

Héritage et spécialisation/généralisation

Faculté des sciences, Université de Montpellier

Module HA8403I

Programmation par objets

Héritage et spécialisation/généralisation

Héritage et spécialisation/généralisation

- Exprimer des hiérarchies de classification entre concepts
 - D'un domaine métier : comptes bancaires, appartements, rectangles/carrés
 - Techniques : menus, boutons, structures de données
- Intérêt
 - Structurer
 - Réduire
 - Description par **genre** et **différence/spécificité**
 - Ex.1 Un carré est un **rectangle** dont tous les côtés sont égaux
 - Ex.2 Un e-commerce est un **commerce** qui vend en ligne (sur un site web)
 - Réutiliser
 - Faciliter l'extension

Spécialisation/généralisation et Héritage

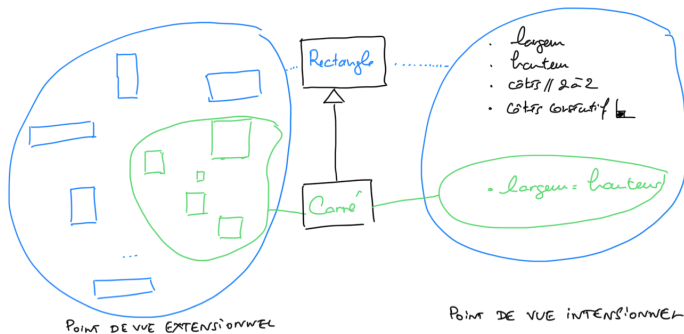
Les concepts s'organisent par spécialisation

- 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



Héritage et spécialisation/généralisation

Cas d'étude

L'agence immobilière gère des appartements de plusieurs catégories

- appartement
 - adresse
 - superficie
 - nombre de pièces
- appartement normal
 - avec des nuisances dans leur environnement
 - nuisance = industrie polluante, boulevard bruyant, terrain inondable
 - on mémoriserait seulement le nombre de nuisances (entier)
- appartement de luxe
 - avec des services fournis
 - service $\in \{\text{internet, livraison, blanchisserie, ménage, ...}\}$
 - on les mémoriserait dans une chaîne de caractères

Solution 1 : une seule classe

Appartement
- adresse : String
- superficie : double
- nbPieces : int
- services : String
- nbNuisances : int
- type : String

Inconvénients

- Certains attributs ne doivent pas être utilisés : risque d'utilisation par un programmeur inattentif ou mal informé, occupation mémoire inutile
- Le code est complexe car il doit prévoir les différentes catégories
 - Il utilise des tests sur la catégorie traitée
- L'ajout d'une nouvelle catégorie demande de modifier la classe
 - ajout d'attributs
 - modification des méthodes pour ajouter le cas de la nouvelle catégorie
 - du code qui fonctionnait déjà peut devenir corrompu

Solution 2 : deux classes

AppartementNormal

- adresse : String
- superficie : double
- nbPieces : int
- nbNuisances : int

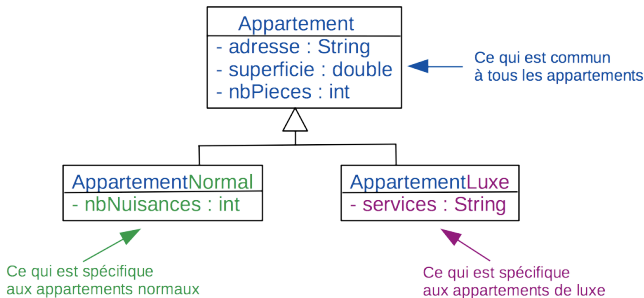
AppartementLuxe

- adresse : String
- superficie : double
- nbPieces : int
- services : String

Inconvénients

- Répétition des attributs et de parties de codes dans les méthodes
- Risque d'incohérence (sur les parties communes) entre les différentes sortes d'appartements
 - Ex. 'nombre de pièces' orthographié *nbPieces* ou *nombreDePieces*
 - Ex. type de l'adresse *String* (en un seul morceau) ou *ArrayList<String>* en séparant l'adresse en plusieurs parties
- Risque d'erreurs lors des modifications car il faudra intervenir à plusieurs endroits

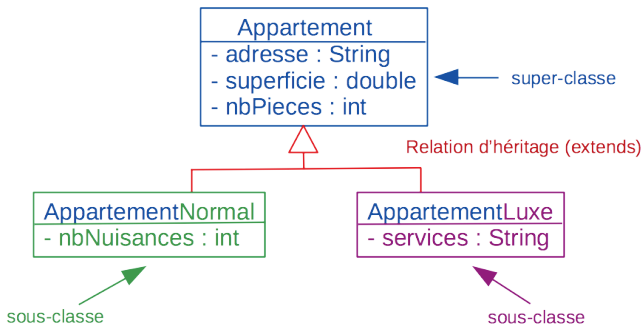
Solution 3 : trois classes



Avantages

- Economie : Définition par **genre** et **différence/spécificité**
- Cohérence : pas de répétitions, pas de tests de type dans les méthodes
- Extension facilitée : par ajout d'une nouvelle classe si besoin (Ex. Appartement de fonction)

Solution 3 : trois classes



Solution 3 : forme d'une instance d'une sous-classe

Une instance d'appartement de luxe dispose des attributs de la super-classe (**Appartement**) et des attributs de la sous classe (**AppartementLuxe**)

<u>a1 : AppartementLuxe</u>
adresse = "Montpellier" superficie = 100 nbPieces = 5 services = "internet piscine"

Précisions sur les relations de généralisation/spécialisation

Discriminant

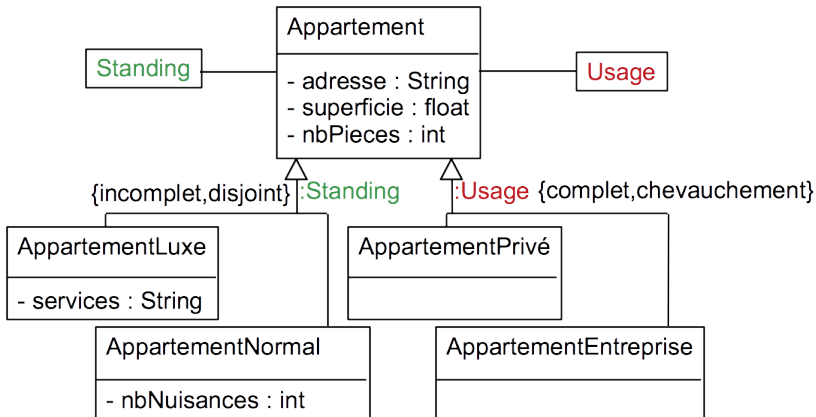
- C'est un critère de classification
- Il est représenté par une classe et par une annotation sur un ensemble de relations de spécialisation/généralisation

Contraintes

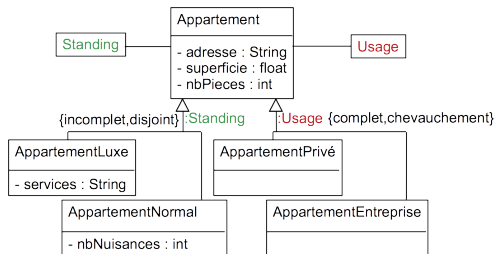
- Elles annotent, entre accolades, un ensemble de relations de spécialisation/généralisation
- Il en existe quatre (deux paires) :
 - complet, incomplet : les instances des sous-classes couvrent (ne couvrent pas) l'ensemble des objets de la superclasse
 - *complete, incomplete*
 - disjoint, chevauchement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

Discriminants et contraintes

Les appartements peuvent être spécialisés selon leur standing, ou selon leur usage ; le discriminant apparaît comme une classe et comme une annotation derrière le caractère `:`



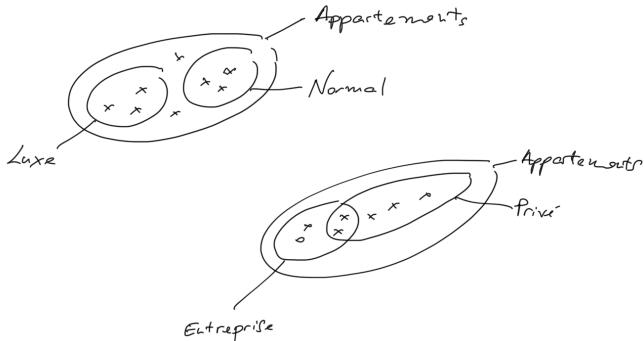
Discriminants et contraintes



- **(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 utilisé de manière privée, ou par une entreprise (pas d'autre possibilité)
- **(chevauchement)** un appartement peut être utilisé à la fois pour un usage privé et pour un usage par une entreprise

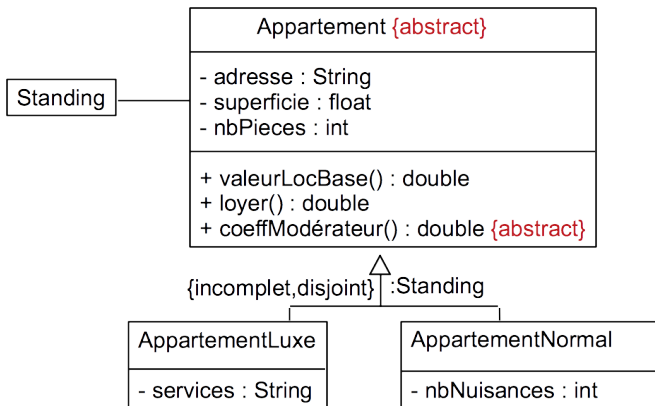
Discriminants et contraintes

- (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 utilisé de manière privée, ou par une entreprise (pas d'autre possibilité)
- (chevauchement) un appartement peut être utilisé à la fois pour un usage privé et pour un usage par une entreprise



Classe abstraite

Supposons que la classe *Appartement* n'admette pas d'instance directe : les instances seront des instances de ses sous-classes. Elle est dite abstraite. Son nom figure en italiques sur le schéma UML ou suivi du mot-clef **abstract**.

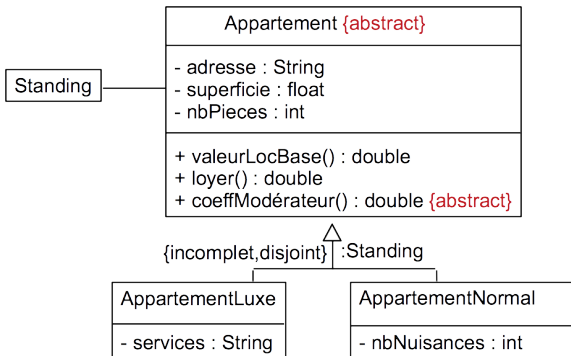


Méthode abstraite

Supposons que le calcul du loyer consiste à multiplier une valeur locative de base par un coefficient modérateur. Le coefficient modérateur se calcule ainsi :

- 1.1 pour les appartements de luxe
- $1 - 0.1 \times nbNuisances$ pour les appartements normaux

La méthode sera indiquée dans la classe `Appartement`, sous une forme abstraite (pas d'instruction), avec le mot-clef **abstract**.



Passons du côté de la programmation en Java : Classe de base

Au départ nous ne faisons pas une classe abstraite ...

```
public class Appartement {  
    // Attributs  
    private String adresse = "adresse inconnue";  
    private double superficie;  
    private int nbPieces;  
    .....  
}
```

Toute classe est implicitement sous-classe de la classe **Object** de l'API

```
public class Appartement extends Object {  
    // Attributs  
    private String adresse = "adresse inconnue";  
    private double superficie;  
    private int nbPieces;  
    .....  
}
```

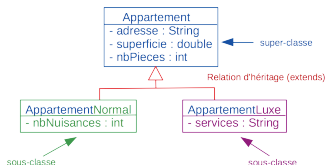
Classe **Object**

Toute classe est implicitement sous-classe de la classe **Object** de l'API
Sauf **Object** elle-même !

On ne l'écrit pas mais on utilise ses méthodes. Que peut-on trouver à l'intérieur ?

```
... public class Object {  
    // quelques méthodes  
  
    public String toString(){ ... }  
    // Returns a string representation of the object.  
  
    public boolean equals(Object obj){ ... }  
    // Indicates whether some other object is "equal to" this one.  
  
    .....  
}
```

Sous-classes



```

public class AppartementLuxe extends Appartement {
    // Attributs
    // adresse, superficie, nbPieces sont hérités
    // on ne les réécrit pas
    // on introduit le nouvel attribut

    private String services = "aucun service" ;
    .....
}
  
```

Exercice

Ecrire l'entête et l'attribut de la classe **AppartementNormal**

Forme des instances

Forme des instances de la classe **AppartementLuxe**

	adresse	<i>hérité</i>
	superficie	<i>"</i>
	nbPieces	<i>"</i>
	services	<i>spécifique</i>

Exercice

Décrire la forme des instances de la classe **AppartementNormal**

Constructeurs

Constructeurs de base : ceux de la classe `Appartement`

```
public class Appartement {  
  
    public Appartement () {}  
  
    public Appartement (String adresse, double superficie, int nbPieces) {  
        this.setAdresse(adresse);  
        this.setSuperficie(superficie);  
        this.setNbPieces(nbPieces);  
    }  
  
    .... }  
}
```

Constructeurs

```
public class Appartement {  
.....  
    public Appartement (String adresse, double superficie, int nbPieces) {...}  
.....  
}
```

Constructeur avec paramètres de la sous-classe **AppartementLuxe** :

- Placer des paramètres pour les attributs **hérités** et les attributs **spécifiques**
- Appel d'un constructeur de la super-classe directe avec **super(...)**
- Initialisation des attributs spécifiques

```
public class AppartementLuxe extends Appartement {  
  
    public AppartementLuxe(String adr, double sup, int nbP, String serv) {  
        super(adr, sup, nbP); // Appel constructeur super-classe  
        this.setServices(services); // Initialisation attribut spécifique  
    }  
  
}
```

Constructeurs

Quelques règles pour les constructeurs :

- Sauf exception **prévoir un constructeur sans paramètres**, car il sera appelé implicitement par tout constructeur d'une sous-classe qui n'appellerait pas un constructeur avec paramètre.
- Chaque classe s'occupe d'initialiser ses attributs spécifiques et transmet à un constructeur de la super-classe des valeurs pour les attributs hérités (c'est une nouvelle application du **principe de responsabilité des classes**).

Ordre d'exécution des constructeurs **du haut vers le bas** :

- Object() (même si on ne l'a pas appelé explicitement)
- **Appartement**(...)
- **AppartementLuxe**(...)

Exercice

Ecrire un constructeur avec paramètres pour la classe **AppartementNormal**, qui initialiserait tous les attributs.

Créer des objets

"Montpellier"	adresse	<i>hérité</i>
100	superficie	"
5	nbPieces	"
"internet piscine"	services	<i>spécifique</i>

```
public class MainAppartement {
    public static void main(String[] args) {
```

```
        AppartementLuxe a1
```

```
            = new AppartementLuxe ("Montpellier", 100, 5 ,"internet piscine");
```

```
    }
```

```
    .....
```

```
}
```

Exercice

Créer une instance de la classe **AppartementNormal**

Affectation polymorphe

"Paris"	adresse	<i>hérité</i>
50	superficie	"
2	nbPieces	"
"conciergerie"	services	<i>spécifique</i>

Type de la variable (**Appartement**) \neq Type de l'objet (**AppartementLuxe**)

```
public class MainAppartement {
    public static void main(String[] args) {
        Appartement a2
            = new AppartementLuxe("Paris",50,2,"conciergerie");
    }
}
```

Appartement est le **type statique**, utilisé par le compilateur

AppartementLuxe est le **type dynamique**, utilisé par l'interprète, ou JVM (Java Virtual Machine)

Exercice

Créer une instance de la classe **AppartementNormal** avec affectation polymorphe

Affectation polymorphe

Type de la variable (**Appartement**) \neq Type de l'objet (**AppartementLuxe**)

```
public class MainAppartement {  
    public static void main(String[] args) {  
        Appartement a2  
            = new AppartementLuxe("Paris",50,2,"conciergerie");  
    }  
}
```

Règle

Le type de la variable doit être **un super-type** (une super-classe) du type dynamique.

Logique : un **AppartementLuxe** est un **Appartement** (mais pas le contraire).

Héritage et définition des méthodes

L'héritage permet plusieurs schémas de définition des méthodes, par jeu entre ce que l'on écrit dans une super-classe et ce que l'on écrit dans une sous-classe :

- 1 héritage
- 2 redéfinition avec **masquage**
- 3 redéfinition avec **spécialisation**
- 4 méthode **abstraite**
- 5 définition par **généralisation**

Points à comprendre :

- la **factorisation** de code
- la **liaison dynamique**

Compilation et interprétation

- Le compilateur a pour rôle de **vérifier que la méthode appelée sur une variable existe**. Il utilise le type statique (le type de la variable).
- L'interprète a pour rôle de **choisir la méthode la plus spécifique à appeler pour un objet**. Il utilise le type dynamique (le type de l'objet, indiqué derrière l'opérateur `new`).

(1) Héritage de méthode

La valeur locative de base se calcule pour tous les appartements de la même manière (en supposant moins de 10 pièces) : $\text{superficie} \times 5 \times (1 + \text{nbPieces}/10)$

```
public class Appartement {.....
    public double valeurLocativeBase() {
        return this.superficie * 5 * (1+this.nbPieces/10);
    }
    .....}
```

La classe **AppartementLuxe** ne contient rien !
Mais par héritage, elle dispose de `valeurLocativeBase`

```
public class MainAppartement {
    public static void main(String[] args) {
        AppartementLuxe a1 = new AppartementLuxe(...);
        System.out.println(a1.valeurLocativeBase()); //valeurLocativeBase d'Appartement
    }
}
```

(1) Héritage de méthode

La valeur locative de base se calcule pour tous les appartements de la même manière (en supposant moins de 10 pièces) : $\text{superficie} \times 5 \times (1 + \text{nbPieces}/10)$

```
public class Appartement {.....
    public double valeurLocativeBase() {
        return this.superficie * 5 * (1+this.nbPieces/10);
    }
    .....}
```

La classe **AppartementLuxe** ne contient rien !
Mais par héritage, elle dispose de `valeurLocativeBase`

```
public class MainAppartement {
    public static void main(String[] args) {
        Appartement a2 = new AppartementLuxe(...);
        System.out.println(a2.valeurLocativeBase()); //valeurLocativeBase d'Appartement
    }
}
```

(1) Héritage de méthode

La valeur locative de base se calcule pour tous les appartements de la même manière (en supposant moins de 10 pièces) : $\text{superficie} \times 5 \times (1 + \text{nbPieces}/10)$

```
public class Appartement {.....
    public double valeurLocativeBase() {
        return this.superficie * 5 * (1+this.nbPieces/10);
    }
    .....}
```

```
public class MainAppartement {
    public static void main(String[] args) {
        Object a3 = new AppartementLuxe(....); // a3 a pour type statique Object
        // On ne peut pas compiler l'instruction suivante :
        System.out.println(a3.valeurLocativeBase()); // Object n'a pas valeurLocativeBase!
    }
}
```

(2) Redéfinition par masquage

Principe : la méthode est définie dans la super-classe et complètement redéfinie dans la sous-classe (la méthode de la super-classe est masquée)

```
public class Appartement {.....  
    public String toString() {  
        return "Appartement d'adresse "+this.adresse;  
    }.....}
```

```
public class AppartementLuxe extends Appartement {.....  
    public String toString() {  
        return "Appartement de luxe qui a pour services "+this.services;  
    }....}
```


(2) Redéfinition par masquage

```
public class Appartement {.....  
    public String toString() {  
        return "Appartement d'adresse "+this.adresse;  
    }.....}
```

```
public class AppartementLuxe extends Appartement {.....  
    public String toString() {  
        return "Appartement de luxe qui a pour services "+this.services;  
    }....}
```

```
public class MainAppartement {  
    public static void main(String[] args) {  
        Appartement a2 = new AppartementLuxe(....);  
        System.out.println(a2.toString()); // affiche seulement le message et les services  
    } ....}
```

a2 a pour type statique **Appartement** : le compilateur vérifie que `toString` existe pour cette classe

a2 a pour type dynamique **AppartementLuxe** : l'interprète cherche la méthode `toString` à partir de la classe **AppartementLuxe** (si besoin en remontant dans les super-classes, ici ce n'est pas nécessaire)

(3) Redéfinition par spécialisation

On écrit d'une meilleure manière la méthode `toString`, en respectant la règle de responsabilité

```
public class Appartement {.....  
    public String toString() {  
        return "Appartement d'adresse "+this.adresse+" superficie "+this.superficie  
            +" nbPieces "+this.nbPieces;  
    }.....}
```

```
public class AppartementLuxe extends Appartement {.....  
    public String toString() {  
        return super.toString()+" services "+this.services;  
    }.....}
```

`super.toString()` provoque l'appel de `toString()` de `Appartement`

```
public class MainAppartement {  
    public static void main(String[] args) {  
        Appartement a2 = new AppartementLuxe(....);  
        System.out.println(a2.toString()); // affiche toutes les informations  
    } ....}
```

(3) Redéfinition par spécialisation (exercice)

```
public class Appartement {.....  
    public String toString() {  
        return "Appartement d'adresse "+this.adresse+" superficie "+this.superficie  
            +" nbPieces "+this.nbPieces;  
    }.....}  
  
public class AppartementLuxe extends Appartement {.....  
    public String toString() {  
        return super.toString()+" services "+this.services;  
    }.....}
```

Exercice

Ecrire toString pour la classe **AppartementNormal** sur ce modèle.

(3) Redéfinition par spécialisation (exercice)

Exercice

Ecrire des méthodes de saisie pour les trois classes en respectant la règle de partage des responsabilités.

(4) Méthode abstraite

Un coefficient modérateur intervient dans le calcul du loyer, son calcul est différent suivant les types d'appartements :

- 1.1 pour les appartements de luxe
- $1 - 0.1 \times nbNuisances$ pour les appartements normaux (on suppose ici que le nombre de nuisances ne peut dépasser 10)

On ne sait pas l'écrire dans la classe `Appartement`, on la déclare alors abstraite.

```
public ..... class Appartement {.....  
    abstract public double coeffModerateur(); // Noter qu'il n'y a pas de bloc {}  
    .....}
```

```
public class AppartementLuxe extends Appartement {.....  
    public double coeffModerateur() {  
        return 1.1;  
    }.....}
```

Exercice

Ecrire `coeffModerateur` pour la classe `AppartementNormal`.

(4) Classe abstraite

Une classe qui contient une méthode abstraite est forcément abstraite aussi (sa définition n'étant pas complète).

```
public abstract class Appartement {.....
    abstract public double coeffModerateur(); // Noter qu'il n'y a pas de bloc {}
    .....}
```

On ne peut pas créer d'instance d'une classe abstraite.

```
public class MainAppartement {
    public static void main(String[] args) {
        Appartement a2 = new AppartementLuxe(....);
        Appartement appt = new Appartement(....);
        System.out.println(a2.coeffModerateur());
    } ....}
```

Nota : Une classe peut être abstraite mais sans méthode abstraite à l'intérieur.

(5) Définition par généralisation

Une méthode appelle une autre méthode dont des versions spécifiques pourront être trouvées dans les sous-classes.

Le loyer se calcule comme le produit de la valeur locative de base par le coefficient modérateur.

```
public abstract class Appartement {.....  
  
    abstract public double coeffModerateur();  
  
    public double loyer(){  
        return this.valeurLocativeBase() * this.coeffModerateur();  
    }  
.....}
```

(5) Définition par généralisation

```
public abstract class Appartement {.....

    abstract public double coeffModerateur();

    public double loyer(){
        return this.valeurLocativeBase() * this.coeffModerateur();
    }
    .....}

public class MainAppartement {
    public static void main(String[] args) {
        Appartement a2 = new AppartementLuxe(.....);
        System.out.println(a2.loyer());    } ....}
```

Appelle :

- 1 loyer() de **Appartement** (car il n'y a pas de telle méthode dans **AppartementLuxe**)
- 2 valeurLocativeBase() de **Appartement** (car il n'y a pas de telle méthode dans **AppartementLuxe**)
- 3 coeffModerateur() de **AppartementLuxe** (car il y en a une dans **AppartementLuxe**)

(5) Définition par généralisation

Exercice

Compléter `toString` afin que le coefficient modérateur et le loyer soient également affichés.

Règle de redéfinition des méthodes

Règle de redéfinition d'une méthode pour la liaison dynamique

- même nom
- même liste de type de paramètres
- type de retour identique ou plus spécialisé
- visibilité identique ou plus large (ex. `protected` -> `public`)

```
public abstract class Appartement{  
    ...  
    public boolean estIdentiqueA(Appartement a) {  
        return this.getAdresse().equals(a.getAdresse());  
    }  
  
    protected Appartement copie() {  
        return null; // on ne peut pas instancier Appartement  
    }  
    ...  
}
```

Règle de redéfinition des méthodes

Règle de redéfinition d'une méthode pour la liaison dynamique

- même nom
- même liste de type de paramètres
- type de retour identique ou plus spécialisé
- visibilité identique ou plus large (ex. `protected` -> `public`)

```
public class AppartementLuxe extends Appartement{
    public boolean estIdentiqueA(Appartement a) {
        if (a instanceof AppartementLuxe){
            AppartementLuxe aConvertiEnLuxe = (AppartementLuxe)a;
            return super.estIdentiqueA(aConvertiEnLuxe)&&
                this.getServices().equals(aConvertiEnLuxe.getServices());
        }
        else return false;
    }
}
```

instanceof : vérifie la classe de l'objet (**AppartementLuxe**) : réalise un typecast

@Override : annotation recommandée devant la méthode, pour vérifier la règle de bonne redéfinition par le compilateur

Règle de redéfinition des méthodes

Exemple de **mauvaise** redéfinition : **le type du paramètre n'est pas le même**

Cela compile mais ne sera pas appelé par la liaison dynamique

C'est de la surcharge statique et non pas de la redéfinition (pas de liaison dynamique)

L'annotation `@Override` ne sera pas acceptée ici

```
public class AppartementLuxe extends Appartement{  
    ...  
    public boolean estIdentiqueA(AppartementLuxe a) {  
        return this.estIdentiqueA(a)&&  
            this.getServices().equals(a.getServices());  
    }  
    ...  
}
```

Exemple de spécialisation du type de retour

Exemple de **bonne** redéfinition :

- **même nom**
- **même liste de type de paramètres (vide)**
- **type de retour identique ou plus spécialisé**
(Appartement -> AppartementLuxe)
- **visibilité identique ou plus large**
(protected -> public)

```
public class AppartementLuxe extends Appartement{  
    ...  
    @Override  
    public AppartementLuxe copie() {  
        return new AppartementLuxe(this.getAdresse(),  
                                     this.getSuperficie(),  
                                     this.getNbPieces(), this.getServices());  
    }  
    ...  
}
```

Synthèse

- Spécialisation/généralisation, discriminants, contraintes
- **extends** : définition de sous-classe
- **super(...)** : appel du constructeur de la super-classe
- **super.meth()** : appel de la méthode `meth` de la super-classe
- **Liaison dynamique**
- Différents **schémas de définition** des méthodes
- Classes et méthodes **abstraites**
- Règle de **redéfinition** pour la liaison dynamique
- **instanceof** et typecast