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

Table des matières

[1. Sous-classes et super-classes 3](#_Toc150761883)

[1.1 Syntaxe de sous-classes et super-classes 4](#_Toc150761884)

[Utilisation des membres d’une classe parent 4](#_Toc150761885)

[Le mot clé super 5](#_Toc150761886)

[1.2 La surcharge de méthodes en héritage 6](#_Toc150761887)

[1.3 Les modificateurs en héritage 7](#_Toc150761888)

[1.4 Les constructeurs en héritage 8](#_Toc150761889)

[1.5 La redéfinition de méthodes (Overriding) 9](#_Toc150761890)

[L05A\_Sous-classes - Formes 10](#_Toc150761891)

[Retours sur le laboratoire L05A\_Sous-classes - Formes 10](#_Toc150761892)

[2. La classe DecimalFormat 11](#_Toc150761893)

[L05B\_DecimalFormat 11](#_Toc150761894)

[L05C\_Héritage - Maisons 11](#_Toc150761895)

[L05D\_Héritage - Routes 11](#_Toc150761896)

[3. Les classes abstraites 12](#_Toc150761897)

[3.1 Classe et méthode abstraite : concept 12](#_Toc150761898)

[3.2 Classe abstraite ou Super-classe (concrète)? 12](#_Toc150761899)

[Exemple : classes d’animaux invertébrés - Mammifères, Oiseaux, Poissons etc. 12](#_Toc150761900)

[3.3 Caractéristiques de classe abstraite 13](#_Toc150761901)

[3.4 Syntaxe des classes abstraites 14](#_Toc150761902)

[3.5 Méthode abstraite 15](#_Toc150761903)

[3.6 Caractéristique de méthode abstraite : 15](#_Toc150761904)

[3.7 Syntaxe de méthode abstraite 15](#_Toc150761905)

[4. Les interfaces 16](#_Toc150761906)

[4.1 Caractéristiques des interfaces 16](#_Toc150761907)

[4.2 Exemple : taxable et pesable 17](#_Toc150761908)

[4.3 Syntaxe des interfaces 18](#_Toc150761909)

[5. Tour de magie 19](#_Toc150761910)

[L05E\_Interfaces 21](#_Toc150761911)

# 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 modificateurs en héritage

Les modificateurs d’accès en héritage ont un comportement particulier. Le comportement de base des modificateurs d’accès traite les sous-classes comme des classes différentes.

**public**

Un membre public d’une super classe est disponible partout et sera aussi public dans la sous-classe.

**private**

Un membre private d’une superclasse ne sera PAS accessible dans ses sous-classes. Un tel membre existe mais n’est pas visible. Cette configuration force l’accès par des méthodes non private, même pour les sous-classes.

Si on veut que la sous-classe « voit » le membre, il faut utiliser un autre modificateur d’accès.

**protected**

On revisite protected ici, avec une nouvelle définition :

* Un membre protected d’une super-classe est disponible dans toutes les sous-classes, même package et autres packages, ET dans les autres classes du même package.

Un membre protected d’une classe sera disponible dans une classe du même package (pas d’héritage).

Un membre protected d’une superclasse sera disponible dans une sous-classe de tous les package et y sera protected.

**default**

Un membre default d’une super-classe sera disponible dans une sous-classe du même package et sera default.

Un membre default d’une super classe ne sera pas accessible dans une sous-classe d’un autre package.

L’utilisation de default n’est pas recommandée, spécialement en héritage.

**Note** : il n’y a pas de modificateur d’accès « seulement les sous-classes ». On peut forcer ce comportement en conservant dans un seul package une superclasse et ses sous-classes et aucune autre classe.

OPTIONNEL : démo MA\_Heritage.

## 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

## Classe et méthode abstraite : concept

**Classe abstraite**

Classe parent qui ne peut pas être instanciée : on ne peut pas créer d’objets de cette classe.

Tous ses membres sont destinés à l’héritage.

Une classe abstraite peut avoir des méthodes abstraites et non abstraites (concrètes)

**Méthode abstraite**

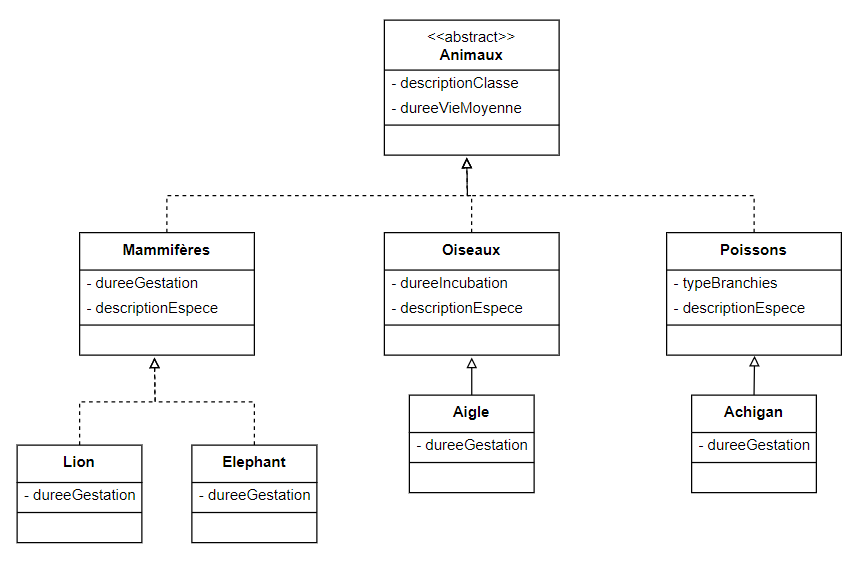
Méthode qui n’a pas de bloc de code, elle est obligatoirement définie dans une sous-classe.

## Classe abstraite ou Super-classe (concrète)?

Ça dépend du contexte, du besoin.

Pour décider si une classe parente devrait être abstraite ou non, on se pose la question : est-ce qu’on aura besoin de créer des objets de cette classe?

### Exemple : classes d’animaux invertébrés - Mammifères, Oiseaux, Poissons etc.



**Contexte A**

L’application n’utilisera jamais d’objet **Mammifere** ou **Oiseaux** ou **Poisson**. Le seul objectif de ces classes est de transmettre des champs et méthodes aux espèces d’animaux. Alors ces classes pourraient être des classes **abstraites**.

**Contexte B**

Dans l’application, on produit une page d’information qui explique les différences entre les classes **Mammiferes**, **Oiseaux** etc et qui permet d’interagir avec leurs informations. Alors des **objets** de ces classes seront requis et elles devraient être des **super-classes (concrètes).**

## Caractéristiques de classe abstraite

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

## 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***  **public 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(); }**  **public 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 afficherSpecifications(){  System.*out*.println("Prix: " + prix);  System.*out*.println("Nom: " + nom);  System.*out*.println("Resolution: " + resolution);   }  }** |

## Méthode abstraite

Méthode qui n’a pas de bloc de code, elle est obligatoirement définie dans une sous-classe.

A comme objectif de s’assurer que les sous-classes définissent cette méthode avec une signature uniforme.

## Caractéristique de 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 de méthode abstraite

On déclare une méthode abstraite dans la classe parent sans bloc de code – utilise ; et pas {}

On définit la méthode dans la classe enfant, avec son bloc de code.

**Exemple** : voir méthode **afficherSpecifications()** de l’exemple précédent.

# Les interfaces

Interface : classe qui ne peut être instanciée, on ne peut créer d’objets.

On utilise des interfaces pour garantir qu’une méthode sera mise en place par toutes les sous-classes, avec un nom uniforme.

On utilise aussi 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.

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.

## Caractéristiques des interfaces

* Utilise les mots clés **interface** et **implements**
* Ne peut pas être instanciée
* Ne peut avoir de champs sauf des champs **static** et **final** (des constantes)
* A tous ses membres **public** par défaut
* Une sous-classe peut hériter de plusieurs interfaces (pas le cas des super-classes et classes abstraites)
* Une sous-classe peut hériter s’une super-classe/classe abstraite ET implémenter des interfaces

## Exemple : taxable et pesable

* À l’épicerie, les produits considérés essentiels sont non-taxables, alors que les produits considérés de luxe et les 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 hiérarchique 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;  }**  **...  }** |

# Tour de magie

On peut déclarer un paramètre de méthode comme étant un objet de la classe Parent, mais à l’appel de la méthode passer un objet de la classe Enfant.

**Exemple 1 – super-classe simple**

|  |  |
| --- | --- |
| Classe **Parent** | Classe **Utilitaire** (utilise les autres classes)   * Méthode accepte un paramètre de la classe Parent. * Méthode ne peut PAS utiliser methodeEnfant |
| public class ClasseParent {    public void methodeParent(){  System.out.println("Execute méthode parent.");  }  } | public class ClasseUtil {  public void methodeUtil(ClasseParent cp){  cp.methodeParent();  }  } |
| Classe **Enfant**   * Méthode redéfinie du parent * Plus une méthode spécifique | **Main**   * Peut passer un objet Parent en à methodeUtil * Peut aussi passer un objet Enfant à methodeUtil * La méthode utilisée correspond à l’objet reçu |
| public class ClasseEnfant extends ClasseParent{  @Override  public void methodeParent(){  System.out.println("Execute override de méthode parent dans enfant.");  }  public void methodeEnfant(){  System.out.println("Execute methode enfant");  }  } | public class Main {  public static void main(String[] args) {  ClasseParent objCP = new ClasseParent();  ClasseEnfant objCE = new ClasseEnfant();  ClasseUtil objCU = new ClasseUtil();  objCU.methodeUtil(objCP);  objCU.methodeUtil(objCE);  }  } |

Dans methodeUtil, on reçoit on déclare un paramètre de la classeParent.

Lorsqu’on appelle methodeUtil dans le main, on peut passer un **objet Parent ou Enfant**.

La methodeUtil reconnait automatiquement le type d’objet passé et utilise la méthode qui correspond.

**Exemple 2 – classe abstraite**

|  |  |
| --- | --- |
| Classe **Parent**   * Classe abstraite | Classe **Utilitaire** (utilise les autres classes)   * Méthode accepte un paramètre de la classe Parent. * Méthode ne peut PAS utiliser methodeEnfant |
| public abstract class ClasseParent {    public void methodeParent(){  System.out.println("Execute méthode parent.");  }  } | public class ClasseUtil {  public void methodeUtil(ClasseParent cp){  cp.methodeParent();  }  } |
| Classe **Enfant**   * Méthode redéfinie du parent * Plus une méthode spécifique | **Main**   * On ne peut PAS créer un objet Parent * On ne peut PAS passer un objet Parent à methodeUtil * La méthode utilisée est nécessairement celle de Enfant |
| public class ClasseEnfant extends ClasseParent{  @Override  public void methodeParent(){  System.out.println("Execute override de méthode parent dans enfant.");  }  public void methodeEnfant(){  System.out.println("Execute methode enfant");  }  } | public class Main {  public static void main(String[] args) {  ~~// ClasseParent objCP = new ClasseParent();~~  ClasseEnfant objCE = new ClasseEnfant();  ClasseUtil objCU = new ClasseUtil();  ~~// objCU.methodeUtil(objCP);~~  objCU.methodeUtil(objCE);  }  } |

Fonctionnement semblable à l’exemple 1 MAIS, la classe Parent est abstraite.

On ne peut PAS créer d’objet de la classe Parent, à l’appel, on passera obligatoirement un objet enfant.

**La magie**

Dans methodeUtil(), on manipule un objet reçu en paramètre de la classe Parent (cp). **Mais la création d’un tel objet est impossible**. En réalité, l’objet n’est pas créé dans cette classe, il sera toujours traité comme un objet d’une classe enfant. En fait de n’importe quelle classe enfant.

La methodeUtil() peut traiter des objets de classes différentes mais seulement si elles sont enfant de classe Parent. Pour ces raisons, seules les méthodes qui sont héritées peuvent être utilisées.

**Exemple 3 – interface**

|  |  |
| --- | --- |
| Classe **Parent**   * Interface * Pas de code dans la méthode | Classe **Utilitaire** (utilise les autres classes)   * Méthode accepte un paramètre de la classe Parent. * Méthode ne peut PAS utiliser methodeEnfant |
| public interface ClasseParent {    public void methodeParent();  } | public class ClasseUtil {  public void methodeUtil(ClasseParent cp){  cp.methodeParent();  }  } |
| Classe **Enfant**   * Méthode redéfinie du parent * Utilise implements * Plus une méthode spécifique | **Main**   * On ne peut PAS créer un objet Parent * On ne peut PAS passer un objet Parent à methodeUtil * La méthode utilisée est nécessairement celle de Enfant |
| public class ClasseEnfant implements ClasseParent{  @Override  public void methodeParent(){  System.out.println("Execute override de méthode parent dans enfant.");  }  public void methodeEnfant(){  System.out.println("Execute methode enfant");  }  } | public class Main {  public static void main(String[] args) {  ~~// ClasseParent objCP = new ClasseParent();~~  ClasseEnfant objCE = new ClasseEnfant();  ClasseUtil objCU = new ClasseUtil();  ~~// objCU.methodeUtil(objCP);~~  objCU.methodeUtil(objCE);  }  } |

Même principe que pour une classe abstraite, sauf que le code des méthodes n’est présent que dans les classes enfant.

### 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.