|  |  |
| --- | --- |
| |  | | --- | |  | |
| C34-VM Programmation Orientée Objet 1  **MODULE 05**  **Sous-classes – classes abstraites - Interfaces**  Automne 2023 |

Table des matières

[1. Sous-classes et super-classes 2](#_Toc261928761)

[1.1 Syntaxe de sous-classes et super-classes 3](#_Toc2034083461)

[Utilisation des membres d’une classe parent 4](#_Toc501943272)

[Le mot clé super 4](#_Toc1113384119)

[1.2 La surcharge de méthodes en héritage 5](#_Toc575821774)

[1.3 Les constructeurs en héritage 6](#_Toc70169405)

[1.4 La redéfinition de méthodes (Overriding) 7](#_Toc1811355110)

[L05A\_Sous-classes - Formes 8](#_Toc1516372803)

[Retours sur le laboratoire L05A\_Sous-classes - Formes 9](#_Toc607342524)

[2. La classe DecimalFormat 9](#_Toc1549670412)

[L05B\_DecimalFormat 10](#_Toc1901626044)

[L05C\_Héritage - Maisons 10](#_Toc1103006868)

[L05D\_Héritage - Routes 10](#_Toc1241279352)

[3. Les classes abstraites 11](#_Toc640957579)

[3.1 Syntaxe des classes abstraites 12](#_Toc1863521246)

[3.2 Quand utiliser des classes abstraites? 13](#_Toc1476387139)

[A- Une classe abstraite ne peut être instanciée : on ne peut pas créer d’objets de cette classe. 14](#_Toc1083264519)

[4. Les interfaces 14](#_Toc802014309)

[4.1 Syntaxe des interfaces 16](#_Toc1963137519)

[L05E\_Interfaces 17](#_Toc1428133835)

# Sous-classes et super-classes

Une sous-classe est une classe qui fait partie d’une autre classe.

On appelle super-classe la classe parent et sous-classe la classe enfant.

Une sous-classe hérite des champs et méthodes de la super-classe en y ajoutant d’autres membres (champs et méthodes).

Exemple :

Super-classe **Animaux**

* Champ **dureeDeVieMoyenne**

Sous-classe **Mammifères** (sous-classe de Animaux)

* Champ **dureeAllaitement**

Sous-classe **Oiseaux** (sous-classe de Animaux)

* Champ **TailleOeufMoyenne**

Alors :

La classe Mammifères aura 2 champs : **dureeDeVieMoyenne** et **dureeAllaitement**

La classe Oiseaux aura 2 champs : **dureeDeVieMoyenne** et **TailleOeufMoyenne**

L’avantage principal est la réutilisation du code : dans ce cas-ci, **dureeDeVieMoyenne** est commun à toutes les classifications d’animaux et on a besoin de le déclarer une seule fois.

Le principe fondamental **d’héritage** est ce qui permet à la sous-classe de recevoir les champs et méthodes de sa super-classe.

L’exemple montre l’héritage pour des champs, les méthodes peuvent aussi être héritées.

## Syntaxe de sous-classes et super-classes

Une **superclasse** est déclarée comme une classe normale, c’est la présence de sous-classe qui en fait une super-classe.

On utilise le mot clé **extends** pour indiquer qu’une sous-classe hérite d’une super-classe, au moment de sa déclaration.

Exemple :

|  |
| --- |
| public class *nom de la sous-classe* **extends *nom de la classe*** |

### Utilisation des membres d’une classe parent

Les membres de la classe parent ne sont pas déclarés dans la sous-classe, mais on peut y accéder.

Exemple, accès d’une variable de super-classe dans une méthode de sous-classe :

|  |
| --- |
| **public class Animaux{  protected int dureeDeVieMoyenne; }  public class Mammiferes extends Animaux{  private int dureeAllaitement;    public void afficherInfo(){  System.*out*.println("La duree moyenne de ce Mammifères est" + dureeDeVieMoyenne);  System.*out*.println("La duree d'allaitement de ce Mammifère est" + dureeAllaitement);  } }** |

### Le mot clé super

Dans l’exemple précédent il n’y a pas de confusion parce qu’aucun membre de la sous-classe ne porte le même nom qu’un membre de la classe parente.

On peut identifier un membre de la classe parente avec la notation complète : NomDeLaClasse.NomDuMembre.

Dans le cas d’héritage on préfère remplacer le nom de la classe parente par le mot clé **super** (à l’opposé de *this*). Le mot clé **super** remplace la classe parente, avec l’avantage d’indiquer clairement que cet élément est hérité.

**Exemple 1** – mot super pour un champs:

|  |
| --- |
| **public class Animaux{  protected String description; }  public class Mammiferes extends Animaux{  private String description;   public void afficherInfo(){  System.*out*.println("Voici la description de cette classe d'animaux: " + super.description);  System.*out*.println("Voici la description de ce mammifère: " + this.description);  } }** |

Super peut être utilisé dans plusieurs contextes, pour utiliser des méthodes de la classe parent. C’est le cas d’un constructeur de la classe parent.

Note : lorsque possible, utilisez des noms distinctifs pour vos éléments de code. Ex : **descriptionAnimaux** et **descriptionMammifere**, cela évite des erreurs.

## La surcharge de méthodes en héritage

Les membres d’une instance d’une sous-classe incluront ceux de la superclasse ET de la sous-classe.

On peut faire de la surcharge de méthodes incluant des méthodes définies dans la superclasse ET dans la sous-classe.

Les règles habituelles s’appliquent : les méthodes doivent porter le même nom et avoir une signature différente.

**Exemple :**

|  |
| --- |
| **public class Animaux{  protected double vitesseAuSol; *//Ne pas oublier protected* public void setVitesse(double vitesse){  this.vitesseAuSol = vitesse;  } }  public class Mammiferes extends Animaux{  *//La méthode existe déjà dans la classe parent et même fonctionnement* }  public class Oiseaux extends Animaux{  private double vitesseEnVol; *//Peut être private parce que pas de sous-classe à Oiseaux* public void setVitesse(double auSol, double enVol){  vitesseEnVol = enVol; *//Pas hérité, n'existe pas dans Mammifères* vitesseAuSol = auSol; *//Hérité pourrait aussi: super.vitesseAuSol = auSol* } }** |

Notez ici qu’on a de **l’héritage** ET du **polymorphisme**.

## Les constructeurs en héritage

Les **constructeurs** ne sont pas hérités d’une superclasse. C’est une méthode qui porte le nom de la classe, il ne peut donc pas être le même dans la sous-classe.

Mais il peut être appelé à partir d’une sous-classe.

**Exemple 1** – utilisation de super pour un constructeur:

|  |
| --- |
| **public class Animaux{  public Animaux(){}; *//Constructeur dans la classe parent* }  public class Mammiferes extends Animaux{  public Mammiferes(){ *//Constructeur dans la sous-classe* super(); *//Appelle le constructeur de la super-classe, même nom que la super classe* } }** |

## La redéfinition de méthodes (Overriding)

Comme la surcharge (overloading), la **redéfinition** (overriding) permet de créer des méthodes qui portent le même nom, mais qui ont une fonctionnalité différente.

Contrairement à la surcharge, la redéfinition ne requiert pas que les signatures soient différentes. Cependant, on ne peut pas utiliser la redéfinition à l’intérieur d’une même classe, c’est un mécanisme réservé à l’héritage (sous-classe, classe abstraite, interface).

Une méthode redéfinie :

* Est une forme de polymorphisme
* Utilise l’annotation **@Override**
* Doit être redéfinie dans une sous-classe
* Doit avoir la même signature que la classe parente
* Doit avoir le même type de retours que la classe parente (ou un sous-type)
* Doit avoir un modificateur d’accès avec le même accès ou plus d’accès, mais pas moins
* Ne peut pas redéfinir une méthode parente **private**, **static** ou **final**
* Ne fonctionne pas avec les constructeurs
* Est obligatoire pour les méthodes abstraites et les interfaces (concept expliqué plus loin)

**Exemple :**

Dans un magasin à grande surface, tous les produits auront une méthode calculer prix. Certains produits électroniques doivent inclure un écofrais, pour les ordinateurs portables, c’est $0.20.

On définit une superclasse **Produit** avec une méthode **calculerPrix** et une sous-classe **Portable** qui redéfinit cette méthode pour tenir compte des écofrais.

|  |
| --- |
| **public class Produit {  private String description;  protected double prix;   public double calculerPrix(){  return prix \* 1.14975; } }** |

|  |
| --- |
| **public class Portable extends Produit {  public static final double *ECOFRAIS* = 0.20;   @Override  public double calculerPrix(){  return ( prix + *ECOFRAIS* ) \* 1.14975; } }** |

### L05A\_Sous-classes - Formes

Le laboratoire L05A permet de pratiquer les superclasses et sous-classes.

### Retours sur le laboratoire L05A\_Sous-classes - Formes

Retours sur la syntaxe et l’utilisation des sous-classes.

# La classe DecimalFormat

Les objets **DecimalFormat** permettent de modifier l'apparence (formater) des nombres décimaux. Un objet DecimalFormat comprend un modèle qui est appliqué aux différents nombres décimaux à l'aide de la méthode format.

Les **DecimalFormat** ne servent qu'à modifier l'affichage d'un nombre. La valeur intrinsèque du nombre demeure intacte.

**DecimalFormat** utilise des patterns (modèles) un peu comme les expressions régulières. Un objet de la classe DecimalFormat définit le pattern à utiliser et fournit les méthodes pour l’appliquer à une valeur.

**Exemple** : utilisation de **DecimalFormat** pour l’affichage.

|  |
| --- |
| **double statPourcent = 0.19; *//Valeur : 19 pourcent conservé en décimal  //Déclare un objet DecimalFormat avec son pattern* DecimalFormat df = new DecimalFormat("#%"); *//Le pattenr ici est "#%"  //Utilise l'objet pour formater l'affichage de la donnée* System.*out*.println(df.format(statPourcent));  *//Produira 19%*** |

La classe est sous **java.text.DecimalFormat** et doit être importée pour être utilisée.

### L05B\_DecimalFormat

Le laboratoire L05B vous invite à rechercher la syntaxe appropriée pour utiliser la classe **DecimalFormat** pour quelques formats différents.

### L05C\_Héritage - Maisons

Le laboratoire L05C vous permet de pratiquer les sous-classes avec un projet.

Retours en classe après la portion **Chalet**, puis après la portion **MaisonMobile**.

### L05D\_Héritage - Routes

Le laboratoire L05D vous permet de pratiquer les sous-classes avec un projet.

Retours en classe après le laboratoire.

# Les classes abstraites

Les super-classes telles que vues jusqu’à maintenant peuvent être utilisées par elle-même, sans faire appel aux sous-classes.

Dans nos exemples, on pourrait faire appel à un champ **description** de la classe **Animaux**, qui décrirait le règne animal, en opposition aux végétaux et minéraux par exemple.

Un objet (une instance) de la classe **Animaux** aurait une utilité même si on n’utilise pas ses sous-classes.

On peut aussi faire des classes dont la seule utilité est de fournir des membres communs aux sous-classes. On les appelle des **classes abstraites**, parce que leur seul rôle est de donner une couche d’abstraction à leurs sous-classes.

**Exemple :**

Pour un magasin de vente de produits en ligne, on pourrait avoir une classe **Produit**.

Les produits seraient divisés en catégories, puis en modèles etc.

Tous les produits auront un nom, une description et un prix.

Ils pourraient aussi avoir des méthodes, comme **getPrix**, **afficherDescription** etc.

Des sous-classes de Produit pourraient être **Ecran**, **TelephoneMobile**, **CableUSB** etc., qui auront leurs propres caractéristiques.

Dans ce contexte, on n’utilisera jamais un objet **Produit**, ce sera toujours des objets **Ecran**, **TelephoneMobile** etc.

L’objectif de la classe Produit est d’offrir une couche d’abstraction qui cache les détails des sous-classes et permet de définir des champs (ou des méthodes) communes à toutes les sous-classes, pour éviter de les définir dans chaque sous-classe.

Dans le cas des méthodes, elles seront alors redéfinies dans chaque sous-classe.

Une classe abstraite :

* Utilise le mot clé **abstract**
* Permet l’utilisation de champs et de méthodes communes à des sous-classes de type semblable
* Peut avoir un constructeur
* Ne peut pas être instanciée (pas d’objet créés à partir de cette classe)
* Peut avoir des méthodes régulières ou abstraites
* Permet les champs static ou non-static et final ou non-final

Une méthode abstraite :

* Requiert une classe abstraite
* Utilise le mot clé abstract
* N’a pas de bloc de code
* Renvoie à des méthodes des sous-classes

## Syntaxe des classes abstraites

**Exemple :**

|  |
| --- |
| ***//Un objet Produit ne peut pas être créé //La classe abstraite sert à fournir des membres communs aux sous-classes***  **abstract class Produit{**  ***//Une constante commune à toutes les sous-classes* static final double *taxeTPS* = 0.05;  *//Tous les produits ont un prix mais il n'est pas le même* protected double prix;  *//Tous les produits ont un nom mais il n'est pas le même* protected String nom;**  ***//Produit.getTauxTPS est accepté parce static => pas besoin d'instance* static double getTauxTPS(){  return *taxeTPS*;  }   *//Méthodes abstraite, redéfinie pour chaque produit  //Pas de bloc de code  //tous les produits ont des spécifications,  //mais elles ne sont pas les même pour chacun (écran a une résolution, disque dur a une taille)* abstract public void afficherSpecifications(); }**  **class Ecran extends Produit{  *//Pas besoin de redéclarer prix ou nom, déjà déclaré dans la classe Produit* private double resolution; *//Existe pour les ecrans mais pas pour d'autres produits   //Constructeur pour cette sous-classe, inclus la résolution* public Produit(double pPrix, String pNom, double pResolution){  prix = pPrix;  nom = pNom;  resolution = pResolution;  }   *//Implémentation de la méthode déclarée dans la classe abstraite* public void afficherSpecefications(){  System.*out*.println("Prix: " + prix);  System.*out*.println("Nom: " + nom);  System.*out*.println("Resolution: " + resolution);   }  }** |

## Quand utiliser des classes abstraites?

Pourquoi utiliser une classe abstraite plutôt qu’une super-classe avec héritage?

### Une classe abstraite ne peut être instanciée : on ne peut pas créer d’objets de cette classe.

**Exemple 1**

Soit une classe Employe dans un programme d’entreprise.

Tous les employés ont les même champs et méthodes à une exception : les représentant sur la route ont aussi un compte de dépense avec un montant et une méthode pour calculer le total.

On créé une classe Representant, sous-classe de la classe Employe.

Si le code traite un représentant, l’objet est de la classe Representant, mais pour tous les autres employés, on créé des objets de la super-classe Employe. Ceci n’est pas possible avec une classe abstraite.

Dans un tel cas, on n’emploierait pas une classe abstraite.

**Exemple 2**

Soit un programme qui permet de traiter des formes pour calculer l’aire et le périmètre. On a une classe Forme et des sous-classe Cercle, Triangle et Carre.

La classe Forme aura un champ Aire et un champ Périmètre, et des méthodes CalculerAire et CalculerPerimetre.

On veut que peu importe la forme, on appelle toujours les méthodes par : obj.CalculerAire() et obj.CalculerPerimetre.

On définit alors les méthodes abstraites dans la classe Forme, qui doit alors être elle-même abstraite et les méthodes sont redéfinies dans les sous-classes en fonction de leurs calculs respectifs.

On ne traitera jamais un objet Forme, ce sera toujours un Rond, Triangle ou Carre, donc une classe abstraite peut être utilisée ici.

Si un autre programmeur créé une classe Rectangle, sous-classe de Forme, il DOIT créer des méthodes CalculerAire et CalculerPerimetre, assurant ainsi une uniformité du code.

Aussi, on s’assure qu’on ne peut pas utiliser d’objets de la classe forme dans une autre partie du code.

# Les interfaces

Les interfaces sont apparentées aux classes abstraites dans le sens où elles ne peuvent pas être instanciées, on ne peut pas créer un objet à partir de ces classes, seulement de ses sous-classes.

On utilise souvent les interfaces pour exposer des champs et des méthodes à d’autres classes en gardant le contenu des sous-classes cachées.

C’est un mécanisme utilisé entre autres lorsque des équipes de développeurs différentes travaillent sur des entités communes.

**Exemple :**

Un manufacturier automobile créé le code qui fait fonctionner l’ordinateur de ses voitures. Ce code est confidentiel et secret. Mais l’ordinateur de la voiture expose une quantité importante d’informations qui peuvent être utilisées par d’autres produits.

Par exemple, un créateur de logiciels qui créé une application mobile qui récolte des informations sur la voiture et contrôle certains de ses paramètres, comme la température, le volume de la radio, l’éclairage etc.

Le manufacturier automobile ne veut pas exposer le code de classes définies dans l’ordinateur de la voiture, mais veut permettre à l’application mobile de faire appel à des méthodes comme **DemarrerChauffage**, **AllumerPlafonier** etc.

Une interface est une classe qui exposera des champs et des méthodes à d’autres classes sans ne rien révéler du code, qui est entièrement compris dans des sous-classes. Dans le cas ci-dessus, le détail des classes interfaces est fourni au développeur d’application mobile, mais pas le reste.

On utilise aussi les interfaces pour définir des champs ou des méthodes pour des classes qui n’ont pas de lien logique entre elles. Par exemple, une classe **Enregistrement** pourrait avoir une méthode **enregistrer()** utilisée par des classes **Musique**, **Vidéo**, **Conference** etc. Ces classes ne sont pas des sous-catégories d’un ensemble Enregistrement. Cette façon de faire permet de créer des méthodes avec une syntaxe universelle pour des classes différentes.

Aussi, une classe peut implémenter plusieurs interfaces ce qui n’est pas possible avec les classes normales ou abstraites.

Une interface :

* Utilise les mots clés **interface** et **implements**
* Ne peut pas être instanciée
* A tous ses membres **public** par défaut
* Peut hériter de plusieurs classes (ce n’est pas le cas des classes abstraites)

**Exemple :**

* À l’épicerie, les produits considérés essentiels sont non-taxables, alors que les produits considérés de luxe et ls produits non alimentaires sont taxables.
  + On veut que tous les produits taxables implémentent une méthode **calculerTaxe(prix)s**.
* Certains produits ont un prix fixe alors que d’autres sont vendus au poids.
  + On veut que les produits vendu au poids implémentent une méthode **calculerPrixGramme(prix, nbGramme).**

On a la structure hiérarchique de classe suivantes :

* Produits
  + Alimentaire
    - FruitsEtLegumes
      * Frais Pesable
      * Conserve
    - Breuvages
      * Laitier
      * BoissonGazeuse Taxable
    - NonPerissable
      * Farine Pesable
      * NoisettesAuChocolat Pesable, Taxable
  + NonAlimentaire Taxable
    - Vaisselle
    - Shampoing

La structure ne permet pas d’ajouter les méthodes requises dans une classe donnée et toutes ses sous-classes.

* Les classes Taxable et Pesable seront des interfaces.
* Les classes **Frais, Farine** et **NoisettesAuChocolat** implémenteront la classe Pesable.
* Les classes **BoissonGazeuse**, **NonAlimentaire** et **NoisetteChocolat** implémenteront la classe Taxable.
* **NoisettesAuChocolat** est Pesable **ET** taxable, ce qui n’est pas possible avec les interfaces, ni les super-classes héritées.

Lorsqu’un comportement doit être implémenté par des classes qui n’ont pas de lien entre elles, on fait généralement appel à des interfaces.

## Syntaxe des interfaces

**Exemple :**

|  |
| --- |
| ***// Une interface est une classe nécessairement public***  **public interface Taxable{**  ***// Les méthodes d’une interface n’ont pas de code   // Notez l’absence de { } qui sont remplacées par un ;* public float *calculerTaxe*(float prix);  }**  **class NoisettesAuChocolat extends NonPerissable implements Taxable,Pesable{  *//Mot clé implements pour les interfaces***  ***//Plusieurs interfaces peuvent être implémentées par une classe***  ***//Une classe peut hériter (extends) d’une classe ET implémenter une/des interface(s)***  ***//Implémentation de la méthode de l’interface Taxable* public float calculerTaxe(float prix){  return prix \* 0.15975;  }**  **...  }** |

### L05E\_Interfaces

Le laboratoire L04E vous permet de pratiquer les interfaces.

NOTE : une classe peut hériter (extends) d’une classe ET mettre en place une interface (implements)

Retours en classe après la portion automobile, puis après la portion motoneige.