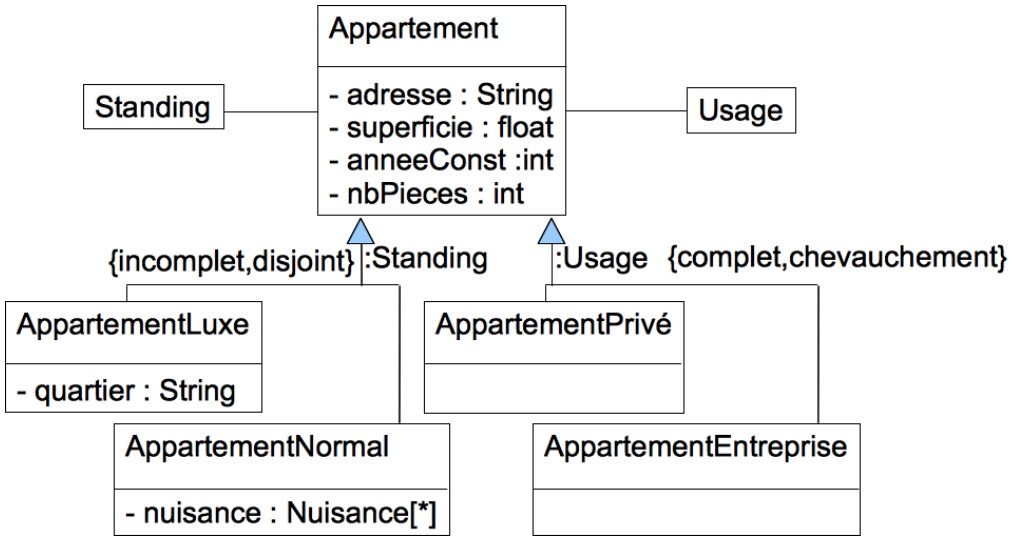
Contraintes

- annotent, entre accolades, un ensemble de relations de spécialisation/généralisation
- il en existe quatre (deux paires) :
  - complet, incomplet : les instances des 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



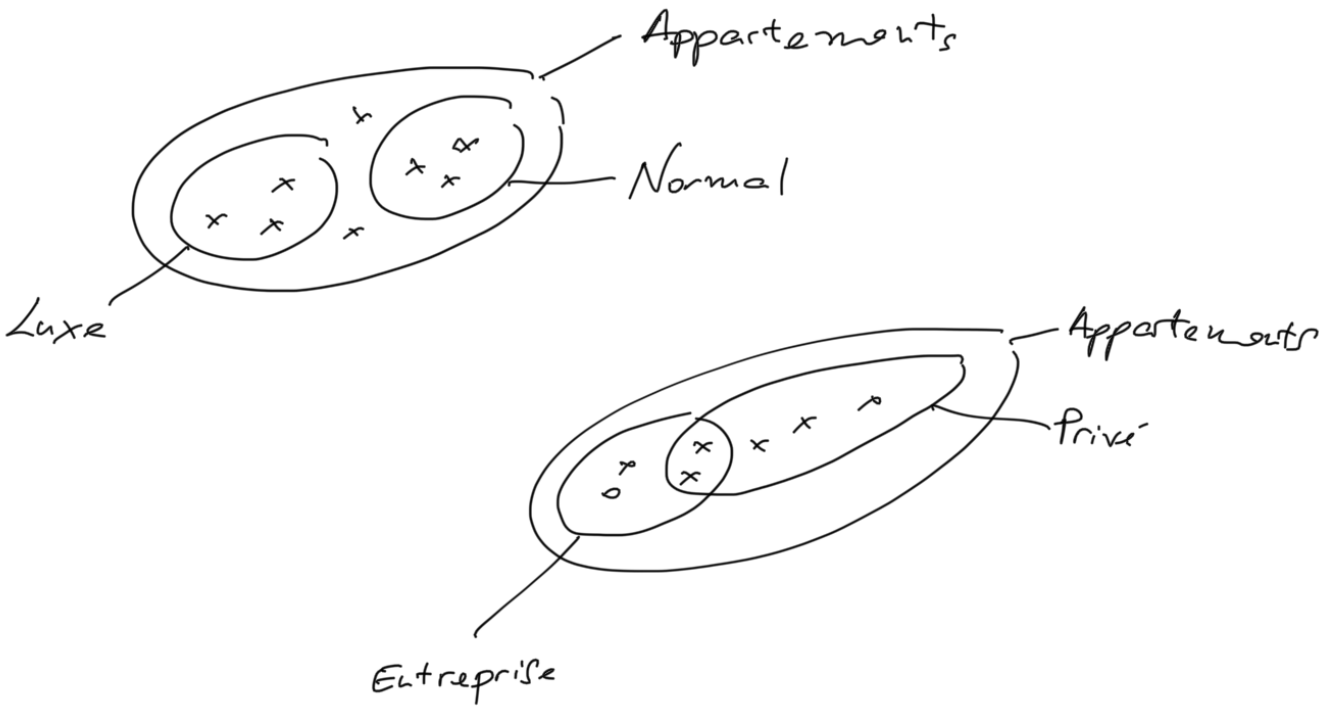
Discriminants et contraintes



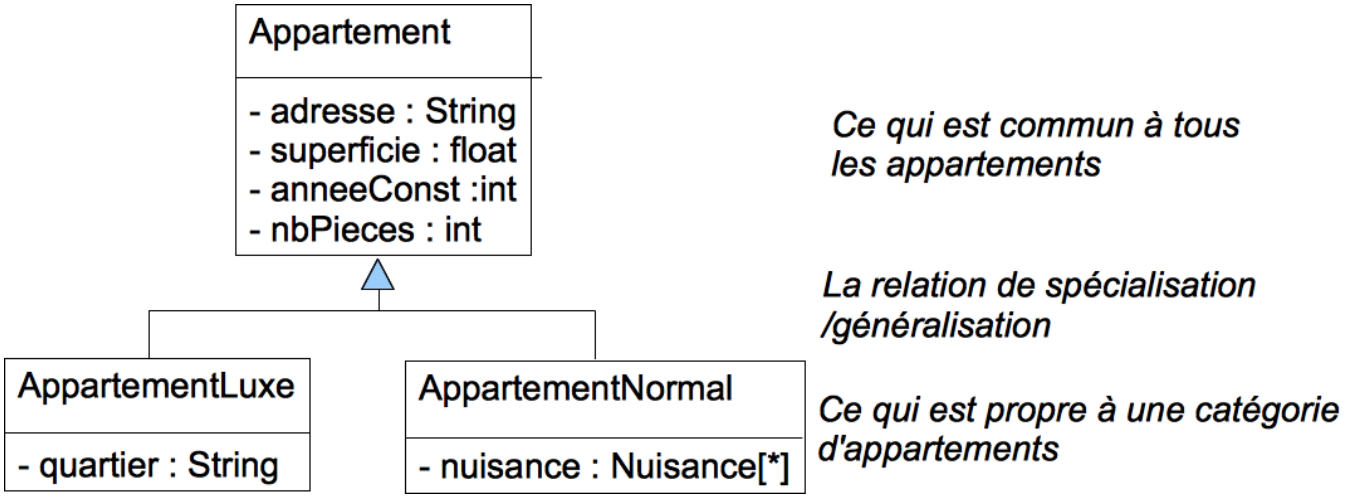
- (incomplet) il existe des appartements autres que de luxe ou normaux
- (disjoint) un appartement ne peut être en même temps de luxe et normal
- (complet) un appartement est soit privé, soit pour une entreprise
- (chevauchement) un appartement peut être utilisé à la fois pour un usage privé et pour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

Discriminants et contraintes

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



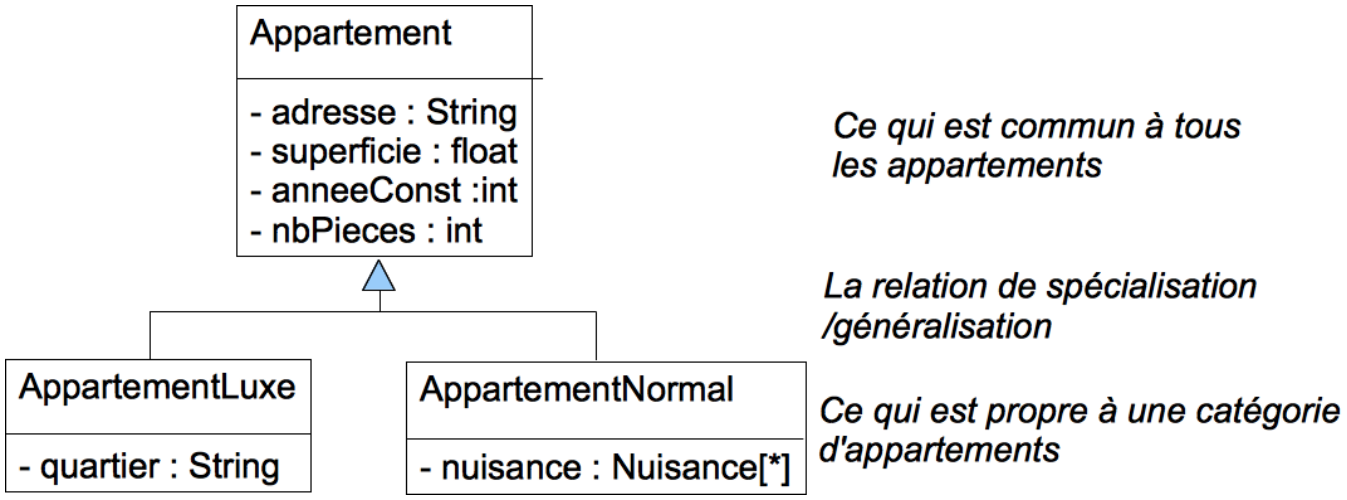
Traduction en Java



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



Discriminant et contraintes ne sont pas explicites en Java, ce sera géré dans les instructions

Listing 2 – Classe Appartement de luxe/ attributs

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

Affectation polymorphe

Listing 3 – 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 :
(l2) la variable a un type (Appartement) qui est super-type du type de l'instance (AppartementLuxe)
(l3) la variable a un type (Object) qui est super-type du type de l'instance (AppartementLuxe)

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

Classe Object

`Object` est la super-classe implicite de toutes les classes en Java
Elle propose des méthodes communes à toutes les classes : `equals`, `toString`

Listing 4 – equals et toString de Object

```
1     public static void main(String[] args) {
2         Object o1 = new Object();
3         Object o2 = new Object();
4
5         System.out.println(o1.toString()); //java.lang.
              Object@6e8dacdf
6         System.out.println(o1); //java.lang.Object@6e8dacdf
7         System.out.println(o1.equals(o2)); // false
8         o1 = o2;
9         System.out.println(o1.equals(o2)); // true
10    }
```

Classe Object

`Object` est la super-classe implicite de toutes les classes en Java
Elle propose des méthodes communes à toutes les classes : `equals`, `toString`

Listing 5 – equals et toString de Object

```
1  public class Rectangle {
2      private double largeur, hauteur;
3      public Rectangle(double l, double h)
4      {this.largeur=l; this.hauteur=h;}
5
6      public static void main(String[] args) {
7          Object o1 = new Rectangle(2,3);
8          Object o2 = new Rectangle(2,3);
9
10         System.out.println(o1.toString()); //coursHeritage.
               Rectangle@6e8dacdf
11         System.out.println(o1); // coursHeritage.
               Rectangle@6e8dacdf
12
13         System.out.println(o1.equals(o2)); // false
14     }
15 }
```

Classe Object

`Object` est la super-classe implicite de toutes les classes en Java
Ses méthodes peuvent être redéfinies !

Listing 6 – Redéfinition de equals et toString

```
1  public class Rectangle {
2      private double largeur, hauteur;
3      public Rectangle(double l, double h)
4      {this.largeur=l; this.hauteur=h;}
5      public String toString() {
6          return "h="+this.hauteur+"l="+this.largeur;}
7      public boolean equals(Rectangle r) {
8          return this.hauteur== r.hauteur&& this.largeur== r.
               largeur;}
9
10     public static void main(String[] args) {
11         Object o1 = new Rectangle(2,3);
12         Object o2 = new Rectangle(2,3);
13         System.out.println(o1.toString()); //h=3.0l=2.0
14         System.out.println(o1); // h=3.0l=2.0
15         System.out.println(o1.equals(o2)); // true
16     }
17 }
```

Définition des méthodes

Quatre principaux schémas

- 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
- Généralisation : la méthode appelle des méthodes qui sont elles-même spécialisées dans les sous-classes

Héritage de méthode

Listing 7 – 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 8 – 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 9 : annee de construction = 1988

Spécialisation de méthode

Listing 9 – Spécialisation de méthode avec super.nomMeth()

```
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

Définition par généralisation

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 10 – Définition de coeff

```
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   {
10     return 1 - 0.1 * nuisances.length;
11    // en supposant moins de 10 nuisances
12  }
13 }
14 ...
15 public class AppartementLuxe extends Appartement{
16 ...
17  public float coeff(){return 1.1;}
18 }
```

Classes et méthodes abstraites

Listing 11 – Loyer est définie par généralisation

```
1  public abstract class Appartement {
2  ...
3  // la methode loyer appelle la methode abstraite coeff
4      public double loyer(){
5       return valeurLocativeBase()* coeff();
6      }
7  }
8  ...
9  public class AppartementNormal extends Appartement{
10   ...
11 // pas de methode loyer
12 }
13 ...
14 public class AppartementLuxe extends Appartement{
15 ...
16 // pas de methode loyer
17 }
```

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 trouver une méthode de signature adaptée
- On parle de liaison dynamique car le choix est réalisé lors de l'exécution
- `Appartement` : type statique, type de la variable, pour le compilateur, sert à vérifier que la méthode existe
- `AppartementLuxe` : type dynamique, type de l'instance, pour l'interprète, sert à choisir la forme de méthode exécutée

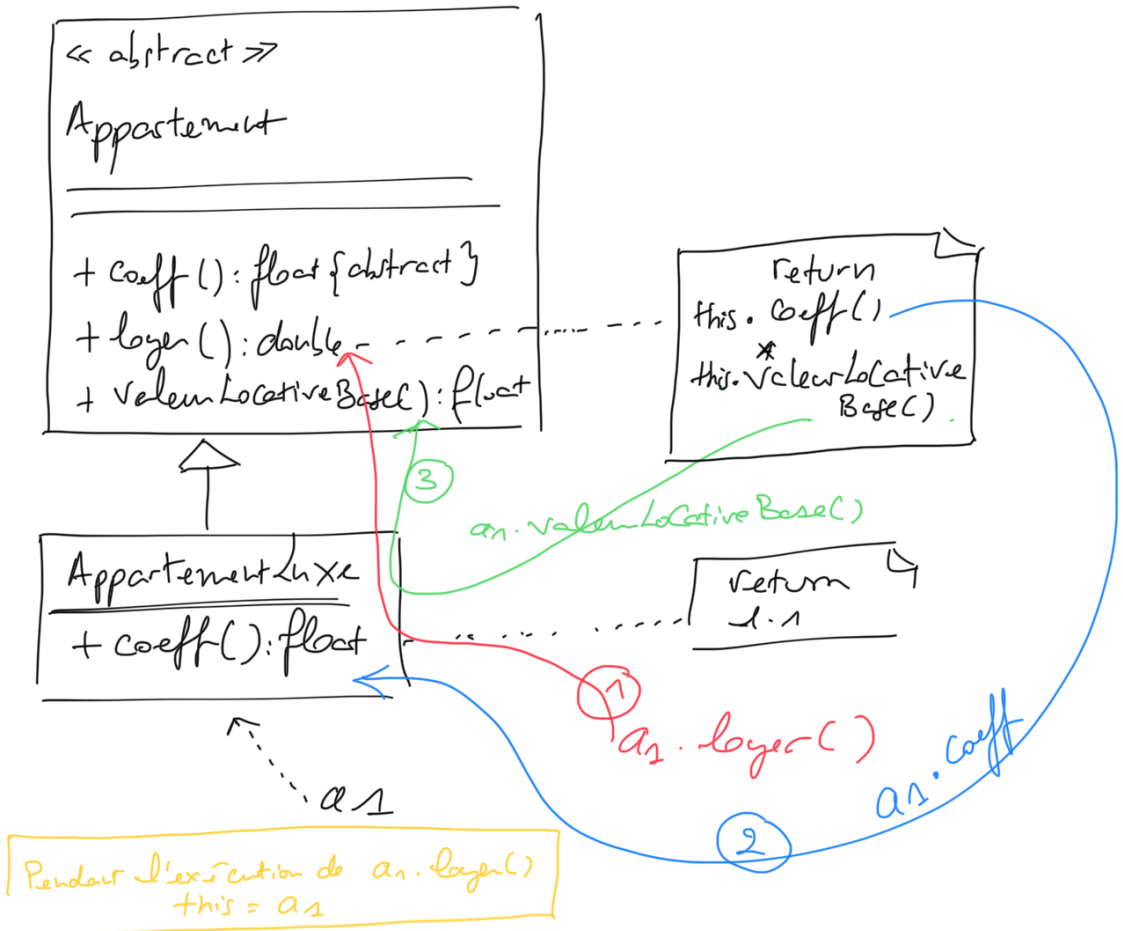
Listing 12 – Classe abstraite

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

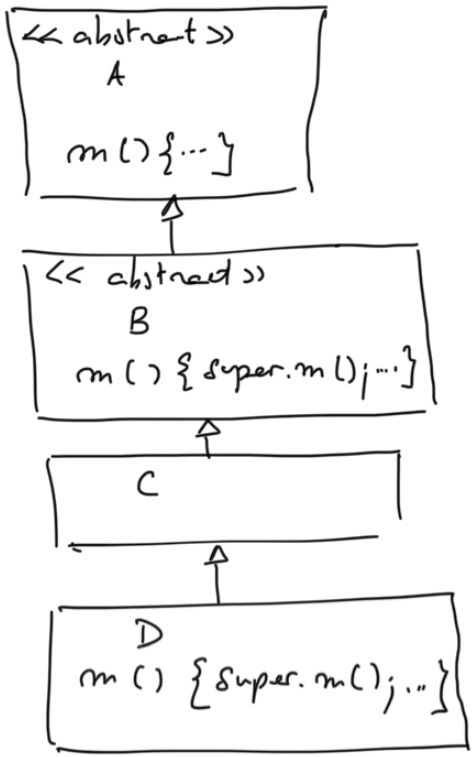
Liaison dynamique

Choix des méthodes 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



Les chaînes de spécialisation de méthodes peuvent sauter des étages

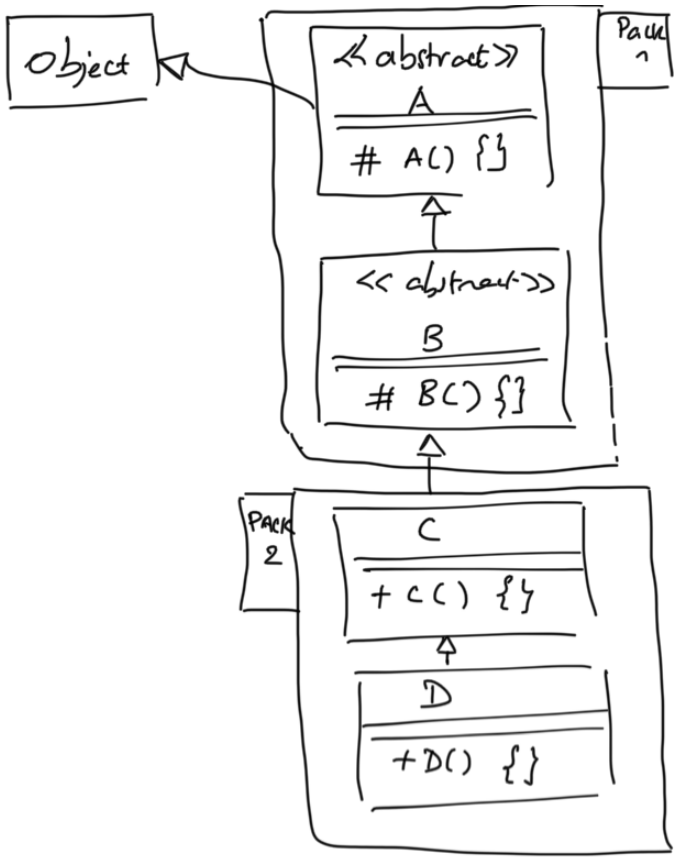


La méthode m n'a pas besoin d'être présente à tous les niveaux (Contrairement aux constructeurs)

```
A obj = new D();
obj.m();
```

>> m dans D
super.m()
>> m dans B
super.m
>> m dans A

Les chaînes de constructeurs ne sautent pas d'étage



Dans un main ...

```
A obj1 = new D();
```

>> Object() A() B() C() D()

```
D obj2 = new D();
```

>> Object() A() B() C() D()

- CHAQUE CLASSE X DOIT POSSÉDER UN CONSTRUCTEUR
- S'IL N'Y EN A AUCUN LE COMPILATEUR INSÈRE `X() {}`
- LE CONSTRUCTEUR COMMENCE TOUJOURS PAR UN APPEL À UN CONSTRUCTEUR DE SA SUPERCLASSE. SI LE PROGRAMMEUR NE L'A PAS ÉCRIT, LE COMPILATEUR INSÈRE `super();`

Répartition des responsabilités

Listing 13 – Répartition des responsabilités.
Chaque classe s'occupe de ses spécificités, ici ses attributs.

```
1  public abstract class Appartement {
2  ...
3  public String toString(){return "appt - adresse ="+ adresse
       + "superficie = "+superficie+" m2";}
4     }
5  }
6  ...
7  public class AppartementNormal extends Appartement{
8    ...
9    public String toString(){return super.toString()+" nombre
         de nuisances = "+nuisances.length;}
10 }
11 ...
12 public class AppartementLuxe extends Appartement{
13 ...
14 public String toString(){return super.toString()+" quartier
       = "+quartier;}
15 }
```

Répartition des responsabilités

Listing 14 – Mauvaise répartition des responsabilités

```
1  public abstract class Appartement {
2   public String toString(){
3    String res= "appt-adr␣="+adresse+"sup␣=␣"+superficie+"m2";
4    if (this instanceof AppartementNormal)
5     res+="␣nombre␣de␣nuisances="+((AppartementNormal)this).
          getNbNuisances();
6    else
7     res+="␣quartier="+((AppartementLuxe)this).getQuartier();
8    return res;     }
9  }
10 public class AppartementNormal extends Appartement{
11  public int getNbNuisances() {return this.nuisances.length;}
12 }
13 public class AppartementLuxe extends Appartement{
14  public String getQuartier() {return this.quartier;}
15 }
```

Ce code est un exemple de ce qu'il ne faut pas faire : il ne sera pas extensible ; il sera difficile à maintenir

Eléments de test de type et de coercition

- test de type `instanceof`
  - `o instanceof T` retourne vrai si `o` a pour type dynamique (par *new*) la classe `T` ou l'une de ses sous-classes
- coercition `((T)o)`
  - on indique au compilateur qu'il doit considérer l'objet `o` comme étant du type `T`
  - si `o` n'a pour type dynamique (par *new*) ni la classe `T` ni une de ses sous-classe, cela provoquera une erreur à l'exécution `ClassCastException`

Ces éléments sont à utiliser seulement dans des cas très restreints de redéfinition

Exemple typique où c'est autorisé, redéfinir :
`boolean equals(Object o)`, comme vu dans la classe Rectangle plus haut

Synthèse

- spécialisation/généralisation en UML
  - contraintes
  - discriminant
  - héritage multiple (une classe peut avoir plusieurs super-classes directes)
  - multi-instanciation (un objet peut avoir plusieurs classes incomparables)
- héritage en Java
  - `extends`
  - `super` pour transmission de paramètres ou appel de la méthode spécialisée
  - héritage simple sur les classes
  - mono-instanciation
  - liaison dynamique
  - test de type et coercition (mais le moins possible)
  - répartition des responsabilités