Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Dr. Tahar Moulay de Saida
Faculté de Technologie

Département d'Informatique

Programmation Orientée Objet

Présenté par :

Dr. MEKOUR Mansour

Maître de conférences « B » en Informatique

Octobre 2018

Programmation Orientée Objet

Notes de cours

Année universitaire 2018 - 2019

* * * * *

Université de Saïda

(Dr Tahar Moulay)

MEKOUR Mansour

Faculté de Technologie
Département Informatique

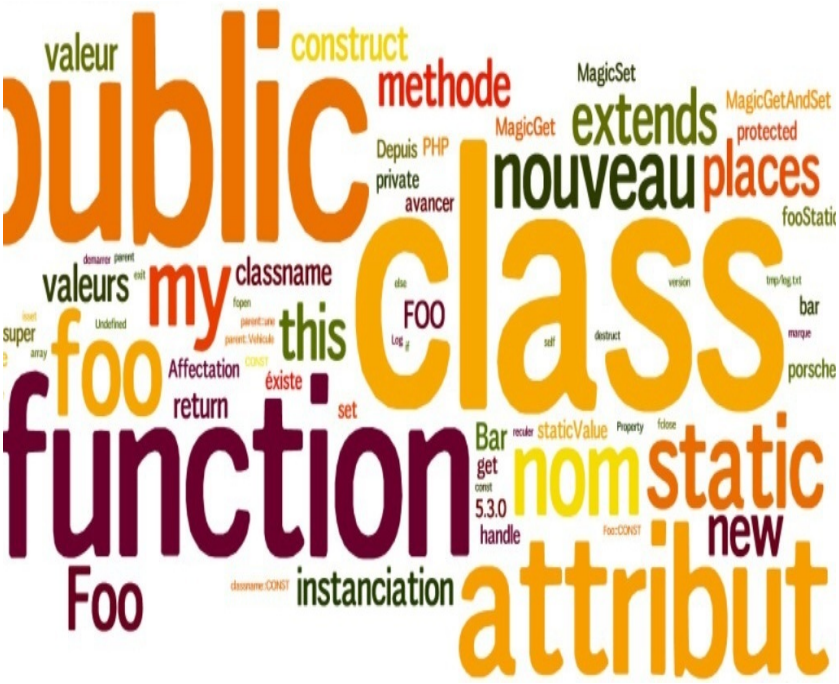
public
valeur
construct
methode
MagicSet
MagicGetAndSet
protected
extends
MagicGet
Depuis PHP
private
avancer
nouveau
places
fooStatic
class
valeurs
my
classname
FOO
bar
porsche
super
this
foo
Affectation
existe
return
set
function
Bar
staticValue
nom
static
get
5.3.0
handle
new
Foo
instanciation
attribut

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

L'orienté objet est un paradigme parmi les paradigmes de programmation et de développement des applications informatiques. Ce paradigme confère aux programmes de bonnes propriétés, et il permet une plus grande clarté conceptuelle, et il offre principalement une grande robustesse face aux modifications futures avec une maintenance logicielle très efficace. Le but de ce cours est de présenter principalement les concepts essentiels de ce paradigme en ignorant les aspects techniques proprement liés aux langages de programmations. On va traiter les concepts de bases, les méthodes de développement, ainsi que les conseils pertinents qui aboutissent aux développements de bons programmes. Il s'agit d'une une initiation à la programmation OO (orienté objet) en se focalisant principalement sur la présentation des concepts fondamentaux communs à la plupart des langages de programmation OO généraliste. Pour se faire, on suppose que les notions de bases de la programmation dite procédurale (les types, les variables, les boucles, les fonctions, etc) sont plus au moins acquises. Le cours est déroulé en adoptant le langage de programmation *Java* comme un moyen d'illustration des concepts fondamentaux de la programmation OO. Ainsi, pour aborder ces concepts fondamentaux, on suppose une familiarité avec les bases de la syntaxe du langage *Java* afin de les mettre en pratique. Alors, on se focalise sur les concepts purement OO (orienté objet) en ignorant des aspects proprement techniques liés au langage de programmation qui sont proprement dite API (Application Programming Interface). Le cours permettre au fur et à mesure de

se contrôler et de s'orienter par soi-même. Alors, d'une manière progressive, qu'on ait bien compris le contenu du cours, il se faudra pratiquer. Pas de pratique, pas de bonne de programmation, voire même pas de programmation du tout. J'insiste sur le fait de la pratique individuelle, puisque l'apprentissage de la langue d'une manière générale, et de la programmation en particulier nécessite un travail assidu et rigoureux.

Chapitre 1

Abstraction et Encapsulation

Sommaire

La programmation OO est une manière spécifique et très efficaces de programmation, et elle n'est pas relative à un langage de programmation particulier. La programmation OO confère à fournir aux programmes des propriétés très intéressantes et très riches en termes de modularité et de maintenabilité. Pour l'apprentissage de telle programmation, il est censé connaître certains concepts de bases à-propos de la programmation tout court : (i) le codage des traitements en utilisant des structures de contrôle. (ii) la structuration de données à travers des tableaux. (iii) et la modularité des programmes par le biais de procédures et de fonctions. Ce genre de

concepts permet d'avoir un style de programmation de base dite programmation procédurale, dont les données (les variables) et les traitements sont séparées et indépendantes dans un programme. Malgré qu'elles soient vue séparées, les traitements peuvent s'exprimer par le biais de modules (procédures et fonctions), et le lien entre les données et les traitements est réalisé au travers de passage de paramètres. Alors, les traitements manipulent les données, dont lesquelles influencent certainement les traitements (voir figure 1.1). Une des caractéristiques essentielles de la programma-

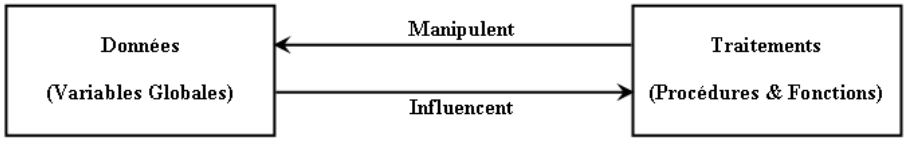

FIGURE 1.1 – Programmation procédurale

tion OO est le regroupement des traitements et des données en une seule et même entité. Partons d'un exemple où en considérant la notion d'étudiant. Si on souhaite par exemple calculer les moyennes des étudiants pour le module de programmation OO. Alors, un étudiant on peut le caractériser par : un nom, un prénom, une note de travaux dirigés, et une note d'examen. En programmation dite procédurale, celle qu'on savait déjà, on peut utiliser des variables pour le nom, le prénom, la note de TD, et la note d'examen. Ensuite, on peut y attribuer des valeurs précises (voir figure 1.2). Par la suite, on peut calculer la moyenne de l'étudiant pour le module programmation OO en passant la note de TD et la note d'examen à une fonction (dite *moyenne* par exemple) pour faire le traitement souhaité. Évidemment, les entités (les données et les traitements) selon ce style de programmation s'expriment de façon séparée. Les variables $Note\_TD$ et $Note\_Examen$ sert à stocker les données, et la fonction *moyenne* permet de réaliser les traitements de calcule de la moyenne pour le module POO. Alors, un lien existe entre les deux entités par le biais du passage des paramètres. Il s'agit de la principale critique à saisir sur ce programme, est l'absence de tel lien sémantique entre les différentes entités. Le lien sémantique

```
GestionArtificielle {

 String Nom="Amine";
 String Prénom="Lamouri";
 double Note_Examen=17;
 double Note_TD=13;

 double moyenne(double note_Examen,  double note_TD) {
 return ((note_Examen*2 + note_TD)/3);
 }

 System.out.println("La moyenne de l'étudiant " + Nom +" " +
   Prénom + " est : " + moyenne(Note_Examen, Note_TD));
}
```

FIGURE 1.2 – Gestion artificielle : Programmation procédurale

qui unit le *Note_TD* et le *Note_Examen* est invisible dans le programme, alors qu'il s'agit de *Note_TD* et de *Note_Examen* pour un même étudiant. Pareillement pour le lien entre le *Nom* et le *Prénom*, il n'est pas immédiatement visible. En outre, les données le *Nom*, le *Prénom*, la *Note_TD* et la *Note_Examen* sont tousse liées à un même étudiant, alors qu'il est totalement invisible dans le programme. Conceptuellement, c'est bien la notion d'étudiant qu'on veut manipuler au travers les données *Nom*, *Prénom*, *Note_TD* et *Note_Examen*. C'est bien la moyenne d'un étudiant pour le module POO qu'on souhaite calculer. Le fait de pouvoir regrouper en une seule et même entité les données (le nom et le prénom, la notes de TD et la notes d'examen), ainsi que les traitements (la fonction *moyenne*) qui s'y unissent spécifiquement permettent d'établir d'une manière explicite le lien entre ces différentes entités. Effectivement, la programmation OO fournit un certain nombre d'outillages en renforçant la notion de robustesse, de modularité, et de lisibilité aux programmes afin d'avoir une meilleure maintenabilité. La robustesse par rapport au changement face aux améliorations futures en franchissant l'obligation

de réécriture totale des programmes, et également la robustesse vis-à-vis des erreurs de manipulation des données. En effet, ces dernières années, la majorité des applications développées consistent à améliorer et étendre des programmes préexistant pour pouvoir minimiser le coût de développement. Afin de bien favoriser ce genre de développement par un mécanisme dé réutilisation de codes, et de ne pas bâties de zéro, des applications très modernes et très riches, il devient très intéressant et très recommander de mieux organiser les programmes en tenant en compte la robustesse, lisibilité, modularité et maintenabilité. Il s'agit alors de l'un des fondamentaux de l'approche OO, par le biais de quatre concepts centraux qui sont l'encapsulation, l'abstraction, l'héritage et le polymorphisme. Ces concepts ne sont pas spécifiques à un langage de programmation bien déterminé, mais plutôt à l'approche OO elle même. On va définir dans ce chapitre la notion d'encapsulation et d'abstraction. L'héritage et le polymorphisme seront présentés dans les chapitres 3, et 4.

1.1 Principe d'Abstraction et d'Encapsulation

L'idée consiste à regrouper dans une même entité des données, et des traitements agissants sur ces données. Par exemple, on peut regrouper dans une seule et même entité (voir figure 1.3) le nom, le prénom, la note de TD et la note d'examen caractérisant un étudiant, ainsi que la fonctionnalité de calcule de la moyenne de l'étudiant. En termes de jargon, on parle de la notion d'attributs pour désigner les données,

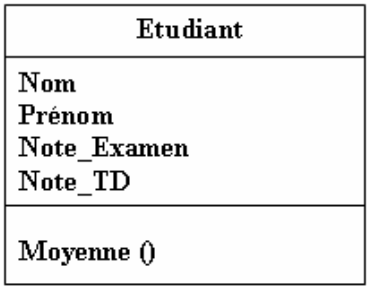

FIGURE 1.3 – Regroupement des données et des traitements

et de la notion de méthodes pour les fonctionnalités. En programmation OO, on a la possibilité de définir des nouveaux types par le biais de la notion de classe. Ces nouveaux types peuvent être adoptés pour travailler effectivement avec des éléments de type plus abstrait. En termes de jargon, on parle de la notion d'objets pour désigner ces éléments. Ces objets sont caractérisés par leurs attributs et leurs méthodes, et ils peuvent être cohabités et interagis dans le programme. Alors, un programmeur en OO va travailler avec la notion d'objet. Partons d'un exemple où on souhaite manipuler un certain nombre d'étudiants, et non seulement un seul. Selon l'approche procédurale, on doit adopter des variables *Nom*, *Prénom*, $Note\_TD$, et $Note\_Examen$ pour chaque étudiant. Alors, s'il s'agit d'un nombre important d'étudiants, il devient beaucoup plus fatigant et pénible. Par exemple, pour calculer la moyenne de chaque étudiant pour le module POO, on a besoin d'invoquer la méthode *moyenne* plusieurs fois, selon le nombre des étudiants, en passant à chaque fois les bons paramètres. Ainsi, supposé qu'on se trompe, et on peut faire un calcul de la moyenne en impliquant la note de TD d'un étudiant (ex $Note\_TD_1$) avec la note de d'examens d'un autre étudiant(ex $Note\_Examen_2$).Au travers d'une représentation générique et commune pour toute les étudiants (un *Nom*, un *Prénom*, une $Note\_TD$ et une $Note\_Examen$), on peut effectuer le même calcul de la moyenne pour toutes les étudiants. Alors, le mécanisme de l'encapsulation va permettre de regrouper par le biais de la notion *Etudiant* tout ce qui est nécessaire à le modéliser. Il va permettre de pouvoir travailler avec des entités plus abstraites, et en se focalisant sur l'essentiel. C'est-à-dire, pour tous les étudiants (un premier étudiant, un deuxième étudiant, etc.) qui sont de type *Etudiant*, on va invoquer la fonctionnalité de calculs de la moyenne sur ces étudiants. En d'autres termes, on n'est plus de se préoccuper du fait qu'un étudiant est décrit de façon plus abstraite par un nom, un prénom, et des notes de TD et examen, tandis qu'on se focalise alors sur les aspects essentiels en travaillant avec la notion *Etudiant*, et en calculant la moyenne

sur un étudiant. En tant que programmeur utilisateur de la notion *Etudiant*, On ne aperçois de l'étudiant que ce que l'on appelle en jargon orienté objet, interface d'utilisation, c'est-à-dire les fonctionnalités qui sont offertes pour la manipulation de l'étudiant, comme le calcul de la moyenne (voir figure 1.4). Faisons une analo-

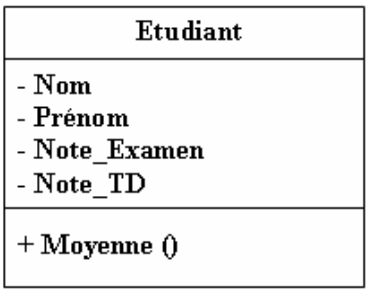

FIGURE 1.4 – Encapsulation en programmation OO

gie avec un objet de la vie courante. Pour conduire une voiture, on n'a besoin de savoir que l'interface d'utilisation. Plus précisément, on a besoin d'avoir le volant, les trois pédales (une pédale de l'accélérateur, une pédale du frein et une pédale de l'embrayage), et le Levet de vitesse. Il n'est pas utile de connaître la manière de constitution du moteur, et ni la façon de son fonctionnement. C'est absolument le même point de vue en programmation OO. Pour utiliser un nouveau type (une nouvelle classe en programmation OO), il est nécessaire de savoir que l'interface d'utilisation envisagée par son programmeur concepteur, plutôt de connaitre les détails internes d'implémentation. On peut distinguer alors deux niveaux de perception d'un objet. Un niveau interne conçu par un programmeur concepteur, et un niveau externe utile au programmeur utilisateur. Pour notre exemple, le niveau interne qui représente le nouveau type *Etudiant* (la classe *Etudiant*), et qui n'est pas forcément utile à celui qui utilise ce nouveau type. C'est lui qui se préoccuper de tous les détails d'implémentation, à savoir qu'un étudiant est décrit par un nom, un prénom et une note de TD et une note d'examen. Ainsi, c'est lui qui définie concrètement la manière de calcul de la moyenne. Le niveau externe, l'utilisateur de ce type *Etudiant* est le niveau qui intéresse le programmeur utilisateur. C'est lui qui peut déclarer

des variables de type *Etudiant*, et les initialiser avec des valeurs adéquates. Ainsi, c'est lui qui s'intéresse aux fonctionnalités utiles pour le calcule de la moyenne. En programmation OO, on n'a pas uniquement la notion de regroupement en une entité unique des données et des traitements, mais également la possibilité de définir des niveaux de visibilité. Le programmeur concepteur de nouveau type va pouvoir d'indiquer les détails d'implémentation qui seront invisible au programmeur utilisateur, et les fonctionnalités que l'on souhaite offrir à l'utilisateur externe qui seront visible au programmeur utilisateur (on parle alors de l'interface d'utilisation). Pour le nouveau type *Etudiant* (la classe *Etudiant*), on a comme interface d'utilisation la fonctionnalité de calcul de la moyenne (*moyenne*). Le reste, les attributs *Nom*, *Prénom*, $Note\_TD$ et $Note\_Examen$ représente les détails interne d'implémentation qui sont inaccessibles au programmeur utilisateur de la classe *Etudiant*. Grace a ce genre de fondamentaux de la programmation OO, on peut aboutir a une meilleure robustesse des programmes vis-à-vis aux changements future. Typiquement, lorsqu'on change de voiture, même si la technologie du moteur est changée, l'interface de conduite de la voiture reste globalement la même. On sait conduire toujours une voiture, même si les détails internes de fonctionnement de la voiture sont changés. Puisque, on ne perçoit de la voiture que quelque chose d'abstrait, On ne considère concrètement que ce qui permet de la conduite (le volant, les trois pédales, et le levet de vitesse). On peut dire alors, l'encapsulation consiste à regrouper en une entité unique les données et les traitements qui la caractérisent en empêchant l'accès aux ses détails d'implémentation et en offrant son interface d'utilisation. Une interface d'utilisation d'une entité, classe d'objets, permet d'aboutir à une abstraction, une vision abstraite de l'objet, par le biais de l'outillage d'encapsulation. L'abstraction ne permet de perçoir de l'objet que ce que fournit son interface d'utilisation en termes de manipulations possibles. Concrètement, une interface d'utilisation doit être décrite par le bais de méthodes offertes aux programmeurs utilisateurs. L'un

des objets fondamentaux d'abstraction et d'encapsulation est d'assurer une bonne visibilité, et une bonne cohérence au programme. Par exemple, pour une approche OO où je manipule explicitement la notion d'étudiant. On établit le lien sémantique entre l'étudiant et sa moyenne, alors que pour une approche procédurale où je manipule des données de bas niveau, le lien est établi de façon très implicite. En outre, le programmeur utilisateur de la classe *Etudiant* ne peut l'utiliser qu'au travers de l'interface d'utilisation prévue par le programmeur concepteur de cette classe, ce qui autorise au programmeur concepteur de modifier les détails internes d'implémentation sans influencer l'utilisation concrète. Concrètement, si le programmeur concepteur de la classe *Etudiant* décide de revoir son implémentation initiale en utilisant un tableau de doubles pour représenter les attributs $Note\_TD$ et $Note\_Examen$, au lieu de les adopter au moyen de deux doubles.Alors, s'il adapte conformément le calcul de la fonction moyenne, alors le programmeur utilisateur de calcul de la fonction moyenne n'est nullement impacté. La séparation entre les niveaux interne (détails d'implémentation), et externe (interface d'utilisation) assure une utilisation plus rigoureuse. Les modifications à l'intérieur de la structure interne restent invisibles à l'extérieur. En programmation OO, la modélisation des attributs et des méthodes nécessitent des choix techniques d'implémentation. Habituellement, on considérer les attributs comme des détails d'implémentation inaccessibles depuis l'extérieur. Tandis que, certain nombre de méthodes bien choisies sont considérées comme une interface d'utilisation. Après avoir arrivé au terme de la présentation des concepts fondamentaux de l'OO qui sont l'encapsulation et l'abstraction. On va, essayer à les pratiquer concrètement en *Java* dans ce qui suit.

1.2 Classes et Objets

Après avoir vu les concepts généraux de la programmation OO, Cette section présente ces concepts concrètement en *Java*. Comme il est déjà introduit, selon

l'approche OO, le résultat de processus d'abstraction et d'encapsulation permet de définir la notion de classe dont les différents membres sont des attributs et des méthodes. Par exemple, la définition de la classe *Etudiant* avec les attributs *Nom*, *Prénom*, *Note_TD* et *Note_Examen*, et la méthode *moyenne* (voir figure 1.5). En

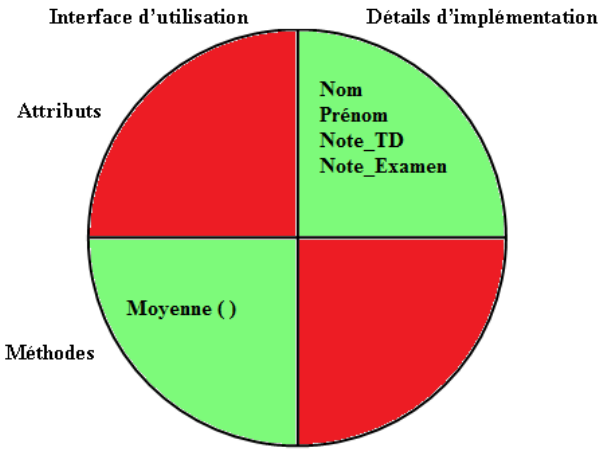

FIGURE 1.5 – Classe (Type) *Etudiant*

termes de langage de programmation, une classe permet de créer un type de définir des variables en regroupant à la fois des données et des traitements. Par exemple les trois variables $E1$, $E2$, $E3$ sont de type *Etudiant*. Au sens de programmation OO, ces variables ont comme valeurs des instances (des objets), et on parle dans ce cas aux variables d'instances. Concrètement en java, pour déclarer une classe, on va utiliser le mot réservé *class* suivie d'un identificateur de la classe et d'une définition de cette classe délimité par les deux accolades (voir figure 1.6). La définition d'une classe c'est

```
class NomClasse {
// Déclaration des  attributs de la classe
//----------------------------------
//Déclaration des  méthodes de la classe
}
```

FIGURE 1.6 – Syntaxe de déclaration d'une classe

une définition d'un nouveau type qui peut être utilisé comme n'importe quel autre type pour déclarer des variables dont les valeurs sont des instances. Pour ce faire, on utilisera l'identificateur de la classe (l'identificateur du type) suivi par l'identificateur de variable. Par exemple, pour déclarer une variable nommé $E1$ de type *Etudiant*, on écrira carrément *Etudiant* $E1$;. En terminologie orientée objet, une valeur d'une variable correspond à une instance, malheureusement en *Java*, elle est en effet une référence sur un objet concret en mémoire. Alors, on doit faire attention pour manipuler (par exemple pour l'affectation, la comparaison, et l'affichage) ces objets. Toutes ce genres de manipulation sera traité dans le chapitre 2. Pour définir des classes, on peut les mettre dans un seul fichier, ou les mettre par fichier (chaque classe dans un fichier). Par exemple, supposant qu'on a la classe *GestionArtificielle*, qui utilise la classe *Etudiant* qui sont toutes les deux déclarées dans un seul et même fichier nommé *CodeGestionArtificielle.java* (voir figure 1.7). Pour le compiler, on

```
class GestionArtificielle {
 public static void main(String[] args) {
    Etudiant E;
    //........
 }
}//_____________________________________________________________
class Etudiant {
 String Nom;
 String Prénom;
 double Note_Examen;
 double Note_TD;
  //........
}
```

FIGURE 1.7 – Déclaration des classes dans un même fichier

utilisera la commande *javac CodeGestionArtificielle.java*. Cette compilation va génère les deux fichiers *.class* : *GestionArtificielle.class* et *Etudiant.class*. Ensuite, pour exécuter le programme, on utilisera la commande *java GestionArtificielle.class*

(Puisque la méthode *main* est définit dans la classe *GestionArtificielle*). Pour un IDE (environnement de développement intégré) comme *Eclipse*, on peut utiliser la commande *Build*, même on peut lancer directement l'exécution a partir de la classe *GestionArtificielle*. Si on suppose maintenant qu'on a pour chaque classe un fichier à part. On déclare alors la classe *Etudiant* dans un fichier nommé *Etudiant.java*, et la classe *GestionArtificielle* dans le fichier *GestionArtificielle.java*. Dans ce cas, pour les compiler depuis le terminal, il faudrait exécuter séparément la commande pour chacun des fichiers. Pour un IDE, en appelant une seule fois la commande *Build* pour les compiler séparément. L'IDE se chargerait de les compiler, et il génère comme précédemment les deux fichiers *.class*. Pour les exécuter depuis le terminal on utilise la commande *Java GestionArtificielle* (puisque c'est dans la classe *GestionArtificielle* qu'on a définit la fonction *main*. C'est exactement ce que fait l'IDE pour le lancement de l'exécution. Pour la déclaration des attributs, en mettant (comme d'habitude) pour chacun des attributs le type suivi par l'identificateur de l'attribut, suivi par le point virgule. Pour notre exemple (voir figure 1.8), les attributs de la classe *Etudiant* sont *Nom*, *Prénom*, *Note_TD* et *Note_Examen*. Pour accéder aux attributs d'un objet, on adopte la

```
class Etudiant {

  String Nom;
  String Prénom;

  double Note_Examen=17;
  double Note_TD=13;

}
```

FIGURE 1.8 – Déclaration des attributs

syntaxe $IdentificateurInstance.IdentificateurAttribut$ (c'est- à-dire l'identificateur de l'instance suivi par un point et l'identificateur de l'attribut). Par exemple, pour utiliser l'attribut Nom de l'objet étudiant $E1$, on écrira $E1.Nom$. Partons d'un exemple concret. Dans un seul et même fichier, on définie une classe $GestionArtificielle$ qui contient la méthode $main$, et qui utilise la classe $Etudiant$. Dans cette dernière, on déclare les attributs Nom, $Prénom$, $Note\_TD$ et $Note\_Examen$ (voir figure 1.9). Dans la classe GestionArtificielle, on a créé une instance de la classe $Etudiant$ (au ni-

```
class GestionArtificielle {
 public static void main(String[] args) {
  Etudiant E = new Etudiant ();
  Etudiant E = new Etudiant();
  E.Note_Examen = 13;
  E.Note_TD = 15.0;
  System.out.println("Note de TD : " + E.Note_TD);
 }
}//-------------------------------------------------------------
class Etudiant {
 String Nom;
 String Prénom;
 double Note_Examen;
 double Note_TD;
}
```

FIGURE 1.9 – Accès aux attributs

veau de la ligne). On a utiliser la syntaxe ($Etudiant\ E1 = new\ Etudiant()$), pour la déclaration et l'initialisation d'une variable d'instances $E1$ de la classe $Etudiant$. La syntaxe $new\ Etudiant()$ permet de créer et initialiser tous les attributs de la classe $Etudiant$ avec des valeurs par défaut. En programmation en java, on a les valeurs par défauts *false, 0, 0.0*, et la valeur spéciale *null* pour les type *boolean, int, double,* et *les objets de type classe*. Ainsi, les attributs de type *String* ont aussi la valeur spéciale *null* comme les objets, par ce que le type *String* est une classe prédéfinit par le langage *Java* (voir la section 2). Pour notre exemple, les attributs *Nom*, *Prénom*,

Note_TD et *Note_Examen* ont des valeurs par défauts avant qu'on les modifie. En programmation OO. Les méthodes sont carrément des fonctions qui doivent être toujours déclarées au sein de la classe. Alors, la déclaration d'une méthode en OO est exactement la même que d'habitude, où on doit spécifier un type de retour, un identificateur de la méthode, une liste des arguments entre les parenthèses, et ensuite la définition de corps de la méthode entre deux accolades) (voir figure 1.10). Alors,

```
typeRetour IdentificateurMethode (typeParam1 nomParam2, ...) {
 // corps de la méthode
 //...
}
```

FIGURE 1.10 – Syntaxe de déclaration d'une méthode

on peut constater que, la différence unique par rapport à la programmation procédurale, est qu'une méthode en programmation OO doit être existée dans la classe dont laquelle elle est membre. Pour notre exemple, la méthode *moyenne* doit être déclarée dans une classe, plus précisément dans la classe *Etudiant* dont laquelle elle est membre (voir figure 1.11). Comme il est indiqué par cette figure, contrairement

```
class Etudiant {

  //.........

  double moyenne() {
  return ((Note_Examen*2 +Note_TD)/3);
  }
}
```

FIGURE 1.11 – La méthode moyenne selon l'approche OO

comme on le fait généralement selon l'approche procédurale, la méthode *moyenne*

de la classe *Etudiant* est déclarée cette fois-ci sans aucun arguments. En effet, si on définit une méthode en dehors de toute classe, alors on a y besoin de passer comme paramètres les $Note\_TD$, et $Note\_Examen$ pour effectuer le calcul souhaité (voir figure 1.12). Selon l'approche OO, les données $Note\_TD$ et $Note\_Examen$ sont

```
double moyenne(double note_Examen, double note_TD) {
      return ((note_Examen*2 +note_TD)/3);
}
```

FIGURE 1.12 – La méthode moyenne selon l'approche procédurale

des attributs membres de la classe *Etudiant*, et la méthode *moyenne* fait partie des membres de cette classe *Etudiant*. C'est pour cela qu'elle accès à toutes les membres de la classe *Etudiant*, et en particulier aux attributs $Note\_TD$ et $Note\_Examen$ de la classe. C'est ce qu'on appelle concrètement en OO une portée de classe, c'est-à-dire que tous les membres (attributs et méthodes) d'une classe sont connus de l'entièreté de cette classe, et particulièrement de chacune de ses méthodes. Alors, toutes les méthodes de la classe *Etudiant* ont la capacité d'accès aux attributs $Note\_TD$ et $Note\_Examen$, il devint intitule de les passer comme arguments aux méthodes de la classe *Etudiant*, et c'est une erreur de débutant en OO, de passer les attributs d'une classe comme arguments pour une méthode de même classe. En revanche, si une méthode nécessite des données de l'extérieur de la classe pour pouvoir fonctionner, alors il devient fortement nécessaire dans ce cas de les définir comme paramètres de la méthode. Par exemple, supposons qu'on a dans la classe *Etudiant* les différents membres, et en particulier une méthode qui permet de vérifier cette fois-ci l'activité de l'étudiant dans son groupe. La méthode en question doit recevoir par exemple la note moyenne de son groupe (voir figure 1.13). Alors, elle modifie les instances de la classe, pour accomplir le traitement souhaité en impliquant cette fois-ci la moyenne

```
class Etudiant {
 //.........
 boolean estActif(double moyenneGroupe) {
  boolean actif=false;
  if (((Note_Examen*2 +Note_TD)/3)> moyenneGroupe) actif=true;
  return actif;
  }
}//-------------------------------------------------------------

class GestionArtificielle {

 public static void main(String[] args) {
  double moyenneGroupe = 15;
  Etudiant E ;
  E.estActif(moyenneGroupe);
 }
}
```

FIGURE 1.13 – Des données externes pour la méthode moyenne

de son groupe comme donnée externe. Alors *moyenneGroupe* c'est une donnée externe pour la classe *Etudiant*, et elle ne fait pas partie des membres de cette classe, elle n'est pas un attribut de la classe *Etudiant*. Alors, en programmation OO c'est pour ce genre de situations uniquement qu'on doit passer les données comme paramètres aux méthodes de classes. Après avoir vue la manière de déclaration des méthodes de classes, on va voir maintenant comment les accéder pour pouvoir les utiliser. Similairement à la manière d'accès aux attributs, la syntaxe de base d'accès aux méthodes de la classe est $nomVariableInstances.nomMethode$, puis ensuite entre les parenthèses on fournit la liste des paramètres requise au fonctionnement de la méthode (voir figure 1.14). Par exemple, si on souhaite appeler la méthode *moyenne* sur une variable d'instances $E1$ de type *Etudiant*, alors on écrit simplement $E1.moyenne()$ sans mettre rien entre les parenthèses, puisque la méthode *moyenne* ne nécessite pas de recevoir des données de l'extérieur pour pouvoir fonc-

```
IdentificateurVariableInstances.IdentificateurMethode (
   ValeurParam1, ValeurParam2, ... )
```

FIGURE 1.14 – Accès aux méthodes

tionner. Partons d'un exemple complet, où on a la classe *GestionArtificielle*, qui utiliser dans une méthode *main* une variable d'instances $E1$ de la classe *Etudiant* (voir figure 1.15). Dans la classe *Etudiant*, on a la méthode *moyenne*. Pour appeler

```
class GestionArtificielle {
 public static void main(String[] args) {
 Etudiant E1 = new Etudiant();
 E1.Nom="Amine";
 E1.Prénom="Lamouri";
 E1.Note_Examen = 17.0;
 E1.Note_TD = 13.0;

 System.out.println("La moyenne de l'étudiant "+  E1.Nom +"  "
                   +  E1.Prénom+" est : " +  E1.moyenne());
  }
}//_____________________________________________________________
class Etudiant {
 String Nom;
 String Prénom;
 double Note_Examen;
 double Note_TD;

 double moyenne() {
 return ((Note_Examen*2 +Note_TD)/3);
 }

}
```

FIGURE 1.15 – *GestionArtificielle* : instances de la classe *Etudiant*

cette méthode *moyenne* sur l'instance de $E1$ de la classe *Etudiant*, alors on écrira carrément $E1.moyenne()$. $E1.moyenne()$ retourne un double que l'on peut afficher

avec le message *moyenne* : . Par exemple, il affiche *moyenne* : 16, qui est le résultat de (17*2 +13)/3. Il faut bien comprendre que tout objet a ses propres attributs. Si par exemple, on déclare trois variables d'instances $E1$, $E2$, $E3$ de la classe *Etudiant*, et si on va attribuer par exemple la valeur 10 pour l'attribut $Note\_TD$ de $E1$, et la valeur 13 pour l'attribut $Note\_Examen$ pour E2, on aura les valeurs 11.5 pour $E1$, et 3.8 pour $E2$. Il s'agit alors de valeurs différentes. On aura ainsi dans une autre zone mémoire, les valeurs par exemple 34.3 pour l'attribut $Note\_Examen$ de $E3$, et puis une valeur pour $Note\_TD$. C'est bien qu'on a trois variables différentes qui existent en mémoire, et quand on appelle $E1.moyenne()$, c'est la méthode *moyenne* de la classe *Etudiant*, qui va être appliquer à l'instance de $E1$, ce qui fait pour cet appel de la méthode *moyenne* que $Note\_Examen$ désignera E1.$Note\_Examen$, et $Note\_TD$ désignera E1.$Note\_TD$. Si on fait appelle la méthode moyenne a l'aide de l'instruction $E2.moyenne()$, alors cette fois-ci $Note\_TD$ désignera $E2.Note\_TD$, et $Note\_Examen$ désignera $E2.Note\_Examen$. Noter qu'en toute rigueur que les variables $E1$, $E2$, $E3$ sont des références vers des objets en mémoire, et elles ne contiennent pas des instances effectives (voir la section 2.2 du chapitre 2), mais l'objectif d'avoir insisté sur le fait que toute appelle d'une méthode d'un objet a accès aux différents attributs de ce même objet. Dans cette section, nous avons introduit concrètement la syntaxe de déclaration des membres, les attributs et les méthodes, d'objets (des membres d'instances) en *Java*. Dans la section suivante, on va étudier comment concrètement différencier entre les interfaces d'utilisation et les détails d'implémentation pour une classe en OO.

1.3 Droits d'Accès

Dans la section précédente on a présenté les concepts de base de l'approche OO. On a introduit la manière de définition des classes, des objets. Ainsi que la manière déclaration des membres (des attributs et des méthodes) de ces classes

et ces objets. Dans cette présente section, on va adresser concrètement la manière d'accorder des détailles d'implémentation et des interfaces d'utilisation. En d'autres termes, la façon définir la partie cachée (privé) et la partie visible (publique) pour une classe, et exactement pour les objets.

1.3.1 Détailles d'implémentation

Dans la section 1.1, on a vue que la notion d'encapsulation vise à regrouper en une entité unique les données et les traitements en empêchant l'accès aux détails d'implémentation, et en définissant des interfaces d'utilisation. Plus concrètement, si on souhaite par exemple de reprendre la classe *Etudiant*, où on décide d'avoir utilisé les attributs $Note\_TD$ et $Note\_Examen$ uniquement dans la classe *Etudiant* (c'est-à-dire comme détaille d'implémentation, et on parle en termes de jargon à un accès privé). En java, pour définir la partie privée on va adopter le modificateur *private* pour chacun des membres (attributs et des méthodes) que l'on ne souhaite pas dans l'interface. Donc ils sont inaccessibles depuis l'extérieur de la classe. Concrètement, si par exemple on veut déclare une instance de la classe *Etudiant* dans la méthode *main* (voir figure 1.16). Alors, en d'hors de la classe *Etudiant*, si on souhaite accéder directement à l'attribut $Note\_TD$ par exemple, le compilateur donnera une erreur qui annonce que le champ $Note\_TD$ est en accès privé dans la classe *Etudiant*.

```
class Etudiant {
 private String Nom;
 private String Prénom;
 private double Note_Examen;
 private double Note_TD;
 //.........
}
```

FIGURE 1.16 – Détailles d'implémentation : Accès privé

1.3.2 Interfaces d'utilisation

Contrairement à la partie privée qui limite l'accès uniquement à l'intérieur de la classe, la partie publique permet d'étendre l'accès aux différents membres (attributs et des méthodes) pour toutes les autres classes. Pour ce faire, le langage *Java* offre le modificateur *public*. Par exemple pour permettre l'accès aux autres classe d'utiliser la méthode *moyenne* de la classe *Etudiant*, on ajoute carrément le modificateur *public* devant son en tête et sa définition (voir figure 1.17). Dans ce cas, si on déclare

```
class Etudiant {
 //.........
  public double moyenne() {
  return ((Note_Examen*2 +Note_TD)/3);
  }
}
```

FIGURE 1.17 – Interfaces d'utilisation : Accès publique

une variable d'instances E de la classe *Etudiant* dans la méthode *main* par exemple, alors on peut cette fois-ci appeler la méthode *moyenne* de la classe *Etudiant* sur cette variable d'instance E, puisque la méthode en question est déclarée dans l'interface d'utilisation de la classe *Etudiant* par le biais de modificateur *public*. Si jamais on altère la déclaration comme *private*, alors l'accès à la méthode *moyenne* depuis la méthode *main* est devient impossible.

Cependant, si aucun droit d'accès n'est spécifié pour un membre donné, la méthode *moyenne* par exemple, alors on aura une autorisation de visibilité implicite dite friendly, ce qu'on appelle en OO le droit d'accès par défaut (voir figure 1.18). Il s'agit d'une restriction d'accès, où toutes les classes doivent nécessairement être de même paquetage (répertoire).

```
class Etudiant {
 //.........
 double moyenne() {
 return ((Note_Examen*2 +Note_TD)/3);
 }
}
```

FIGURE 1.18 – Interfaces d'utilisation : Accès par défaut

1.4 Accesseurs et Manipulateurs

Supposons que l'on souhaite accéder aux attributs privés pour les utiliser en d'hors de la classe. Évidemment, on ne peut pas les accéder directement depuis l'extérieur de la classe. Par exemple, si on a une variable d'objets de la classe *Etudiant* dans la méthode *main*, alors on ne peut pas accéder directement a l'attribut privée $Note\_TD$ si on souhaite afficher la note de TD d'un étudiant par exemple. Pour ce genre de situations, on doit mettre dans l'interface d'utilisation de la classe *Etudiant* des méthodes biens choisies. Des méthodes qui vont permettre de fournir au monde extérieur les valeurs des attributs privés. Pour notre exemple on peut définir une méthode ($getNote\_TD$ par exemple) dans la classe *Etudiant* en accédant à l'attribut $Note\_TD$, et en fournissant également sa valeur comme une valeur de retour pour le méthode en question (voir figure 1.19). De telles méthodes sont ce qu'on appelle

```
class Etudiant {
 private double Note_TD;
 //.....
 public double getNote_TD() { return Note_TD; }
}
```

FIGURE 1.19 – Accesseur $getNote\_TD$

en termes de jargon des accesseurs (des méthodes get ou des getters). Il s'agit des méthodes qui font partie de l'interface d'utilisation pour lire les valeurs des attributs privés. Puisque toutes les méthodes d'un objet ont l'autorisation d'accès à tous ses attributs, alors on peut définir de la même façon précédente d'autres accesseurs pour les attributs privés(voir figure 1.20). Si on suppose maintenant que l'on souhaite mo-

```
class Etudiant {
 //.....

 String getNom() { return Nom; }
 String getPrénom() { return Prénom; }
 public double getNote_Examen() { return Note_Examen; }
 public double getNote_TD() { return Note_TD; }
}
```

FIGURE 1.20 – Accesseurs

difier les valeurs des attributs privés en reçoivent des valeurs depuis d'autres classes. C'est bien qu'on ne peut pas les accéder de l'extérieur de la classe. Par exemple, si on a déclaré dans la méthode *main* une variable d'objets de la classe *Etudiant*, alors on ne peut pas accéder directement a l'attribut privé $Note\_TD$, pour y affecter une note de TD pour un étudiant comme une nouvelle valeur. Pour ce faire, on doit mettre dans l'interface d'utilisation de la classe *Etudiant* des méthodes de modification des valeurs des attributs privés. Des méthodes qui permettent la modification à partir du monde extérieur les valeurs des attributs privés. Par exemple, on peut déclarer une méthode ($setNote\_TD$) dans la classe *Etudiant* pour accèder à l'attribut $Note\_TD$, et également lui affecter une nouvelle valeur (voir figure 1.21). Ce genre de méthodes sont ce qu'on appelle des manipulateurs (des méthodes set ou des setters). Il s'agit des méthodes qui font partie de l'interface d'utilisation dédiées pour la modification des valeurs des attributs privés. Puisque toutes les méthodes d'un objet ont l'autorisation d'accès à tous ses attributs, alors on peut définir de

```
class Etudiant {
 //.....
 public void setNote_TD(double note_TD) { Note_TD = note_TD; }
}
```

FIGURE 1.21 – Manipulateur *setNote_TD*

la même façon précédente d'autres manipulateurs pour les autres attribust privés (voir figure 1.22). Partons d'un exemple où on a une variable d'instances $E1$ de la

```
class Etudiant {
 //.....

 void setNom(String nom) { Nom = nom; }
 void setPréom(String prénom) { Prénom = prénom; }
 public void setNote_Examen(double note_Examen) {

          Note_Examen = note_Examen;
     }
 public void setNote_TD(double note_TD) { Note_TD = note_TD; }
}
```

FIGURE 1.22 – Manipulateurs

classe *Etudiant* dans la méthode *main* de la classe *GestionArtificielle* (voir figure 1.23) pour Illustrer le propos. Comme il est montré par la figure, la classe *Etudiant* est conçue de deux parties. La partie détails d'implémentation spécifié par les attributs *Nom*, *Prénom* *Note_TD*, et *Note_Examen*. La partie interface d'utilisation défini par la méthode usuelle *moyenne*, les quatre accesseurs *getNom*, *getPrénom*, *getNote_TD* et *getNote_Examen*, et les deux manipulateurs *setNote_TD* et *setNote_Examen*. Les accesseurs sont définis avec des listes de paramètres vides, parce qu'il n'est pas nécessaire de passer de valeurs comme paramètres, tandis qu'ils ont des valeurs de retour de même type que l'attribut privé pour renvoyer sa va-

```
class GestionArtificielle {
 public static void main (String[] args) {
 Etudiant E1 = new Etudiant();
 E1.setNote_Examen(17.0);
 E1.setNote_TD(13.0);
 System.out.println(" La note d'examen est: "
                 + E1.getNote_Examen());
 }
}
```

```
class Etudiant {
 private String Nom;
 private String Prénom;
 private double Note_Examen;
 private double Note_TD;

 String getNom() { return Nom; }
 String getPrénom() { return Prénom; }
 double getNote_Examen() { return Note_Examen; }
 double getNote_TD() { return Note_TD; }


 double setNote_Examen(double   note_Examen) { Note_Examen =
   note_Examen; }
 double setNote_TD(double   note_TD) { Note_TD = note_TD; }

  double moyenne() {
  return ((Note_Examen*2 +Note_TD)/3);
  }

}
```

FIGURE 1.23 – *GestionArtificielle* : Accesseurs et Manipulateurs

leur. En outre, dans la méthode *main* de la classe *GestionArtificielle*, on a utilisé l'accesseur *getNote_TD* sur l'instance de $E1$ pour lire la valeur de *Note_TD*. On aussi affecter la valeur 13 à l'attribut *Note_TD* de l'instance de $E1$ au travers du manipulateur *setNote_TD*. On aussi user la valeur 17 comme paramètre pour le

manipulateur *setNote_Examen* sur $E1$ pour l'affecter à l'attribut *Note_Examen*. l'une des remarques les plus courantes pour les débutants en programation OO : *il est très compliqué et fastidieux de déclarer les attributs en privés, et ensuite d'adopter des getters et des setters pour les accéder et les modifier*. Il peut être compliqué fastidieux, mais il est très efficace pour la gestion des anomalies. Par exemple, si on a une variable d'instances $E1$ de la classe *Etudiant*, et si on suppose que l'attribut $Note\_TD$ est fait partie de l'interface d'utilisation, alors on peut le manipuler comme on veut. Alors, c'est ça vraiment le problème. Il est tout à fait possible d'attribuer la valeur 50 par exemple à l'attribut publique $Note\_TD$. Il s'agit alors d'une erreur fatale, puisque une note doit être comprise uniquement entre 0 et 20. Pour entamer ce genre d'affectations inattendus, on doit adopter un manipulateur, comme $setNote\_TD$, en empêchant les accès directes aux attributs, et en garantissant des affectations valides et cohérentes au moyen de bon géstion de telles erreurs (voir figure 1.24). De l'autre part, supposons qu'un attribut est cette fois-ci un objet plus

```
class Etudiant {

 private double Note_Examen;
 //...
 public void setNote_Examen(double poo) {
  if((poo>=0)&&(poo<=20)) Note_Examen = poo;
  else {/* Gestion des erreurs */ }
 }
}
```

FIGURE 1.24 – Manipulateur *setNote_Examen*

complexe, un objet de type classe, où il offre une interface d'utilisation. Alors, on peut utiliser par exemple, pour des centaines de fois la méthode *length* de la classe prédéfinie *String* au travers de l'instruction *Etudiant.Nom.length*. Supposant que le concepteur de la classe *Etudiant* décide de revoir la déclaration des attributs

en adoptant une *ArrayList* au lieu de *String*. Il s'agit alors d'un travaille d'adaptation obligatoire et fastidieux. Malheureusement, ce travaille d'adaptation est de la part du programmeur utilisateur, malgré qu'il s'agit d'une maintenance effectuer par le programmeur concepteur de la classe. Dans ce cas, le programmeur utilisateur doit adapter toutes les centaines d'instruction *Etudiant.Nom.length*, malgré qu'il utilise que l'interface d'utilisation de la classe *Etudiant*, et elle n'est pas changée (l'attribut *Nom* est toujours membre dans l'interface). Tandis que, si on passe au travers de l'interface plutôt d'accéder directement à des attributs, la responsabilité de toute modification (par exemple *String* en *ArrayList*) incombera au développeur de la classe uniquement, et les centaines de lignes resteront par contre inchangées puisqu'elles respectent toujours l'interface d'utilisation de la classe. C'est pour cette raison fondamentale, qu' on n'autorise jamais des accès directes à tous les attributs de la classe, mais on doit fournir des mécanismes de manipulation de détailles internes dans l'interface d'utilisation afin de garantir des modifications de maintenances très utiles et indépendantes, ce qu'on appelle comminent la modularisation. On outre, on pousse de passer par le biais de méthodes, et en mettant aucun attribut dans la l'interface d'utilisation. Évidemment, pour des petits programmes, il peut être n'a pas beaucoup de sens, mais il prend tout son sens pour des grands projets logiciel. C'est exactement pour ce genres de projets l'approche OO est envisagée. C'est pour ce genres d'esprit, qu'on essaye de séparer au maximum tous ce qui est interface d'utilisation de ce qui est ne l'est pas. En conséquence, il faut noter en toute rigueur que la phase de conception est très importante. On ne mettre systématiquement jamais des méthodes dans l'interface d'utilisation pour lire ou modifier tous les attributs d'une classe. Tandis qu'on doit décider dès le moment de la conception l'ensemble des attributs que l'on souhaite lire ou modifier depuis l'extérieur au travers de l'interface d'utilisation.

1.5 Masquage et démasquage d'attributs

Le masquage aura lieu lorsqu'un identificateur est utilisé par un certain objet cache un identificateur qui désigner un autre objet. Plus concrètement, en OO lorsqu'un identificateur d'un argument d'une méthode cache un identificateur d'un attribut de la classe. Par exemple le paramètre $Note\_TD$ de la méthode (le manipulateur) $setNote\_TD$ cache l'attribut $Note\_TD$ de la classe $Etudiant$ (voir figure 1.25). Dans ce cas, la variable $Note\_TD$ de la partie gauche de l'affectation désigne

```
class Etudiant {
 String  Note_TD;
 public void setNote_TD(double  Note_TD) {
   Note_TD =  Note_TD;   // l'identificateur Note_TD c'est le
   paramétre de la méthode ou bien l'attribut de la classe.
 }
}
```

FIGURE 1.25 – Masquage des attributs

l'attribut de la classe $Etudiant$, alors que $Note\_TD$ de la partie adroite de l'affectation indique le paramètre du manipulateur $setNote\_TD$. Malheureusement, pour ce genre de situations le compilateur est incapable de faire la différence, margé qu'il n'est pas effectivement terrible. Pour remédier à ce genre de problèmes, on peut utiliser des identificateurs différents (en prenant en compte ainsi, la sensibilité a la case des langages de programmation) comme on a fait avec tous les exemples de notre cours. Tandis qu'on peut adopter Aussi les techniques de démasquage fournit par les langages de programation (la référence *this*, qui désigne l'instance courante, pour le langage *Java*). Pour l'exemple précédent, et afin de lever l'ambiguïté entre la variable $Note\_TD$ entant que paramètre de paramètre, et celle entant qu'attribut de la classe $Etudiant$ on ajoute la référence *this* pour l'attribut de la classe (voir figure

1.26). Cependant, il est déconseillé d'utliser les techniques de démasquage offertes par les langages de programmation (la référence *this* pour le langage *Java*), mais plutôt de choisir des identificateurs très clairs et significatifs pour tout le monde en évitent telles d'ambiguïtés.

```
class Etudiant {
 String  Note_TD;
 public void set Note_TD(double  Note_TD) {
  this. Note_TD =  Note_TD;  // $this$ pour désigner l'
   attribut de la classe.
 }
}
```

FIGURE 1.26 – Démasquage des attributs

Chapitre 2

Création et Manipulation des Objets

Sommaire

Après avoir présenter les concepts fondamentaux de l'approche OO dans le chapitre 1. Plus précisément, après avoir introduit la notion d'abstraction et d'encapsulation, et la manière de définition concrète des classes en *Java*, et également les autorisation d'accès aux différentes membres d'objets de ces classes. Ce chapitre a pour objectif de mettre le point essentiellement sur un aspect très intéressant

qui est l'instanciation (la création) et la manipulation des objets. En outre, on va adressé l'influence de la manipulation des objets au travers des références sur les opérations d'affectations, de comparaisons, et d'affichages des objets, et également la destruction des objets en leur fin de vie.

2.1 Création des Objets

Dans la section 1.2 du chapitre 1, on a adressé concrètement en *Java* la définition des classes, et on a vu l'utilisation des ces classes comme de nouveaux type pour la déclaration des variables afin de permettre la manipulation des données beaucoup plus complexe (des objets de classes). Dans cette section, on va présenter concrètement la création des objets en initialisant leurs attributs avec des valeurs bien précises, des valeurs qui ne sont pas forcément par défaut (c'est-à-dire la valeur $false$ pour un attribut de type *boolean*, 0 pour un attribut de typeint, 0.0 pour un attribut de type *double*, et *null* pour un attribut de type *classe*), en d'autres termes,la manière d'attribution des bonnes valeurs au départ dès la création des instances de classes. Plus concrètement, pour déclarer une variable d'instances E de la classe $Etudiant$ par exemple, on adopte l'instruction $Etudiant\ E;$. Ainsi, pour faire affecter des valeurs aux attributs $Note\_TD$ et $Note\_Examen$, on peut utiliser par exemple des manipulateurs $setNote\_TD$, et $setNote\_Examen$ comme on a vu dans le chapitre précédent. Alors, on affecte tour à tour des valeurs à chacun des attributs (voir figure 2.1). On a déclaré dans la méthode $main$ de la classe $GestionArtificielle$ une variable d'instances E de type $Etudiant$. En outre une autre variable lu de type *double* qui permettre à l'utilisateur, via le clavier, d'affecter une valeur pour l'attribut $Note\_TD$ de E au travers du manipulateur $setNote\_TD$. Similairement, on peut affecter une valeur pour l'attribut $Note\_Examen$ de l'instance de E. Évidemment, il est certainement une très mauvaise façon d'initialisation d'attributs, parce qu'elle exige des modifications depuis l'extérieur, par le biais de fonctionnalité

```
class GestionArtificielle {
 public static void main(String[] args) {
 Etudiant E1;
 System.out.println("Quelle est la note d'examen ? ");
 lu = CLAVIER.nextDouble();
 E.set Note_Examen(lu);
 System.out.println("Quelle est la note d ? ");
 lu = CLAVIER.nextDouble();
 E.setNote_TD(lu);
 }
}
```

FIGURE 2.1 – Affecter individuellement une valeur

offertes dans l'interface publique, pour tous les attributs de la classe *Etudiant*. On peut envisager ainsi, l'implication de tous les attributs de la classe dans la l'interface visible. Ce genre d'autorisation de la modification des valeurs d'attributs depuis l'extérieur de classes est absolument imbuvable, puisqu'il casse certainement l'un des fondamentaux de l'OO qui est l'encapsulation (voir le chapitre 1). Puisque, le but de l'encapsulation étant parfaitement séparer l'interface d'utilisation et l'implémentation interne. L'accès publique direct ou indirect a tous les attributs au travers des manipulateurs, nécessite une traduction énorme des détailles d'implémentation dans l'interface d'utilisation. En revanche, le programmeur concepteur de la classe, c'est lui qui est le responsable de toute initialisation d'attribut, et n'est le programmeur utilisateur de la classe, puisque ce dernier ne doit pas nécessairement connaitre les détailles internes d'implémention d'une classe, et ainsi il risque d'avoir male initialisé les différents attributs. Le programmeur concepteur doit fournir pour le programmeur utilisateur une interface dédié pour certainement initialiser quelques attributs bien choisis. Pour ce faire, les concepteurs du langage *Java* offrent une méthode particulière consacrée à cette fin ce qu'on appelle en termes de jargon un *Constructeur*. Dans ce qui suit section, on va présenter les différentes variantes de cette méthode

particulière. On va introduit les constructeurs explicites (les constructeurs utilisateurs), le constructeurs par défaut (les constructeur uniques), le constructeur par défaut par défaut, et le constructeur de copie. Dans les sections 2.2.2, 2.2.1 et 2.2.3 on va aborder la manière d'affectation, de comparaison et d'affichage des objets, ainsi qu'on va adresser la fin de vie d'un objet dans la section 2.2.4.

2.1.1 Constructeurs

Comme on a déjà présenté, au lieu d'd'initialiser des attributs avec des valeurs par défaut, la bonne solution est la création d'une méthode particulière dédiée à l'affectation de valeurs bien précises. Il s'agit d'une méthode d'initialisation nécessairement fournit aux programmeurs utilisateurs d'une classe comme fonctionnalité de manipulation publique. Par exemple, la méthode *initialiser* dont les paramètres sont *nom* et *prénom* (voir figure 2.2) est pour objectif d'initialiser les attributs *Nom* et *Prénom* de la classe *Etudiant*. Pour utiliser alors la méthode *initialiser*, on doit

```
class Etudiant {
 private String Nom;
 private String Prénom;

 public void initialiser (String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
  }
}
```

FIGURE 2.2 – Définition d'une méthode d'initialisation

déclarer une variable d'instances par exemple E de type *Etudiant*. Ensuite, pour initialiser les attributs *Nom* et *Prénom* on invoque alors la méthode *initialiser* en fournissant les valeurs appropriés. Il s'agit alors, la très bonne solution, et c'est pour cette raison que les concepteurs de langage de programmation *Java* l'adopte comme

un moyen d'initialisations d'attributs, c'est ce que l'on appelle en programation OO un constructeur. En fait, un constructeur permet d'initialiser convenablement et systématiquement tous les attributs des instances d'une classe dés le début leur vie. Ils effectuer tout ce qui est nécessaire à accomplir pour la création des objets, et en particulier d'initialiser leurs attributs. Concrètement, pour définir un constructeur, on adopte naturellement la même syntaxe de déclaration d'une méthode en spécifiant convenablement une liste de paramètres nécessaires à l'initialisation des attributs. Contrairement à une méthode, le choix de l'identificateur d'un constructeur n'est pas libre, mais il doit être le même que l'identificateur de la classe dans laquelle il est défini, ainsi un constructeur ne possède pas un type de retour. (voir figure 2.3). Par exemple, pour la classe *Etudiant*, la méthode particulière *Etudiant*

```
class Classe {
  //.........
 IdentificateurClasse(paramètre1, paramètre2, ...) {
  Attribut1=paramètre1;
  Attribut2=paramètre2;
   //.........
  /* initialisation des attributs en utilisant
   liste_paramètres */
 }
}
```

FIGURE 2.3 – Syntaxe de base de la déclarration de constructeur

adopte deux paramètres *Nom* et *Prénom*, et elle dispose un même identificateur que la classe, il s'agit alors d'un constructeur dédier pour initialiser conformément les attributs *Nom* et *Prénom* (voir figure 2.4). Alors un constructeur est une méthode bien particulière dont les caractéristiques sont les suivantes :

1. elle n'a pas un type de retour, et également ni le type vide (*void*),
2. elle a exactement le même identificateur de sa classe.

```
class Etudiant {
 private String Nom;
 private String Prénom;
 private double Note_Examen;
 private double Note_TD;
 Etudiant (String nom, String prénom, double note_Examen,
   double note_TD){
  Nom = nom;
  Prénom=prénom;
  Note_Examen = note_Examen;
  Note_TD=note_TD;
 }
}
```

FIGURE 2.4 – Le constructeur de la classe "Etudiant"

3. elle est systématiquement invocable pour chaque création d'un nouveau objet.

Comme toutes méthodes, les constructeurs peuvent être surchargés. Alors, on peut avoir dans une même classe, non seulement un constructeur unique mais plutôt plusieurs constructeurs dont les signatures sont différentes, c'est-à-dire des constructeurs qui ne sont pas de même nombre de paramètres, de même types de paramètres, et de même ordre de paramètres. (voir figure 2.5). Partant d'un exemple, où on considère la classe *Etudiant* et en définissant un seul constructeur (voir figure 2.6). Comme il est indiqué par la figure, on a défini la classe *Etudiant* comme d'habitude. Dans la partie interne on a déclaré les attributs *Nom* et *Prénom*, et dans la partie externe ou publique on peut déclaré des différents accesseurs et manipulateurs, et également des méthodes qu'on a les bien choisit. Ainsi, on a rajouté en plus le constructeur *Etudiant* comme une méthode particulière dédiée pour la création des objets. Bien entendu, le constructeur *Etudiant* est fait partie de l'interface publique pour pouvoir être invoquer par les programmeurs utilisateurs de la classe *Etudiant*. Le constructeur en plus d'être accessible de l'extérieur de la classe, il reçoit deux paramètres *nom* et *prénom* pour pouvoir initialiser les attributs *Nom*

```
class Etudiant {

 private String Nom;
 private String Prénom;

 private double Note_Examen;
 private double Note_TD;

 Etudiant(String nom, String prénom, double note_TD){
  Nom = nom;
  Prénom=prénom;
  Note_Examen = note_TD;
  Note_TD=note_TD;
 }
 Etudiant(String nom, String prénom, double note_Examen,
   double note_TD){
  Nom = nom;
  Prénom=prénom;
  Note_Examen = note_Examen;
  Note_TD=note_TD;
 }
}
```

FIGURE 2.5 – Surcharge des constructeurs

et *Prénom*. Alors, pour déclarer et créer au même temps une instance en initialisant des attributs bien choisit, on doit adopte le constructeur de la classe selon la syntaxe indiquée par la figure 2.7. Par exemple, pour la classe *Etudiant*, on peut déclarer une variable d'instances de la classe *Etudiant* en faisant appel le constructeur *Etudiant* qui prend deux paramètres pour l'initialisation les attributs *Nom* et *Prénom* des objets de la classe *Etudiant* (voir figure 2.8). Alors, on a créer effectivement un objet, et également initialisé directement ses attributs avec une appelle du constructeur *Etudiant* en passant la valeur "Lamouri" au paramètre *nom* pour initialiser l'attribut *Nom*, et la valeur "Amine" au paramètre *prénom* pour initialiser l'attribut *Prénom* de l'instance de *E*.

```
class Etudiant {

 private String Nom;
 private String Prénom;

 private double Note_Examen;
 private double Note_TD;

 // Accesseurs/modificateurs si nécessaire
 //.........

  public Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
  Note_Examen = note_Examen;
  Note_TD=note_TD;
 }

 public double moyenne() {
  return ((Note_Examen*2 + Note_TD)/3);
 }
}
```

FIGURE 2.6 – Gestion artificielle : Définition du constructeur

```
IdentificateurClasse instance =
 new IdentificateurClasse (paramètre1, paramètre2, ...);
 /* où paramètre1, paramètre2, ... sont les valeurs des
  paramètres du constructeur.*/
```

FIGURE 2.7 – Appel aux autres constructeurs

2.1.2 Constructeur par défaut

Comme toute méthode, un constructeur peut avoir aucun paramètre. Un constructeur sans que l'on fournisse aucune valeur pour l'initialisation des attributs. Dans ce cas, on parle de constructeurs par défaut. Alors, un constructeur par défaut est

```
class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E = new Etudiant("Amine", "Lamouri");
  // invocation du constructeur de la classe Etudiant
  // ...
 }
}
```

FIGURE 2.8 – Gestion artificielle : Appel du constructeur

tout simplement est une méthode particulière qui ne possède pas de paramètres, et il a pour objectif d'initialiser les attributs d'un objet par dés valeurs par défaut (*false* pour le type *boolean*, 0 pour le type *int*, 0.0 pour le type *double*, *null* pour les objets). Pour la classe *Etudiant* par exemple, on peut définir trois constructeurs afin d'offre trois variantes d'initialisation des objets de type *Etudiant* (voir figure 2.9). Le constructeur par défaut qui est sans paramètre permet d'initialiser les attributs *Nom* et *Prénom* à des valeurs par défaut. Concrètement, après avoir déclaré une variable d'instances par exemple E avec le type *Etudiant*, elle sera initialisée en invoquant le constructeur par défaut par le biais de la tournure *new Etudiant()*. Alors, le faite de spécifier l'appelle du constructeur avec une liste d'arguments vide, il signifie qu'on est en train de créer un objet par défaut (un objet dont les déférents attributs sont initialisés avec des valeurs par défauts) pour la classe *Etudiant*. Les attributs *Nom*, *Prénom* de tels objets soient initialisés avec les valeurs par défauts respectivement *"Amine"*, *"Lamouri"*. Cependant, les deux autres constructeurs de la classe *Etudiant* ont une liste d'arguments non vide, alors ils ne soient pas des constructeurs par défaut. Pour les utiliser, il faut bien spécifier les bonnes valeurs de leurs paramètres. Si on souhaite par exemple d'invoquer le constructeur avec deux paramètres, il est nécessaire d'utiliser la tournure particulière *Etudiant* $E =$

```
class Etudiant {

 private String Nom;
 private String Prénom;

 Etudiant(){
  Nom = "Amine";
  Prénom="Lamouri";
 }

 Etudiant(String nom){
  Nom = nom;
 }
 Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
 }
}
```

FIGURE 2.9 – Des constructeurs pour la classe *Etudiant*

new Etudiant("Lamouri", "Amine"). Alors, de la même manière qu'auparavant, on doit déclarer une variable E de type *Etudiant*, et au même temps on invoque, le constructeur par la même tournure mais on doit fournir en argument cette fois-ci les valeurs appropriées pour initialiser les attributs *Nom* et *Prénom*.

2.1.3 Constructeur par défaut par défaut

Cette section a pour objectif de présenter un constructeur particulier lors que dans une classe aucun constructeur ni spécifié de manière explicite. Puisque la création et l'initialisation des objets est une tâche beaucoup plus indispensable, alors un constructeur doit être automatiquement généré pour pouvoir crée et initialiser les objets de la classe, lors que le programmeur concepteur ne définie explicitement aucun constructeur. Il s'agit alors d'un constructeur par défaut qui qui ne dispose d'aucuns paramètres, et qui initialise des attributs avec de valeurs par défauts. Ce

constructeur est lui-même par défaut, parce qu'il est nécessairement disponible à défaut lors de l'absence de tout autre constructeur explicite dans la classe. Ce qu'on appel communément de *constructeur par défaut par défaut*. Un constructeur par défaut par défaut fait un minimum de travail d'initialisation, il va initialiser tous les attributs, qui ne sont pas initialisés au niveau de la déclaration, avec des valeurs par défaut ($false$, 0, 0.0, et *null* pour les types respectivement *boolean*, *int*, *double*, *class*). Supposé qu'on définisse une classe *Etudiant* comme d'habitude avec les attributs *Nom* et *Prénom* de type *String* (qui ont comme valeurs des objets de la classe *String*), ainsi que les attributs $Note\_Examen$ et $Note\_TD$ de type *double* (voir figure 2.10). Évidemment, dans la classe *Etudiant*, on a aucun constructeur

```
class Etudiant {

 String Nom;
 String Prénom;
 double Note_Examen;
 double Note_TD;
}
```

FIGURE 2.10 – Constructeur par défaut par défaut

explicite, alors il existe systématiquement un constructeur par défaut par défaut de la classe *Etudiant* pour initialiser des instances de type *Etudiant*. Par exemple, on peut déclarer une variable avec le type *Etudiant*, et l'initialiser avec une instance en se basant sur le constructeur par défaut par défaut qui n'a pas de paramètres, et qu'il est généré automatiquement et implicitement par le compilateur *Java*. Alors, après avoir l'invoqué, les attributs *Nom*, *Prénom*, $Note\_TD$, et $Note\_Examen$ seront initialisées au moyen des valeurs par défaut. La valeur *null* pour les attributs *Nom* et *Prénom* puisque ils sont d'attributs de type de complexe (objet de la classe prédéfini *String*). La valeur 0.0 pour les attributs $Note\_Examen$ et $Note\_TD$ car

il s'agit cette fois-ci d'un type primitive (double). Pour cet exemple, on'a pas indiqué les droits d'accès. Tandis que, pour une bonne encapsulation, il est nécessairement de déclarer en privé chacun des attributs. En outre, il faut bien noter qu'un constructeur par défaut par défaut n'est plus disponible une fois qu'un constructeur (que ce soit par défaut ou non) est définit explicitement dans une classe. Ainsi, si dans une classe il y a un constructeur qui n'est pas un constructeur par défaut, on doit fournir des valeurs d'initialisation pour construire un objet, puisque tout constructeur par défaut soit disparu, et également son invocation est illicite. Ce genre d'indisponibilité du constructeur par défaut par défaut est très intéressante pour bien pris le soin de la programmation explicite des constructeurs dans une classe, où on ne souhaite pas que langage de programmation (*Java*) on en glisse un constructeur par défaut systématiquement sans qu'on manifeste explicitement l'utilisation des constructeur par défaut.

Exemple d'illustration

Pour plus d'illustration, on va adresse trois variantes pour la même classe *Etudiant*. La différence entres ces variantes est relative à la manière de définition des constructeurs de cette classe (voir figure 2.11). Dans la première variante, la classe *Etudiant* ne possède aucun constructeur explicite, mais elle peut y prévoir éventuellement d'autre contenu. Dans la deuxième variante, elle contient un constructeur explicite (un constructeur par défaut) qui sert à initialiser les attributs *Nom* et *Prénom* au moyen de valeurs par défaut. Enfin, dans la troisième variante, la classe *Etudiant* dispose d'un constructeur explicite qui adopte deux paramètres pour initialiser convenablement les attributs *Nom* et *Prénom*. Pour chacune de ces variantes, on va discuter les points suivantes (voir table 2.1) :

1. la disponibilité de constructeur par défaut,
2. la création des objets selon la tournure *new Etudiant* (), et

```
class Etudiant {
 private String Nom;
 private String Prénom;
}//_____________________________________________________________

class Etudiant {
 private String Nom;
 private String Prénom;

 Etudiant(){
  Nom =  null;
  Prénom = null;
 }
}//_____________________________________________________________

class Etudiant {

 private String Nom;
 private String Prénom;

 Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
 }
}//_____________________________________________________________
```

FIGURE 2.11 – Disponibilité du constructeur par défaut

3. la création des objets selon la tournure *new Etudiant* (*Nom*, *Prénom*).

– Pour la première variante, où on n'a pas de constructeurs explicites, alors on aura un constructeur par défaut généré systématiquement par le langage de programmation (il s'agit de constructeur par défaut par défaut). Dans ce cas, la création d'objets selon la première tournure est licite, où on utilise un constructeur sans paramètre (*new Etudiant ()*). Alors, les attributs *Nom* et *Prénom* sont initialisés au moyen de valeurs par défaut (null). En outre, puisqu'on a uniquement le constructeur par défaut par défaut, alors on ne peut créer des objets que selon la première tournure. La deuxième tournure

Critère / Variante	1	2	3
La première variante	*OUI* Constructeur par défaut par défaut	*OUI*	*NON*
La deuxième variante	*OUI* Constructeur par défaut explicitement défini	*OUI*	*NON*
La troisième variante	*NON* Pas de constructeur par défaut	*NON*	*OUI*

TABLE 2.1 – Les trois variantes de constructeurs

qui adopte les deux paramètres (*Nom* et *Prénom* est alors illicite.

- Pour la deuxième variante, où on a dans la classe *Etudiant*, un constructeur par défaut qui est définit explicitement par le programmeur concepteur. Alors, on peut évidemment l'utiliser suivant la première tournure. Alors, on aura le même effet que le constructeur par défaut par défaut, parce que les attributs *Nom* et *Prénom* sont initialisés avec les mêmes valeurs utilisées par le constructeur par défaut par défaut. Puisqu'il n'existe pas dans la classe *Etudiant* aucun constructeur autre que le constructeur par défaut, alors dans ce cas, la deuxième tournure est toujours illicite.
- Enfin, pour la troisième variante, où on a un constructeur explicite qui possède deux paramètres pour pouvoir initialiser les attributs *Nom* et *Prénom*. Alors, le constructeur par défaut n'est plus disponible. Parce qu'une fois un constructeur explicite est définit (que ce soit par défaut ou non) le constructeur par défaut

par défaut soit disparait. Dans ce cas, la création des objets est licite au travers la deuxième tournure, par le biais d'initialisation des attributs *Nom* et *Prénom* au moyen des paramètres du constructeur. Par contre la deuxième tournure est illicite , puisque on a plus de constructeur qui n'a pas de paramètres.

Cependant, s'il existe plusieurs constructeurs dans une même classe, on peut favoriser la notion de réutilisation de codes, et également on offre certaines facilités pour la construction des objets et l'initialisation des attributs. On peut permettre dans la définition d'un constructeur l'appelle d'un autre constructeur de la même classe. Pour ce faire on adopte ce qu'on appelle la méthode d'invocation des constructeurs. Le langage *Java*, offre une méthode particulière *this* en fournissant convenablement les paramètres nécessaire à l'invocation du constructeur qu'on souhaite utiliser. Par exemple, pour la classe *Etudiant*, où on considère un constructeur explicite qui dispose une seule paramètre (*Nom*). On peut définir explicitement un autre constructeur qui dispose cette fois-ci deux paramètres (*Nom* et *Prénom*). Dans ce cas, on peut utiliser la définition du premier constructeur pour la définition du deuxième constructeur. Similairement, on peut utiliser la définition du deuxième constructeur dans la définition d'un constructeur par défaut (voir figure 2.12). Le constructeur qui a deux paramètres, au moyen de la tournure *this*(*nom*) utlise le constructeur qui dispose d'un seul paramètre en passant en guise la valeur de l'argument. Pareillement, le constructeur par défaut, au moyen de la tournure *this*() invoque le constructeur qui dispose deux paramètres en passant en guise la valeur "*Amine*" pour le premier paramètre, et également "*Lamouri*" pour le second paramètre. Il est bel et bien qu'il s'agit d'une tournure très efficace pour permettre la réutilisation convenable du code. Cependant, pour une définition d'un constructeur, il ne peut y avoir qu'une seule appelle à cette méthode particulière *this*, et également elle doit être la toute première instruction. En revanche, il est tout à fait possible d'initialiser des attributs au moment de leur déclaration sans passer par au-

```
class Etudiant {

 private String Nom;
 private String Prénom;

 Etudiant(String nom){
  Nom = nom;
 }

 Etudiant(String nom, String prénom){
  this(nom);
  Prénom=prénom;
 }

 Etudiant(){
  this("Amine", "Lamouri");
 }
}
```

FIGURE 2.12 – Appelles des Constructeurs

cun constructeur, ce qu'on appel communément de valeurs par défaut qui ne sont pas de valeurs par défaut propre à un langage de programmation, mais plutôt de valeurs attribué explicitement par le programmeur concepteur de la classe. Par exemple, au niveau de déclaration de l'attribut *Prénom*, on peut l'initialiser directement avec la valeur "*Amine*" comme une valeur par défaut explicite (voir figure 2.13). Lors qu'on invoque par exemple, le constructeur par défaut de la classe *Etudiant* (voir figure 2.13), où il n'y a aucune valeur explicitement attribuée aux *Nom* et *Prénom* au travers la constructeur. Alors, les valeurs par défaut qui soient utilisées. Au moyen de la construction par défaut, on aura un objet de type *Etudiant* dont l'attribut *Nom* est initialisé à la valeur par défaut *null*, et l'attribut *Prénom* également à "*Amine*". Si on suppose maintenant que le constructeur par défaut, initialise l'attribut *Nom* à "*Lamouri*" au lieu d'avoir un corps vide, (voir figure 2.14). Au travers du constructeur par défaut, on aura un objet de type *Etudiant*, dont l'attribut "*Nom*" soit

```
class Etudiant {

 private String Nom;
 private String Prénom = "Amine";

 Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
 }

 Etudiant(){
 }
}
```

FIGURE 2.13 – Initialisation par défaut des attributs : au niveau de déclaration

```
class Etudiant {

 private String Nom;
 private String Prénom = "Amine";

 Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
 }

 Etudiant(){
  Nom = "Lamouri";
 }
}
```

FIGURE 2.14 – Initialisation par défaut des attributs : au niveau de constructeur

initialisé avec la valeur "*Lamouri*", et également l'attribut "*Prénom*" soit attribué la valeur ("*Amine*") spécifiée au moment de la déclaration, parce que aucune valeur explicite n'est spécifié dans le constructeur en question. En effet, pour expliciter les intentions de programmation, il est mieux d'initialiser les attributs au moyen

des constructeurs plutôt que les initialiser au niveau de la déclaration. Puisque, si dans chacun des constructeurs on accompli l'entièreté d'initialisation nécessaires, il devient très facile de comprendre les initialisations réalisées par une simple lecture des corps de ces constructeurs. Tels aspects en programmation OO rendre intitule la vérification de toutes les initialisations implicites dans d'autres endroits de la classe.

2.1.4 Constructeur de copie

Cette section a pour objectif de présenter un autre constructeur particulier. Il s'agit d'un constructeur de copie. Comme son nom l'indique, un constructeur de copie permet de créer et initialisé un objet avec une copie d'un autre objet de même classe, en d'autre termes, il permet de créer un nouveau objet en initialisant tous ses attribues par les valeurs des attributs correspondants d'un autre objet. En effet, ce constructeur offre un moyen de création des copies d'instances extraordinaire. Par exemple, si on suppose qu'on a une variable d'instances $E1$ de la classe *Etudiant* initialisée avec un objet dont l'attribut *Nom* est initialisé à "*Lamouri*", et l'attribut *Prénom* est initialisé à "*Amine*". Si on veut maintenant initialiser la variable d'instances $E2$, qui est de la même classe *Etudiant*, avec une copie de l'instance de $E1$. Alors, On souhaite de copier la valeur de l'attribut *Nom* de l'instance de $E1$ dans l'attribut *Nom* de l'instance de $E2$, et également la valeur de l'attribut *Prénom* de l'instance de $E1$ dans l'attribut *Prénom* de l'instance de $E2$. En revanche, les instances de $E1$ et $E2$ sont deux instances distinctes (voir figure 2.15). Alors on a carrément initialisé l'instance de $E2$ avec une copie de selle de $E1$, en

```
Etudiant1 E1 = new Etudiant1("Amine", "Lamouri", 13.0, 17.0);
Etudiant1 E2 = new Etudiant1(E1);
```

FIGURE 2.15 – Création d'une copie d'instance

copiant tour a tour toutes les valeurs des attributs. Grâce au constructeur de copier, on peut réaliser ce genre de création des nouveaux instances. Donc, Le constructeur de copier, c'est une méthode particulière qui sert à créer une nouvelle instance, et également l'initialiser avec une copier d'une autre instance de la même classe. L'identificateur de cette méthode particulière doit être identique à l'identificateur de la classe, puisqu'il s'agit d'un constructeur de classe. Ainsi, sa liste de paramètres est réduite en un seul paramètre qui est une autre instance de la même classe. D'une manière très technique la syntaxe de base pour la définition d'un constructeur de copie est fournit dans la figure 2.16). Partant d'un exemple concret. Dans la classe

```
IdentificateurClasse(IdentificateurClasse autreObjet) { ... }
```

FIGURE 2.16 – Syntaxe du constructeur de copie

Etudiant, on peut définir un constructeur dont la liste de paramètres est réduite à un paramètre unique *autreObjet* de type *Etudiant*(voir figure 2.17). En effet, il

```
class Etudiant {

 private String Nom;
 private String Prénom;

 Etudiant(Etudiant etudiant){
  Nom = etudiant.Nom;
  Prénom=etudiant.Prénom;
 }
}
```

FIGURE 2.17 – Constructeur de copie

permet de récupérer une autre instance de la classe *Etudiant*, pour initialiser les attributs de la nouvelle instance. Alors, on va copier la valeur de l'attribut *Nom*

```
class GestionArtificielle {

 public static void main (String[] args) {
 Etudiant E1 = new Etudiant ("Lamouri", "Amine");
 Etudiant E2= new Etudiant(E1);
 }
}
```

FIGURE 2.18 – Affectation au moyen du constructeur de copie

de l'instance de *autreObjet* reçu en paramètre dans l'attribut *Nom* de la nouvelle instance qu'on souhaite d'initialiser. Ainsi, on va copier la valeur de l'attribut *Prénom* de l'instance de *autreObjet* dans l'attribut *Prénom* de l'instance qu'on veut initialiser. Alors, on peut défnir une instance pour la variable $E1$ avec un attribut *Nom* de valeur "*Lamouri*", et un attribut *Prénom* de valeur "*Amine*". La nouvelle instance affecter pour $E2$ est initialisée au moyen du constructeur de copie. La tournure *Etudiant E2= new Etudiant(E1)* (voir la figure 2.18) lance l'appelle du constructeur de copier en passant au paramètre *autreObjet* l'instance de $E1$. Alors, c'est bien la construction d'une novelle instance comme une valeur pour la variable $E2$. En fait, l'appelle de ce constructeur implique l'affectation de la valeur de l'attribut *Nom* de l'instance de $E1$ dans l'attribut *Nom* de l'instance qu'on a en train de construire (la nouvelle instance de $E2$), et également l'affectation de la valeur de l'attribut *Prénom* de l'instance de $E1$ dans l'attribut *Prénom* de celle de $E2$.

2.2 Manipulation des Objets

Très souvent les langages de programmation, et particulièrement le langage *Java*, s'ils considèrent des données complexes (comme les objets), ils les manipulent au moyen de la notion de références. En d'autres termes, lorsqu'on déclare une variable

d'objets (c'est-à-dire de type classe), cette variable ne contiendra pas comme valeur un objet, mais plutôt une référence (une adresse mémoire) vers un objet. Par exemple, la valeur d'une variable d'instances dont le type est *Etudiant* c'est, en réalité, une référence vers un objet de la classe *Etudiant* avec ses propres attributs. Dans cette section on va présenter la manipulation des objets au travers des références pour la affectation, la comparaison, et l'affichage de ces objets, et également leur destruction en fin de vie.

2.2.1 Affectation d'objets

Sachant que les objets soient manipulés au travers des références, si on souhaite créer un autre objet à partir d'un objet préexistant qu'est de même classe, l'opérateur d'affectation usuel ne permettre pas d'avoir deux objets distincts en mémoire. Contrairement au variable primitives, en *Java*, si on a créé une instance pour $E1$ de type *Etudiant*, alors l'affectation $E2 = E1$ ne permettre pas d'atteindre deux valeurs distinctes (deux objets distincts) en mémoire (voir figure 2.19). Parce que,

```
class GestionArtificielle {

 public static void main (String[] args) {
 Etudiant E1 = new Etudiant ("Lamouri", "Amine");
 Etudiant E2= E1;
 }
}
```

FIGURE 2.19 – affectation de référence d'objet

lorsque on exécute la tournure *E1 = new Etudiant("Lamouri", "Amine")*, on aura en mémoire une référence $ref1$ vers l'objet de la classe *Etudiant* dont les valeurs des attributs *Nom* et *Prénom* sont respectivement "Lamouri" et "Amine". Ensuite, si on exécute l'affectation *E2 = E1*, alors on va affecter la référence $ref1$ de $E1$ dans

$E2$, c'est-à-dire on a uniquement copié la valeur contenu dans $E1$ vers $E2$. Alors, on aura dans $E2$ la référence $ref1$ vers le même objet en mémoire. En d'autres termes, on aura orienté vers le même objet de $E1$. Dans ce cas, les variables $E1$ et $E2$ ne désignent pas deux objets distincts en mémoire. Alors, toute manipulation sur l'objet au travers de $E1$ est également visible au travers de $E2$, et vice-versa. Par conséquence, la modification de la valeur de l'attribut Nom au moyen de $E2$ par le bais de l'instruction $E2.Nom = "Gharbi"$, est aussi visible pour $E1$. Concrètement en *Java*, si on veut que l'instance de $E2$ soit une copie distincte de celle de $E1$, alors on doit utiliser un constructeur usuel (voir figure 2.20), ou un constructeur de copie (voir figure 2.21), plutôt d'adopter l'opérateur d'affectation. En *Java*, on peut adopter ainsi une méthode particulière dite la méthode *clone*.

Utilisation de la méthode clone()

Contrairement à certains langages de programmation où l'opération d'affectation usuelle peut parfaitement créer des copies distinctes d'objets, le langage *Java* n'offre pas automatiquement des copies distinctes d'objets, puisqu'on a pas systématiquement un constructeur de copie. En d'autres termes, on à pas la possibilité de copier des objets sans la définition explicite de ce constructeur particulier, mais plutôt on doit le définir explicitement dans la partie interface d'utilisation si on veut réaliser une construction de copier. Cependant, ce genre de constructeur n'est pas la seule manière de copie des objets en *Java*, mais il existe d'ailleurs une autre méthode particulière très souvent utilisée. Il s'agit en fait, de la redéfinition de la méthode clone de la classe *Objet* prédéfini par le langage *Java* (voir chapitre 3). Partant d'un exemple où on souhaite copier les instances de la classe *Etudiant* au moyen de la méthode *clone*. Dans ce cas, on doit redéfinir tout d'abord la méthode *clone* dans la partie interface de la classe *Etudiant* (voir figure 2.22). Alors, si on veut par exemple copier l'instance de $E1$ dans $E2$, il suffit d'appeler sur l'instance de $E1$ la méthode

```
class Etudiant {

 private String Nom;
 private String Prénom;

 public Etudiant(String nom, String prénom){
  Nom = nom;
  Prénom=prénom;
 }

 public Etudiant copier(){
  return new Etudiant(Nom, Prénom);
 }
}//_____________________________________________________________

class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E1 = new Etudiant("Amine", "Lamouri");
 Etudiant E2 = E1.copier();
 }
}
```

FIGURE 2.20 – Utilisation de constructeur usuel

clone sans aucune nécessité de passage de paramètres comme il est indiqué par la figure, puisque la méthode *clone* est un membre de la classe. Par conséquence, elle a le droit d'accés a toutes les attributs de la classe. Contrairement au constructeur de copie, la méthode *clone* ne nécessite aucune spécification explicite des attributs a copiés, car elle sert à copier tous les attributs d'un objet. En effet, on peut envisager d'autres en-têtes pour la définition de la méthode *clone*. Par exemple, on peut adopter la classe prédéfinie *Object* pour la définition du type de retour (voir figure 2.23). En d'autres termes, il est également possible de définir la méthode *clone* en retournant un objet de type *Object* (voir chapitre 3).

```
class Etudiant {

 private String Nom;
 private String Prénom;

 public Etudiant(Etudiant etudiant){
  Nom = etudiant.Nom;
  Prénom=etudiant.Prénom;
 }
}//_____________________________________________________________

class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E1 = new Etudiant("Amine", "Lamouri");
 Etudiant E2 = new Etudiant(E1);
 }
}
```

FIGURE 2.21 – Utilisation de constructeur copier

2.2.2 Comparaison d'objets

Après avoir présenté, le principe de manipulation des objets au travers des références, et également les aspects qui doivent être pris en comptes lors des affectations des objets, dans cette section on va adresser la comparaison des objets en tenant en compte également les mêmes aspects. Sachant que les objets soient manipulés au travers des références, alors si on souhaite faire une comparaison entres deux objets qui doivent être de même classe, on doit vérifier alors tous ses attributs tour à tour, et s'ils ont les mêmes valeurs pour tous leurs attributs, on dit qu'ils sont équivalents (égaux). Puisque, si on essaye de les comparer au moyen de l'opérateur de comparaison usuel, alors les références vers ces objets qui sont comparer et elles ne sont pas les valeurs de leurs attributs. Par exemple, supposé qu'on a deux objets de la même classe *Etudiant* dont les valeurs des attributs sont égales. Alors les deux objets de la

```
class Etudiant implements Cloneable {

 private String Nom;
 private String Prénom;

 public Etudiant clone(){
  Etudiant E=null;
  try{
   E=(Etudiant) super.clone();
  } catch(CloneNotSupportedException e){
      /* gestion des erreurs */ }
  return E;
 }
}//_____________________________________________________________

class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E1 = new Etudiant("Amine", "Lamouri");
 Etudiant E2 = E1.clone();
 }
}
```

FIGURE 2.22 – Utilisation de la méthode clone : instance de la classe *Etudiant*

classe *Etudiant* ayant la même valeur pour l'attribut Nom, et également la même pour l'attribut *Prénom*. En *Java*, des instances déférentes en mémoire ont nécessairement des références distincts (par exemple $ref1$ pour la première instance, et $ref2$ pour la deuxième). Si on essaye de comparer les instances de la classe *Etudiant* au travers l'opérateur de comparaison usuel (l'opérateur ==), alors les valeurs des attributs ne sont pas comparées entre elles, mais plutôt les références $ref1$ et $ref2$. Évidemment, lorsque de la création d'un objet de classe par le biais de l'appelle de constructeur, on récupère dans une variable (par exemple $E1$) la référence vers cette objet par exemple $ref1$. Alors, la variable $E1$ contient la valeur $ref1$ ($ref1$ c'est une adresse vers un objet de la classe *Etudiant* avec ses propres attributs).

```
class Etudiant implements Cloneable {

 private String Nom;
 private String Prénom;

 public Object clone(){
  Object O=null;
  try{
   O = super.clone();
  } catch(CloneNotSupportedException e){
     /* gestion des erreurs */ }
  return O;
 }
}//_____________________________________________________________

class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E1 = new Etudiant("Amine", "Lamouri");
 Etudiant E2 = (Etudiant) E1.clone();
 }
}
```

FIGURE 2.23 – Utilisation de la méthode clone : instance de la classe *Object*

Pareillement pour le deuxième objet, la variable $E2$ dépose une adresse $ref2$ vers un autre objet de la même classe *Etudiant* avec ses propres attributs dont les valeurs sont les mêmes que selles de l'autre objet). On engendre à chaque appelle de constructeur une référence distincte. Alors, même si les objets ont les mêmes valeurs d'attributs, les valeurs des variables $E1$ et $E2$ sont totalement différentes, puisque elles ont des références distinctes qui pointent vers des zones mémoires différentes. Par conséquence, qu'on doit comparer les données de ces objets et ne sont pas les contenus des variables de références (comme $E1$ et $E2$), puisque le résultat de la comparaison des objets distincts au moyen des références donne toujours la valeur booléenne $false$. En d'autres termes, lorsqu'on veut comparer des objets, ce n'est

pas tant les adresses des objets qu'on souhaite les manipuler, mais plutôt leurs contenus (les contenues de leurs attributs). Pour ce faire, on doit adopte une méthode nécessairement définit par le programmeur concepteur de la classe. Tel programmeur doit concrètement offre une méthode de comparaison qui sert à vérifier tour à tour tous les attributs des objets. L'un des en-têtes envisagés pour la déclaration de telle méthode est nécessairement l'implication des données de même classe. Typiquement, on peut imaginer qu'on a une instance convenablement initialisé, et si on veut la comparer avec une autre instance de même classe, alors de telle méthode de comparaison doit être invoquée sur une des instances en fournissant l'autre instance en argument. Partant un exemple, où on souhaite définir une méthode dite *equals* qui sert à comparer les objets de types *Etudiant*. La méthode *equals* prend comme paramètre E de le classe *Etudiant* (voir figure 2.24). Dans la méthode *equals* on a spécifie des critères au travers lesquels en détermine que les deux objets sont égaux. La valeur de sortie de la méthode est de type *boolen*. Elle return *true* si les deux objets sont considérés comme égaux, et $false$ pour le cas contraire. Très souvent, on doit prendre la prudence de vérifier si l'objet passer en argument avec laquelle on veut comparer l'objet courant ne soit pas *null*, sinon on peut avoir un échec de comparaison. Si l'objet passer en argument est *null*, alors on ne les considèrent pas comme égaux, puisque l'objet au travers lequel la méthode *equals* est invoquée est forcément non *null*. Tandis que, si l'objet fournit en argument est non *null*, alors on retourne *true* seulement si les objets ont les mêmes valeurs pour tous les valeurs des attributs *Nom* et *Prénom*. Dans ce cas, on aura le message *les deux objets de la classe Etudiant sont identiques* est affiché. Ce qui n'est pas le cas, lorsqu'on adopte l'opérateur de comparaison usuel == pour la comparaison des ces objets. En effet, plusieurs en-têtes sont possibles pour la définition de la méthode *equals*. Particulièrement en *Java*, on apeut adopter la classe prédéfinie *Object* pour la définition du paramètre de cette méthode. En d'autres termes, il est possible également de définir

```
class Etudiant {

 String Nom;
 String Prénom;

 Etudiant( String nom, String prénom){Nom = nom; Prénom =
   prénom;}

 boolean equals(Etudiant E) {
  boolean Egaux=false;
  if (E == null) Egaux = (Nom == E.Nom) && (Prénom == E.Prénom
   );
 return Egaux;
 }
}//-------------------------------------------------------------

class GestionArtificielle {

 public static void main(String[] args) {

  Etudiant E1=new Etudiant("Amine","Lamouri");
  Etudiant E2=new Etudiant("Houari","Idrici");

  if (E1.equals(E2)) System.out.println("Les deux objets de la
    classe $Etudiant$ sont identiques");
  else System.out.println("Les deux objets de la classe
   $Etudiant$ ne sont pas identiques");
 }
}
```

FIGURE 2.24 – Comparaison d'objets

la méthode *equals* en passant en argument un objet de type *Object* (voir chapitre 3).

2.2.3 Affichage d'objets

Comme on a déjà présenté dans les sections présidentes, sachant que les objets soient manipulés au travers des références, des précautions doivent être prisent en compte lors de manipulation de ces objets. Cette section a pour bute d'adresser un

autre aspect de manipulation, il s'agit de l'affichage des objets. Par exemple, si on souhaite afficher les données des variables de type classe (des instances), alors on doit manipuler tous les attributs de ces instances. En outre, sachant qu'on manipule ces instance au travers des références, si l'on essaye de les afficher comme d'habitude (comme en faisant pour les données primitives), alors des références vers ces objets qui sont affichées, et elles ne sont pas les valeurs de leurs attributs. Par exemple, supposé qu'on a une instance de la classe *Etudiant* avec ses propres valeurs pour les différents attributs (sa propre valeur pour l'attribut *Nom*, et sa propre valeur pour l'attribut *Prénom*). Lorsqu'on essaye d'afficher une instance de la classe *Etudiant* comme en faisant pour les variables primitives, les valeurs des attributs de l'instance ne sont pas affichées, mais plutôt la référence qui soit affichée. Évidemment, lors de la création d'une instance par le biais d'une appelle de constructeur, on peut récupérer dans une variable (par exemple E) une référence (par exemple ref) vers cette instance. Alors, dans la variable E on a la valeur ref (ref c'est une adresse vers un objet concret de la classe *Etudiant*). Bien qu'on souhaite d'afficher les données de l'instance référencées par la variable E, et non pas la valeur de cette variable E. En d'autres termes, lorsqu'on affiche des objets, ce n'est pas tant les adresses des objets qu'on les afficher, mais plutôt leurs contenus (les valeurs de leurs attributs). Pour ce faire, on doit définit une méthode dans la classe en question. Concrètement, on doit offre une méthode dédié pour afficher tour à tour tous les attributs selon les critères qui semblent sensés pour des membres de type classe. En d'autres termes, on doit définir la manière de constitution de la chaîne de caractères par cette méthode en favorisant son utilisation depuis l'extérieur. Telle méthode sert tout naturellement de construire une chaîne de caractère qui correspond à la représentation préférée lors de l'affichage d'un objet. Typiquement, on décide que la chaîne est construite par concaténation des valeurs des attributs (biens choisit) de l'instance en question afin de produire un affichage de nature à la fois lisible et préféré. Partant un exemple, où

on souhaite définir les méthodes *commeChainne* et *toString* qui serrent à afficher les objets de types *Etudiant*. Telles méthodes ne prennent aucun paramètres, et leur type de retour est une chaîne de caractères (voir figure 2.25). Ainsi, pour telles

```
class Etudiant {

 private String Nom;
 private String Prénom;

 Etudiant(String nom, String prénom) {
  Nom = nom;
  Prénom=prénom;
 }

 String commeChainne() {
  return "Le Nom : " + Nom + ", Le Prénom : " + Prénom;
 }

 String toString() {
  return "Le Nom : " + Nom + ", Le Prénom : " + Prénom;
 }
}
```

FIGURE 2.25 – Affichage d'objet

méthodes, on a spécifié les attributs (*Nom* et *Prénom*) impliqués par l'opération d'affichage, et on n'a pas prend la précaution de vérifier si l'objet à afficher est *null*, puisque telles méthodes doivent être appelées sur un objet non *null*. Alors, lors de l'appelle d'une de cettes méthodes, on aura l'affichage naturel et lisible des valeurs des attributs *Nom* et *Prénom* qui sont propre à l'instance sur laquelle on a l'appelée. Par contre, pour un affichage usuel au moyen de la variable E uniquement, par le biais de la tournure *System.out.println(E)*, on aura un affichage un peu étrange (une sorte d'adresse mémoire). Cependant, contrairement à la méthode *commeChainne*, la méthode *toString* est automatiquement invocable pour assimiler une sorte de conversion d'une instance de la classe *Etudiant* en une représentation sous forme de

chaîne de caractère, puisqu'il s'agit d'une redéfinition de la méthode *toString* de la classe *Object* (voir chapitre 3).

2.2.4 Fin de vie d'objets

Après avoir présenté, les déférentes opérations de création, d'initialisation, et de manipulation (l'affectation, la comparaison et l'affichage) des objets au travers de la notion de références, et également les précaution qui doivent être pris en comptes pour de ce genre de manipulations, dans cette section on va adresser un aspect bien particulier qui est la fin de vie des objets (des données pour des variables de type classe) sachant qu'ils soient manipulées au travers des références. En *Java*, lorsque la référence associée à une instance n'est plus utilisable dans un programme, alors elle devient en état de fin de vie. En d'autres termes, si une instance n'est plus exploitable, alors elle sera détruite systémiquement. Partant d'un exemple où on souhaite créer une instance de la classe *Etudiant* (voir figure 2.26). Dans la mé-

```
class GestionArtificielle {

 public static void main(String[] args) {
  afficheObject();
 }

 static void afficheObject() {
  Etudiant E = new Etudiant ("Amine","Lamouri");
  System.out.println(E);
 }
}
```

FIGURE 2.26 – Fin de vie d'objet

thode *main*, on a utilisé la méthode auxiliaire $affichеObject$. Dans la méthode $affichеObject$ on a créé un objet pour la variable E de type *Etudiant*" en l'initialisant de manière appropriée, et en affichant les valeurs de ses attributs *Nom*

et *Prénom*. La référence créée dans la méthode $affichеObject$ qui est associée à cet objet n'est utilisable nulle part ailleurs. Ainsi, on n'a pas utilisé cette référence pour une autre variable accessible plus globalement. Alors, une fois l'exécution de la méthode $affichеObject$ soit terminé, il devient impossible d'avoir accéder à la référence créée localement dans la méthode. Dans ce cas l'objet pour E créé localement dans la méthode $affichеObject$ est devient en état fin de vie, puisque la référence qui lui est associée n'est plus utilisable. Alors, la mémoire qui lui est associée doit être libérée afin d'être allouée à nouveau pour d'autres données. En effet, l'allocation et la désallocation de mémoire doivent être explicitement prises en charge par les programmeurs. Cependant, le langage *Java* adopte systémiquement le processus (*ramasse-miettes* ou *garbage collection*) pour la récupération transparente de la mémoire pour les objets qui ne sont plus utilisable. Alors, un programmeur en *Java* ne doit pas libérer explicitement la mémoire qu'il utilise. En d'autres termes, la destruction explicite des objets créés n'est plus obligatoire en *Java*, surtout pour un programmeur débuton.

Chapitre 3

Enrichissement et Spécialisation

Sommaire

Après avoir vu les notions d'encapsulation et d'abstraction, nous abordons dans ce chapitre un troisièm concept fondamental de l'approche OO qu'est l'héritage. Supposer qu'on souhaite réaliser une application pour la gestion d'une université où on considère des étudiants, des enseignants, des enseignants chercheurs, des agents d'administration, des agents de sécurité et bien d'autres. Si on d'adopte l'approche OO, alors on doit les modéliser au moyen de classes. Une classe pour les étudiants, une classe pour les enseignants, une classe pour les agents d'administration, ainsi de suite. Subséquemment, on peut caractériser un étudiant par un nom, un prénom, un

groupe, une section, un niveau, une spécialité. Ainsi, il peut assister des cours, des TD, et des TP. Il peut aussi faire des absences, changer son groupe, changer sa section. En outre, un étudiant doit faire des examens de contrôles, des examens finaux, et il peut faire des examens de rattrapages, etc. Pareillement, un enseignant peut être caractériser par un nom, un prénom, un grade, un diplôme, une spécialité, une ancienneté, un bureau, etc. Un enseignant assurer certainement des cours, des TD, et des TP. Ainsi, il doit établie des supports de cours, des fiches de TD, des fiches de TP, et des fiches des examens. En outre, un enseignant peut faire des absences, des séances de rattrapage pour des cours, des séances de rattrapage pour des TD, etc. On peut également considérer un enseignant chercheur, comme une sorte d'enseignant, qui peut avoir en plus une équipe de recherche, un domaine de recherche, un laboratoire, des projets de recherche, etc. il peut faire des contributions scientifiques, il peut diriger des équipes de recherches. Ainsi, il peut rédiger des états d'avancements des projets de recherche, des papiers, et il peut avoir des participations aux conférences nationales et internationales. Similairement, un agent d'administration a un nom, un prénom, un grade, un diplôme, un service, une ancienneté, un bureau etc. Un agent d'administration peut faire ainsi la saisie, l'impression, et l'affichage des emplois du temps, des listes des groupes, des annonces des examens, etc. Egalement, un agent de sécurité peut être caractériser par un nom, un prénom, une ancienneté, etc. Il fait la garde aux niveaux des départements de facultés (le contrôle des salles, des emplies, des labos), les secrétariats, les bibliothèques, la garde de nuit, etc. Par conséquence, si on considère uniquement l'encapsulation et l'abstraction de l'approche OO, on aura une représentation conceptuelle qui ne favorise pas la notion de réutilisation de code (voir figure 3.1). On nécessite beaucoup de temps pour l'écriture de ces données et ces actions, et on rencontra alors des problèmes très sérieux de maintenance. Supposer que :

– l'attribut TEL ne soit plus de type *String*, mais plutôt un nombre entier,

Etudiant
Nom, Prénom, Address, TEL Spécialié, Niveau Section, Groupe
Il parle, Il téléphone, Il marche, Il s'assoit Il assiste des cours, des Td, des TP Il change le groupe, la section Il s'absente, et il s'examine

Enseignant
Nom, Prénom, Address, TEL Grade, Diplôme
Il parle, Il téléphone, Il marche, Il s'assoit Il assure des cours, des TD, des TP Il établit des fiches de TD, de TP, d'examen Il s'absente, et il rattrape des cours, des TD, examens

Enseignant Chercheur
Nom, Prénom, Address, TEL Grade, Diplôme Laboratoire, Equipe, Domain, Projets
Il parle, Il téléphone, Il marche, Il s'assoit Il assure des cours, des TD, des TP Il établit des fiches de TD, de TP, d'examen Il s'absente, rattrape des cours, des TD, examens Il rédige des états d'avancements des projets de recherché Il rédige des papiers Il participe aux conférences nationales Il participe aux conférences internationales Il fait de contribution scientifique Il Supervise des équipes de recherché.

Agent d'administration
Nom, Prénom, Address, TEL Ancienneté Bureau, Service
Il parle, Il téléphone, Il marche, Il s'assoit Il fait la saisie des emplois du temps, listes des groupes, annonces des examens Il imprime des emplois du temps, listes des groupes, annonces des examens Il affiche des emplois du temps, listes des groupes, annonces des examens

Agent de sécurité
Nom, Prénom, Address, TEL Ancienneté Service
Il parle, Il téléphone, Il marche, Il s'assoit Il fait la garde de nuit Il fait la garde de jour

FIGURE 3.1 – Pseudo représentation conceptuelle du personnel d'une université

– on adopte plus l'attribut TEL, mais plutôt un mail pour contacter les différents personnes,

– l'attribut *Adresse* ne soit plus de type *String*, mais plutôt un type beaucoup plus complexe dont les éléments sont bien la rue, la ville, la wilaya, le paye.

Alors, on est dans l'obligation de revoir toutes les classes implémentées pour pouvoir rependre à ce genre d'adaptation. Il s'agit vraiment une mauvaise façon de conception de classes. Afin d'être en mesure de bonne maintenabilité, l'approche OO adopte un concept très important, ce que l'on appel en termes de jargon héritage.

3.1 Notion d'héritage

La notion d'héritage en programmation OO permet de résoudre des problèmes d'ordre maintenabilité, en regroupant parfaitement les parties, les attributs et méthodes, communes dans des classes beaucoup plus abstraites dites des *super − classes*. Il s'agit des classes regroupant des caractéristiques générales, lesquelles soient enrichies et spécialisées par d'autres classes bien spécifiques. Grace à ce principe de regroupement des données génériques, la conception précédente peut être reconçue et améliorée. Typiquement, on aura une classe *Personne* qui permettre de répondre à un des problèmes de la conception présentée auparavant où on n'a pas la classe *Personne* (voir figure 3.1). En prévoyant avec un lien d'héritage, on pourrait avoir une classe *Personne* qui regroupe l'ensemble des données communes (Nom, Prénom, Adresse, TEL, etc.), et également des opérations (méthodes) effectuer par une *Personne* générique (Il parle, Il téléphone, Il marche, Il s'assoit, etc.). Il s'agit d'une classe générique et plus abstraite qui regroupe l'ensemble des caractéristiques communes aux étudiants, aux enseignants, aux enseignants chercheurs, aux agents de sécurité, etc. On a définit la classe *Personne* à partir de quelle on hérite les classes *Etudiant*, *Enseignant*, *EnseignantChercheur*, et la classe *AgentsScurit*. En outre, chacune de ces classes a ses propres données spécifiques. Par exemple, la classe *Etudiant* en plus les données hérités de la classe *Personne*, elle a les attributs *Section* et *Groupe*. La calasse *Enseignant* possède l'attribut *Grade* en plus les don-

nées génériques modélisées dans la classe *Personne*. On a ainsi, une spécialisation avec l'opération *"Il fait la garde de nuit"* pour la classe *AgentsScurit*. On peut avoir encore une autre extension supplémentaire de la classe *Enseignant*, un héritage de plus, pour concevoir la classe *EnseignantChercheur* en modélisant un enseignant qui a en plus la caractéristique spécifique *Equipe* (voir figure 3.2). Selon l'approche

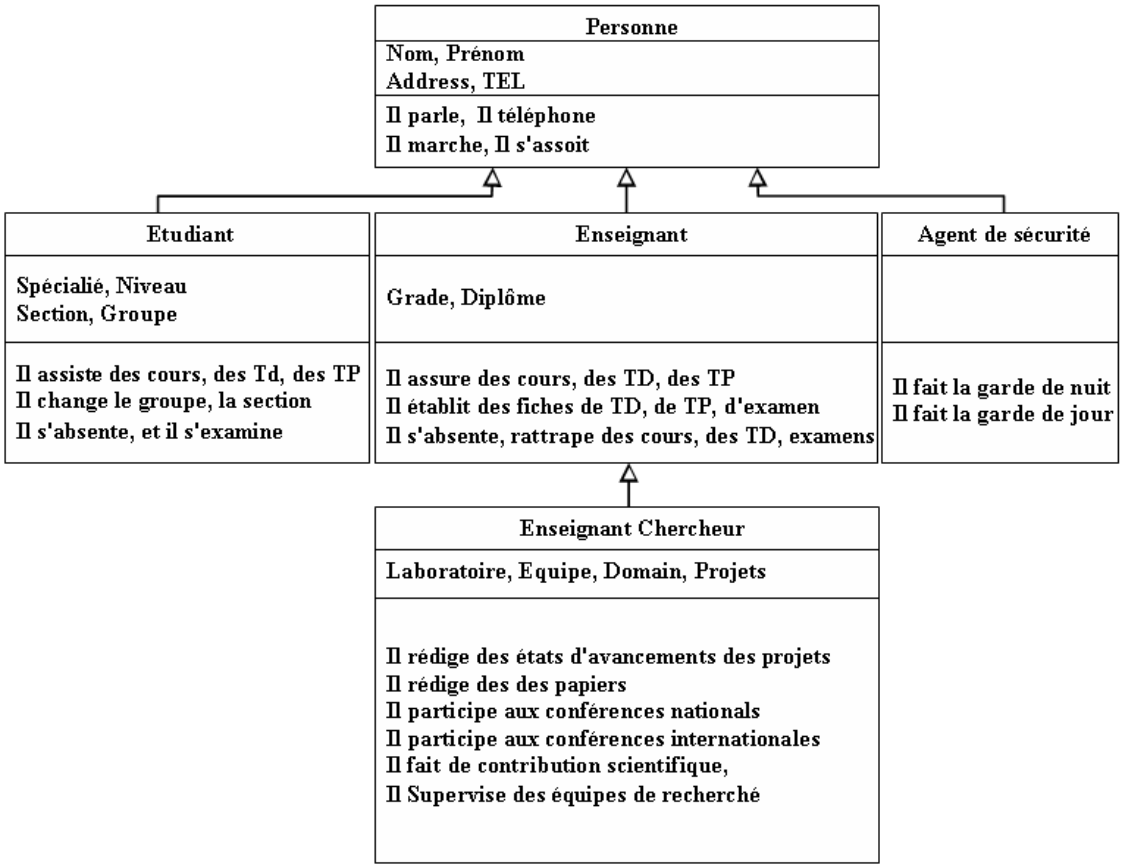

FIGURE 3.2 – Représentation conceptuelle du personnel d'une université

OO, l'héritage modélise la relation $est-un$ en outrepassant la duplication de codes (d'attributs et de méthodes) afin de favorise la simplicité de la maintenance. En outre, l'héritage permet de créer des classes beaucoup plus spécialisées par le biais d'enrichissement de données. En OO, une super-classes est une classe générique à partir de quelle on dérive des sous-classes bien spésifiques. Les sous-classes héritent de la super-classe ses d'attributs et ses méthodes, et on peut définir d'autres attributs ou méthodes beaucoup plus spécialisées. Une spécialisation de méthodes dans des

sous-classes est une extension de la super-classe. En OO, si on a une sous-classe qui hérite d'une super-classe, alors la sous-classe est le même type que la super-classe. Par exemple, si on a déclaré une variable d'instance via la super-classe, et une autre variable au moyen de la sous-classe qui est hérité de la super-classe, alors on peut affecter une instance de la sous classe à la variable d'instance de la super-classe, puisque le type de l'instance de la sous classe est aussi un type de la super-classe. Par ailleurs, la sous classe va hériter de l'ensemble des attributs et de l'ensemble des méthodes de la super-classe, tandis que les constructeurs ne sont pas hérités. Alors, les attributs et les méthodes de la super-classe seront disponibles dans la sous classe sans qu'on ait besoin de les redéfinir explicitement à nouveaux. Ainsi, des attributs et des méthodes supplémentaires peuvent être définis dans la sous classe, et c'est ce qu'on appelle communément l'enrichissement. En outre, les méthodes héritées de la super-classe dans la sous classe peuvent être redéfinies à nouveaux, et c'est ce qu'on appelle la spécialisation. Partant d'un exemple, où on considère la super-classe *Personne*, et les deux sous-classes *Etudiant* et *Responsable*. Comme on a vu auparavant, lorsqu'une sous-classe (par exemple *Etudiant*) est créée à partir d'une super-classe (par exemple *Personne*) le type est automatiquement hérité. C'est à dire, une instance de la classe *Etudiant* est aussi une instance de la classe *Personne*, puisque tout étudiant est une personne. Supposer qu'on a une variable P de type *Personne*, et une variable E de type *Etudiant*. Pour une raison ou une autre, on peut considérer que l'instance de P qui est de type *Personne* c'est une instance de E qui est de type *Etudiant*. Techniquement, on peut tout à fait affecter l'instance de E qui de type *Etudiant* dans la variable P qui est de type *Personne*, et on écrit concrètement l'instruction $(P = E)$. Tandis que, on ne peut pas faire le contraire, on ne peut pas écrire $E = P$, puisque une personne (générique) n'est pas toujours un étudiant, on peut avoir des personnes qui peuvent être des responsables, ou bien des enseignants par exemple. Alors, on ne peut pas affecter une instance de

la classe *Responsable* à une variable de type *Etudiant*, ni qu'une instance de type *Etudiant* à une variable de *Responsable*, puisque un étudient ne peut être jamais un responsable et vis-ver-ça. Subséquemment, l'héritage est une relation orientée, c'est bien un étudiant est une personne. En revanche, on peut définir une fonction pour afficher une instance de la classe *Personne*, et on peut afficher également une instance de la classe *Etudiant* qui est manipuler en tant que instance de la classe *Personne*. On peut afficher une instance de la classe *Etudiant*, parce que une instance de la classe *Etudiant* est une instance de la classe *Personne*. L'héritage est un aspect très important qui permet de recevoir et contenir l'ensemble des attributs et des méthodes de la super-classe. Par exemple une instance de *Etudiant* va recevoir et contenir (hériter) l'ensemble des attributs et des méthodes de la classe *Personne*. Alors, si la classe *Personne* est définit avec les attributs *Nom*, *Prénom*, *Age*, et une méthode $affficher$. Alors dans la classe *Etudiant* on aura aussi les attributs *Nom*, *Prénom* et *Age*et on a également la méthode $affficher$ sans aucune nécessité de les redéfinir a nouveau, puisque la classe *Etudiant* hérite de la classe *Personne*. Supposer qu'on a affecté une instance de la classe *Etudiant* dans E, et une instance de la classe *Responsable* dans R, qui sont aussi des instances de la classe *Personne*. L'instance de E en tant que de type *Etudiant* va hériter la méthode $affficher$ (qui est dans l'interface publique de la classe *Personne*), et on va pouvoir l'appeler au moyen de E ($E.affficher()$). Alors, la méthode $affficher$ est disponible et accessible pour l'instance de E, uisque elle a hérité la méthode $affficher$ de la classe *Personne*. Similairement pour les attributs (qui ne sont pas privés), si on veut définir une méthode dans la classe *Etudiant*, on peut utiliser directement un attribut, dans cette méthode de la classe *Etudiant*, qui est hérité de la classe *Personne*. Il s'agit d'une conséquence automatique de l'héritage sans aucune obligation de rajouter de lignes de code. Cependant, en plus les attributs et les méthodes hérités qui sont utilisable dans les sous-classes, on peut encore rajouter

des attributs et des méthodes supplémentaires, et on parle alors de l'enrichissement. Egalement, on peut redéfinir des méthodes de la super-classe dans une les sous-classe,on parle dans ce cas de la spécialisation (des méthodes). Par exemple dans la classe *Etudiant*, on peut définir le nouveau attribut $Note_Examen$, ainsi on peut redéfinir la méthode $afficher$ dans la classe *Responsable* en impliquant en plus les attributs *Grade* et *Service*. En OO, l'héritage est un concept très important, il permet de mieux organiser le code, et également il favorise la clarté et la lisibilité de la conception. En outre, il permet de rendre explicite les relations structurelles et sémantiques existantes entre différentes classes. Par exemple, un étudiant est une personne, un enseignant chercheur est une sorte d'enseignant. Il permet ainsi d'outrepasser la réécriture repetitive de grandes portions de code, ce qu'on appelle de la redondance de code. Cependant, l'héritage doit être utilisé bien convenablement, on doit l'utiliser pour la représentation de la relation $est - un$, par exemple un étudiant est une personne. Tandis que, pour représenter la relation a ($possde$), on adopte la notion d'encapsulation, et non pas selle d'héritage, puisque on ne peut pas dire qu'un étudiant est une note d'examen par exemple, mais on dit qu'un étudiant a une note d'examen. Pour synthétiser, on peut dire que la classe *Etudiant* hérite de la classe *Personne*, et elle encapsule (elle a comme attribut) l'attribut $Note_Examen$. On a vu qu'une sous-classe hérite d'une super-classe les attributs et les méthodes d'une super-classe, et non pas les constructeurs. En outre, l'héritage est transitif, c'est-à-dire que si on a une super-classe qui hérite une super-super-classe, alors on récupère au niveau d'une sous-classe l'ensemble des méthodes et des attributs de ces deux super-classes. Plus concrètement, si on considère la super-classe *Enseignant* qui hérite la super-super-classe *Personne*. On va récupérer au niveau de la classe *EnseignantChercheur* qui hérite la super-classe *Enseignant* les attributs et les méthodes des classes *Enseignant*, et *Personne*. Puisque la classe *Enseignant* hérite de la classe *Personne*, alors elle récupère les attributs et les mé-

thodes de la classe *Personne*. Alors dans *EnseignantChercheur* on aura bien les attributs et les méthodes de la classe *Enseignant*, et par transitivité on récupère également au travers de la classe *Enseignant* les attributs et les méthodes de la classe *Personne*. En d'autres termes, on a la classe *Personne* dont hérite la classe *Enseignant*, et en disant qu'un enseignant chercheur est une sorte d'enseignant, alors la classe *EnseignantChercheur* hérite de la classe *Enseignant*. Évidemment, dans la classe *EnseignantChercheur*, on aura également les attributs *Nom*, *Prénom*, *Age* ainsi que la méthode $affiche r$ qui sont hérités de la classe *Enseignant*. En outre, avec un enrichissement progressif pour les différentes classes, on peut trouver un schéma arborescent. Une structure d'arbre, où on a toutes les dépendances (toutes les relations $est - un$) entre les classes. Ainsi que les relations d'héritage d'attributs, de méthodes et de types qui va donner ce que l'on appelle une hiérarchie de classes. Une hiérarchie avec les classes les plus générales (les plus abstraites) en haut et les classes les plus spécialisées (les plus enrichies) en bas.

3.2 Héritage en *Java*

Après avoir présenté l'aspect théorique du concept héritage, on va présenter dans la section suivante l'aspect pratique pour hériter une sous-classe d'une super-classe. Partant d'un exemple, où on souhaite définir la classe *Etudiant* avec les attributs *Section* et *Groupe* comme une sous-classe de la classe *Personne*. La classe *Personne* a les attributs *Nom*, *Prénom* et *Adresse*. On aura le diagramme d'héritage suivant (voir figure 3.3). Alors on a, la super-classe *Personne* avec les attributs *Nom*, *Prénom* et *Adresse*. L'instance de la sous-classe *Etudiant* est une instance de la classe *Personne* possède les attributs *Section* et *Groupe* en plus les attributs *Nom*, *Prénom* et *Adresse* hérités de la super-classe *Personne*. Concrètement en *Java*, la syntaxe pour hériter une sous-classe d'une super-classe consiste simplement de rajouter le mot clé ***extends*** suivi par l'identificateur de la super-classe directement

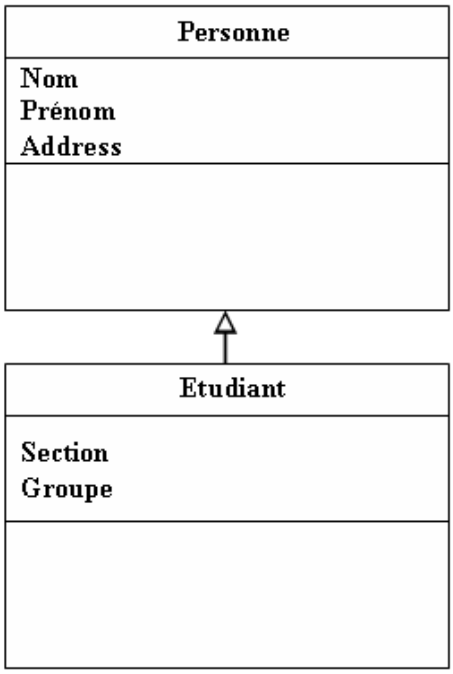

FIGURE 3.3 – La classe étudiant étendre la classe personne

après l'entête de la définition de la sous-classe. Ensuite, la déclaration des attributs et des méthodes spécifiques dans la sous-classe est absolument similaire à la définition usuelle d'une classe. Par exemple, pour la classe *Etudiant* qui hérite la classe *Personne*, on adopte la tournure ***class Etudiant extends Personne*** en rajoutant la définition de la classe *Etudiant* pour définir les deux attributs *Section* et *Groupe* spécifiques à la classe *Etudiant*. La seule différence par rapport à la définition usuelle d'une classe, pour qu'elle hérite une autre classe, est simplement la portion de code ***extends Personne*** (voir figur 3.4).

3.3 Accès Protégé

Cette section à pour objectif d'adresser une autre autorisation d'accès aux membres (des attributs ou des méthodes) d'instances des classes, ce que l'on appel en termes de jargon le droit d'accès protégé. On a déjà vu les autorisations d'accès où un membre peut être :

```
class Personne{
  String Nom;
  String Prénom;
  String Adresse;
}//-------------------------------------------------------------

class Etudiant extends Personne{
  String Section;
  String Groupe;
}
```

FIGURE 3.4 – Lien d'héritage

- utilisable partout sans aucune restriction aussi bien à l'intérieur qu'à l'extérieur de la classe (le droit d'accès publique),
- utilisable uniquement à l'intérieur de la classe (le droit d'accès privé),
- utilisable pour toutes les classes de même paquetage, et également à l'intérieur de la classe (le droit d'accès par défaut, où aucun modificateur de droit d'accès n'est définit explicitement).

Cependant, on à un autre droit d'accès très important en tenant compte la relation d'héritage pour favoriser la notion de réutilisation de code pour une hiérarchie de classes. Supposer qu'on a une sous-classe qui hérite une super-classe. Évidemment, dans une méthode d'une sous-classe, il n'est pas toujours possible d'accéder aux membres d'une super-classe. Supposer qu'on a par exemple, dans une super-classe un attribut privé, alors au moyen de la relation d'héritage, une sous-classe dispose systématiquement cet attribut hérité de la super-classe. En fait, le droit d'accès privé autorise l'utilisation des membres qu'à l'intérieur de la classe dans laquelle ils sont définis. Dans ce cas, l'attribut privé ne soit utilisable que dans les méthodes et les constructeurs de la super-classe. Par conséquence, l'accès à cette attribut privé de puis une méthode de la sous-classe est non autorisé. On se trouve alors dans

une situation un peu paradoxale, puisque une sous classe qui hérite une super-classe dispose alors d'un attribut, mais elle n'a pas le droit de le manipuler. Dans la sous classe on ne peut pas accéder directement à cet attribut privé, malgré qu'elle dispose directement de tous les attributs qui sont directement définis, et également ceux hérités de la super-classe au moyen de lien d'héritage. Pour ce faire, l'approche OO objet offre un autre droit d'accès qui est le droit d'accès protégé. Dans ce cas, lorsqu'un membre (un attribut ou une méthode) est définit comme protégé dans une classe donnée, il devient directement accessible dans toutes les classes de sa descendance (toutes les sous classes). En java, ce droit d'accès se désigne par le modificateur *protected*. L'accès protégé en plus de la visibilité interne, et la visibilité à toutes les sous classes, il assure encore plus la visibilité pour toutes les classes du même paquetage. Il s'agit alors d'une extension du droit d'accès par défaut. Alors, un membre protégé est directement accessible dans toutes les sous classes de cette classe de même paquetage ou de paquetages différents. Ainsi, il assure la visibilité pour toutes les classes de même paquetage même s'il n'existe aucun lien d'héritage. Par exemple, si on a deux classes dans un même paquetage, où il n'y a pas de lien d'héritage. Dans une méthode de l'une de ces classes, on peut accéder directement à l'attribut protégé de l'autre classe sans aucun garde-fou. Par contre, si on a une troisième classe de paquetage différent, elle ne pourrait pas accéder aux membres protégés des autres classes tant qu'il n'y a pas de lien d'héritage. Évidemment, le droit d'accès protégé case la notion d'encapsulation de données, puisque il permet surtout à tous programmeurs utilisateur d'étendre une super classe. En d'autres termes, tout programmeur concepteur d'une classe qui veut hérite une super classe est certainement autorisé d'accéder aux membres protégés de la super classe comme s'ils soient publiques. Ce genre d'autorisation aboutissement une forte dépendance entre les classes, puisque le programmeur concepteur d'une super classe n'a plus la liberté de modification et l'adaptation sur des membres protégés, pour des raisons

de maintenabilité, sans que les programmeurs concepteurs des sous classes ne soient impactés. Le droit d'accès par défaut est alors un peu plus restrictif que le droit protégé, puisque il n'autorise pas l'accès aux membres pour les classes de différents paquetages. Malgré qu'il soit un peu plus restrictif que le droit protégé, il reste inefficace et un peut dangereux pour une bonne encapsulation, puisqu'il favorise des accès non contrôlés depuis toutes autres classes du même paquetage. Partant un exemple, où on considère que la classe *Personne* est définie dans un paquetage particulier (voir figure 3.5). La sous-classe *Etudiant* qui hérite la super-classe *Personne* accède directement aux membres protégés de la classe *Personne*, que ce soit le paquetage dans laquelle est définit. Tandis que, si on a dans un autre paquetage une classe où il n'y a pas un lien d'héritage avec la classe *Personne*, alors l'accès aux membres protégés dans ce cas est impossible. Par exemple, dans la méthode *main* de la classe *GestionArtificielle* qui fasse intervenir un certain nombre personnes de la hiérarchie, on peut définir une variable de type *Etudiant*. Alors l'accès à l'attribut *Nom* par exemple, depuis la méthode *main*, sur un objet de la classe *Etudiant* est impossible, puisque les classes *GestionArtificielle* et *Etudiant* n'ont pas de lien d'héritage entre eux. Alors, on a dans la même situation où l'attribut protégé *Nom* était définit en privé.

3.4 Constructeur d'Héritage

Après avoir présenté le concept d'héritage, dans cette section, on va aborder l'influence de ce concept sur la construction et l'initialisation des objets. Comme on a déjà vu, l'instanciation d'une classe n'est pas autre chose que l'initialisation des attributs de cette classe. Alors, lors de la création d'une instance par le biais de passage de paramètres à un constructeur de la classe, on est initialiser tous ses attributs que ce soit par des valeurs par défauts ou bien par des valeurs bien appropriées. Évidemment, si une classe est une sous classe qui hérite une super-classe, alors elle

```
 package p1;

class Personne{

 protected String Nom;
 protected String Prénom;
}
```

```
 package p2;

class Etudiant extends Personne{
  String Section;
  String Groupe;
}
```

```
package p3;

class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E = new Etudiant();
 E.Nom = "Lamouri"; // erreur d'accès
 }
}
```

FIGURE 3.5 – Accès Protégé

reçoit l'ensemble des attributs de la super classe. Les constructeurs d'une sous classe doivent initialiser tous les attributs de la sous classe, et y compris les attributs hérités de la super-classe. Absolument, il ne soit pas nécessairement le concepteur de la sous classe qui lui-même initialise les attributs de la super-classe. En outre, il peut être même pas incapable de les accéder s'ils sont définit en privés dans la super-classe. Dans ce cas, le concepteur de la sous-classe, il doit faire appel au constructeur de

la super-classe pour initialiser les attributs de la super-classe. L'initialisation des attributs hérités nécessite des appelles aux constructeurs des super-classes dans des constructeurs des sous-classes. En java, pour pouvoir utliser le constructeur de la super-classe dans la définition du constructeur de la sous-classe, on a la méthode particulière *super*. L'appel de la méthode *super* doit être la première instruction dans la définition du corps du constructeur selon la syntaxe indiqué par la figure 3.6. Dans le constructeur d'une sous-classe, tout au début de son corps, on fait ap-

```
IdentificateurSousClasse(paramètre1, paramètre2, ...){
 super(arg1, arg2, ...); // apelle d'un constructeur de la
   super-classe
 // initialisation des attributs de SousClasse
}
```

FIGURE 3.6 – La syntaxe d'apelle de constructeur d'une super-classe

pel à un des constructeurs d'une super-classe en spécifiant la liste des arguments nécessaires pour son invocation. Ensuite, après la première ligne on fait initialiser les attributs défini dans la sous classe. Cependant, si pour une super-classe on a un constructeur par défaut, il devient facultatif de faire appeler explicitement le constructeur d'une super-classe, puisque le constructeur par défaut de la super-classe est systématiquement invoquer par le compilateur. Évidemment, si une sous classe n'a pas de constructeur par défaut, alors il faut absolument écrire explicitement l'appel à un des constructeurs de la super-classe. En d'autres termes, dans la sous-classe elle-même, on doit avoir un constructeur explicite, pour pouvoir appeler un des constructeurs d'une super-classe. Pourtant d'un exemple, où on considère dans la classe *Personne* exactement un sel constructeur explicite dont les paramètres sont *Nom* et *Prénom* pour pouvoir initialiser les deux attributs *Nom* et *Prénom* des instances de la classe *Personne* (voir figure 3.7). Dans la classe *Etudiant* qui

```
class Personne{

  String Nom;
  String Prénom;

  public Personne(String nom, String prénom){
    Nom=nom;
    Prénom=prénom;
  }

  void afficher(){
    System.out.println("Le nom de la personne est : " + Nom +
   ", et son prénom est :"+ Prénom);
  }
}
```

FIGURE 3.7 – La Super-Classe *Personne*

hérite la classe *Personne*, on a un constructeur qui doit avoir des paramètres pour pouvoir initialiser les *Nom* et *Prénom* de sa super-classe *Personne*. Puisque, tout étudiant est une personne, alors il possède par le lien d'héritage les attributs *Nom* et *Prénom* de la super-classe *Personne*. En conséquence, le constructeur de la sous-classe *Etudiant* en plus des paramètres *Section* et *groupe* pour pouvoir initialiser ses propres attributs directes *Section* et *Groupe*, il doit avoir aussi des autres paramètres *Nom* et *Prénom* pour falloir invoquer le constructeur de la super-classe afin d'initialiser convenablement ses autres attributs *Nom* et *Prénom* hérités. Évidemment, le constructeur de la sous classe *Etudiant* fait appeler premièrement le constructeur de la super-classe *Personne* en passant les deux paramètres *nom* et *prénom*. Cette appel au constructeur de la super-classe *Personne* initialisera les attributs propres *Nom* et *Prénom* de la super-classe avec les valeurs des paramètres *nom* et *prénom* du constructeur de la sous-classe *Etudiant* (voir figure 3.8). Par contre, si on a cette fois-ci dans la super-classe *Personne* un constructeur par défaut qui ne prend pas de paramètre, alors il devient non nécessaire de l'appeler explicitement

dans le constructeur de la sous-classe *Etudiant* (voir figure 3.9). Ce constructeur

```
class Etudiant extends Personne {

  String Section;
  String Groupe;

  Etudiant (String nom, String prénom, String section,  String
    groupe) {
   super(nom, prénom);
   Section = section;
   Groupe = groupe;
  }
}
```

FIGURE 3.8 – L'appelle du constructeur de la super-classe

```
class Etudiant extends Personne {

 Etudiant (String nom, String prénom) {
  super();
  Nom = nom;
  Prénom = prénom;
  }
}
```

```
class Etudiant extends Personne {

  Etudiant (String nom, String prénom) {

   Nom = nom;
   Prénom = prénom;
  }
}
```

FIGURE 3.9 – L'appelle du constructeur par défaut est facultatif

par défaut sert à initialiser les attributs de la super-classe *Personne* avec des valeurs par défaut. Dans la sous classe *Etudiant*, On a définit un constructeur usuel qui prend les deux paramètres *nom* et *prénom* pour initialiser les attributs *Nom* et *Prénom*. Malgré qu'il n'y a pas d'appel explicite au constructeur de la super-classe *Personne*, mais en fait, il sera invoqué systématiquement par le compilateur une fois que le constructeur usuel de la sous classe *Etudiant* est exécuté. Cependant, il n'est pas forcément nécessaire d'avoir des attributs supplémentaires dans des sous-classes pour définir leurs constructeurs en utilisant des constructeurs des super-classes. En d'autres termes, un constructeur d'une sous-classe, même si il n'y a pas des attributs spécifiques directement définit dans sa sous-classe, il peut faire des appels spécifiques à des constructeurs d'une super-classe en fournissant bien sur les arguments nécessaire à l'invocation des constructeurs d'une super-classe. Par exemple, si on suppose qu'on a une sous classe *Agent* hérite de la super-classe *Personne*. Un agent est une personne un peu particulier qui ne possède pas des attributs supplémentaires. Dans ce cas, dans la sous classe *Agent*, on ne défini aucun attribut supplémentaire. Mais on doit vouloir définir explicitement dans la sous-classe *Agent* un constructeur spécifique pour initialiser les attributs *Nom* et *Prénom* hérité de la super classe *Personne* en faisant appel au constructeur de la super-classe. Alors au moyen du constructeur la sous-classe *Agent*, on permet la création des instances en l'obligeant l'initialisation des attributs *Nom* et *Prénom* sans tenir compte d'autres attributs additionnels. Dans la sous classe *Agent*, on a rien d'autre que son constructeur qui exige le passage de valeurs en argument afin de pouvoir invoqué le constructeur de la super-classe *Personne*. Dans la sous classe *Agent*, on a rien de plus pour initialiser les attributs *Nom* et *Prénom* (s'il n'y a pas bien sûr de manipulateur ou des setters pour ces attributs). Par contre, si on a dans la super-classe *Personne* des manipulateur pour ces attributs (*setNom* et (*setPrénom*) qui reçoivent en paramètres des valeurs pour initiliser les attributs *Nom* et *Prénom*. Parfois, même

si on a des manipulateurs dans une super-classe pour initialiser ses attributs, on doit définir explicitement des constructeurs afin de pouvoir créer des instances pour des sous classes. Alors, il faut noter en toute rigueur que, lorsqu'on a des constructeurs explicites dans une super-classe, on doit avoir définir au moins un constructeur pour pouvoir appeler un parmi eux au moyen de la méthode particulière *super* en fournissant les bons arguments. Ainsi, l'utilisation de la méthode *super* doit être toute la première instruction, et alors on n'a jamais des utilisations multiples à cette méthode dans un même constructeur. En outre, aucune d'autres méthodes que les constructeurs des sous-classes ne sont autoriser d'utiliser la méthode particulière *super*. On peut noter encore, que l'utilisation de la méthode *super* est facultatif lorsque une super-classe a un constructeur par défaut (que ce soit explicite ou implicite), puisque le compilateur génère systématiquement une appelle au constructeur par défaut de la super-classe. Ainsi, pour une hiérarchie de classes, l'ordre des appelles des constructeurs est nécessairement monotone, c'est-à-dire depuis une sous classe à une super-classe. Partant d'un exemple, où on considère une sous classe *EnseignantChercheur* qui a ses propres attributs et ses propres méthodes, et elle hérite d'une autre sous classe *Enseignant* d'autres attributs et autres méthodes. La sous classe *Enseignant* possède certains attributs et certains méthodes, et elle hérite elle-même de la classe *Personne* d'autres attributs et d'autres méthodes (voir figure 3.10). Si on défini une variable *enCh* de type *EnseignantChercheur* en l'initialisant avec une instance de la classe *EnseignantChercheur*, alors l'appelle du constructeur de la classe *EnseignantChercheur* provoque une autre appelle de l'un des constructeurs de la classe *Enseignant*. Le constructeur de la classe *Enseignant* a son tour fait appel à un des constructeurs de la super-classe *Personne*. Cette appelle du des constructeurs de la super-classe *Personne* initialises alors les attributs impliqués par le constructeur. Une fois initialisés, et l'exécution de ce constructeur (de la super-classe *Personne*) est terminé, les autres instructions impliquées par le constructeur

```
class Personne {
 String Nom;
 String Prénom;
 String Adresse;
 Personne (String nom, String prénom){
 Nom = nom;
 Prénom = prénom;
 }
}//_____________________________________________________________
class Enseignant extends Personne {
 String Grade;
 String Spécialité;
 Enseignant (String nom, String prénom, String grade) {
   super(nom, prénom);
   Grade = grade;
  }
}//_____________________________________________________________
class EnseignantChercheur extends Enseignant {
 String Profile;
 EnseignantChercheur (String nom, String prénom, String grade,
    String profile) {
   super(nom, prénom, grade);
   Profile = profile;
  }
}
```

FIGURE 3.10 – Hiérarchie de classes : Appelles des constructeurs

la classe *Enseignant* seront exécuter, et par conséquence les attributs spécifiés sont également initialisés. En suite, après l'exécution des instructions du constructeur de la classe *Enseignant*, selles du constructeur de la classe *EnseignantChercheur* vont être exécutées, et en fait les attributs impliqués vont être initialisés. En d'autres termes, en adoptant un mécanisme des appelles récursives, une construction d'une sous-sous-sous-classe fait appel tout d'abord au constructeur de sa super-classe directe (la sous-sous-classe) dont on dérive avant l'exécution des instructions d'initialisation des attributs de sa classe. De la même manière, la construction de la sous-sous-classe fait appel au constructeur de sa super-classe directe (la sous-classe),

ainsi de suite pour tous les constructeurs en passant par touts ses super-classes intermédiaires jusqu'à la super-classe principale (la classe mère de toutes l'hiérarchie de classes). En suite, après avoirs invoqué touts les constructeur de l'hiérarchie, on adopte le chemin inverse pour initialises tous les attributs impliqués par les instructions des constructeurs de ces classe, et alors on termine par la partie spécifique au constructeur de la sous-sous-sous-classe dont on est en train de créer son instance.

3.5 Polymorphisme

En OO, le polymorphisme est la capacité d'un variable de s'adapter systématiquement aux différents types de données qui lui sont attribuées. Il s'agit d'un concept très importante, puisqu'il permet d'unifier différents types de données en se basant sur la notion de type générique. Supposer qu'on souhaite manipuler des données pour une hiérarchie de personnes de natures diverses (voir figure 3.11). Par exemple, si on suppose qu'on a dans un tableau un certaines instances qui représentent des personnes de natures diverses, et on veut afficher, au moyen d'une boucle, tous les données caractéristiques de ces personnes (voir figure 3.12). Alors dans le tableau qui est de type *Personne*, on peut avoir des instances de différents types. Par exemple, la première entrée du tableau peut avoir une instance de la classe *Administrateur*, la seconde une instance de la classe *Etudiant*, la troisième une instance de la classe *Enseignant*. Il s'agit alors des données de types variés, malgré que le langage *Java* est fortement typé. Évidemment, les langages fortement typés exige que le type de la partie à gauche d'une affectation doit être le même que la partie à droite. En effet, cette contrainte est parfaitement satisfaite, puisque les instances de type *Enseignant*, *Administrateur* ou *Etudiant* sont certainement des instances des sous-classes de la classe *Personne*, car le tableau est défini avec le même type *Personne*. Lorsqu'on affecte à une variable de type super-classe une référence vers une instance de type sous-classe, on n'est pas en contradiction avec la

```
class Personne {
 String Nom;
 String Prénom;
 String Adresse;
 // ...
 void afficher (){
 System .out. println ("Le nom de la personne est : " + Nom +
   ", et son prénom est : " + Prénom);
 }
}//-------------------------------------------------------------
class Etudiant extends Personne {
 String Section;
 String Groupe;
 // ...
 void afficher (){
 System .out. println ("Le groupe de l'etudiant " + Nom + " "
   + Prénom + "est : " + Groupe);
 }
}//-------------------------------------------------------------
class Enseignant extends Personne {
 String Grade;
 // ...
 void afficher (){
 System .out. println ("Le grade de l'enseignant " + Nom + " "
    + Prénom + "est : " + Grade);
 }
}//-------------------------------------------------------------
class Administrateur extends Personne {
 String Service;
 // ...
 void afficher (){
 System .out. println ("Le service de l'administrateur " + Nom
    + " " + Prénom + "est : " + Service);
 }
}
```

FIGURE 3.11 – Hiérarchie de personnes

contrainte de types (le type de la partie à droite de l'affectation doit être le même de selle à gauche), puisque dans une hiérarchie de classes le type est systématiquement hérité.Alors, une instance de la sous-classe *Enseignant* hérite du type de sa super-

```
class GestionArtificielle {

 public static void main (String[] args) {
 Personne [] P;
 P = new Personne [5];
 P [0] = new Etudiant ("Lamouri", "Amine");
 P [1] = new Enseignant ("Mostfielle", "AEK");
  // ...
 for (int i = 0; i<2; i++) P[i].afficher();
 }
}
```

FIGURE 3.12 – Manipulation d'objets de natures diverses

classe *Personne*. En outre, dans le tableau, on peut avoir également des instances de la classe *EnseignantChercheur*, puisque l'héritage est transitif. En fait, un objet de type sous-classe hérite par transitivité de tous les types de ses ascendances, et il peut avoir alors plusieurs types. Subséquemment, une instance de la sous-classe *EnseignantChercheur* hérite du type de sa super-classe *Personne*. C'est pour cela, qu'on peut stocker dans un tableau de type *Personne* des personnes (des instances) de natures diverses. Ce genre de mécanismes a pour avantage d'unifier la manipulation des instances de type divers (voir figure 3.13).

3.6 La résolution des liens

En effet, on a déjà vu dans le chapitre précédent la manipulation de données bien spécifiques. Par exemple, lorsque la sous-classe *Etudiant* de la classe *Personne* a un affichage particulier, il est tout à fait possible qu'elle redéfinisse a nouveau une méthode déjà défini dans la super-classe *Personne* en impliquant en plus les donnée *Section* et *Groupe*, alors que les autres sous-classes gèrent l'affichage différemment où on n'implique pas les attributs *Section* et *Groupe*. Si on a un objet de la classe

```
class Personne {
 String Nom;
 String Prénom;
 String Adresse;
 // ...
 void afficher (){
 System .out. println ("Le nom de la personne est : " + Nom +
   ", et son prénom est : " + Prénom);
 }
}//-------------------------------------------------------------

class Enseignant extends Personne {
 String Grade;
 // ...
 void afficher (){
 System .out. println ("Le grade de l'enseignant " + Nom + " "
    + Prénom + "est : " + Grade);
 }
}//-------------------------------------------------------------

class EnseignantChercheur extends Enseignant {
 String Profile;
 // ...
 void afficher (){
 System .out. println ("Le grade de l'enseignant " + Nom + " "
    + Prénom + "est : " + Grade);
 }
}
```

FIGURE 3.13 – Transitivité d'héritage de types des spers-classes

Etudiant pour une variable (P) de la classe *Personne*, alors lors de l'invocation de la méthode $affiche r$ sur ce variable, on aura un conflit d'invocation entre la méthode $affiche r$ de la sous-classe *Etudiant*, et celle de la super-classe *Personne*. Si exactement le même problème rencontré précédemment, où on a des objets de la classe *Etudiant* dans un tableau de type *Personne* (dont les données sont des objets de la classe *Personne*). En conséquence, Il devient très importants d'avoir connaître si cette manipulation soit au nivaux des sous-classes, ou bien aux niveaux

de la super-classe. Pour ce faire, deux points de vues peuvent être possibles :

- le type de la variable : dans ce cas, on adopte le type du variable sur lequel l'instance est manipulée pour déterminer les membres à impliqués. Pour l'exemple indiqué par la figure 3.14, le type du variable, sur lequel la méthode $affiche r$ est invoquée, est $Personne$. Si on adopte le type de variable comme un cri-

```
class GestionArtificielle {

 public static void main (String[] args) {
 Personne P;
 P = new Etudiant ("Lamouri", "Amine");
 }
}
```

FIGURE 3.14 – La résolution statique des liens

tère de décision, alors la méthode de la classe $Personne$ qui sera invoquée. En conséquence, l'affichage sera effectué au niveau de la super-classe $Personne$ en impliquant les attributs Nom et $Prénom$. Ce point de vue de résolution de lien entre une méthode et une variable pour déterminer la méthode à invoquée, est on l'appel en termes de jargon la résolution statique des liens. Il s'agit d'une méthode de résolution qui ne nécessite pas l'exécution du programme pour déterminer la méthode ($affiche r$) à appliquer, mais plutôt à la simple lecture du programme au moment de la compilation, on peut connaitre le type de la variable (P est de type $Personne$), et en suite, on désigne la méthode $affiche r$ de la classe $Personne$. Alors, pour la résolution statique des liens, le type apparent du variable est le déterminant pour décider les membres des instances. Telle résolution de liens est supportée pour certains langages, mais elle n'est pas tenue par le langage $Java$.

- le type effectif (le type de donnée) : dans ce cas, on adopte le type de l'objet

effectivement attribué au variable pour déterminer les membres à impliqués. Certainement, la variable P est de type $Personne$, et elle contient une référence vers un objet de la classe *Etudiant* (voir figure 3.14). Alors, cette dernière est le type effectif contenu dans la variable. Dans ce cas, si on adopte ce type effectif comme un critère de décision, alors la méthode de la classe *Etudiant* qui sera invoquée. En conséquence, l'affichage sera effectué au niveau de la sous-classe *Etudiant*.Alors, au lieu d'afficher les valeurs des attributs Nom et $Prénom$, on affiche selles des attributs *Section* et *Groupe*. Évidemment, le type effectif de données contenues réellement dans le variable, plutôt que le type apparent de variable qui détermine la méthode à appliquer. Ce point de vue de résolution de lien adopté pour déterminer la méthode à invoquée, ce qu'on appel en termes de jargon la résolution dynamique des liens. Contrairement à la résolution statique, la résolution dynamique décide la méthode à appliquer qu'au moment de l'exécution du programme, puisqu'on va pouvoir déterminer le contenu effectif des variables qu'après l'affectation concrète des données (objets des classes). Il s'agit alors d'une résolution dynamique, parce que le type est dans ce cas n'est pas permanent, mais il peut être varié en cours d'exécution tout dépend aux types de données affecter. La résolution dynamique des liens est supportée généralement par tous les langages, et aussi bien par le langage *Java*.

En programmation, deux points de vue sont possibles pour la résolution des liens. La résolution statique et la résolution dynamique. Certains langages supportent la possibilité de choisir le point de vue souhaité. Ils offrent aux programmeurs de choisir explicitement la résolution (statique ou bien dynamique) à adopter. Alors que, en *Java*, cette possibilité de choix libre n'est pas autorisée, mais par contre on doit adopte systématiquement la résolution dynamique de liens.

3.7 Accès aux membres d'une super-classe

Les membres définis dans des super-classes ne remplissent pas toujours les exigences d'une sous-classe. Par exemple pour une hiérarchie de personnes, on ne doit pas nécessairement une même façon d'affichage des données pour toutes les personnes. Alors lorsqu'on affiche les données d'une personne, on n'implique pas uniquement toujours son nom et son prénom. Pour être face à ce genre de problème, les concepts de redéfinition ou de masquage sont adoptés par l'approche OO. Si on souhaite cette fois-ci que la méthode $affficher$ ne soit pas satisfaisante pour toutes les classes. On peut considère que les étudiants sont des personnes un peu plus particuliers, où on veut impliquer en plus leur section et leur groupe, au lieu de leur nom et leur prénom uniquement. Dans ce cas, on aperçoit une situation un peut particulière où on considère deux implémentations possibles pour la même méthode $affficher$. Une implémentation pour les étudiants en tant que personnes générique, et une autre pour les étudiants bien particuliers. En d'autres termes, une implémentation pour les personnes qui ne sont pas des étudiants (une personne générique où on affiche uniquement son nom et son prénom), et une autre implémentation pour les étudiants où on affiche cette fois-ci leur section et leur groupe. Pour ce faire, il suffit de définir dans la classe *Etudiant*, une méthode $affficher$ bien spécialisée sans revoir complètement l'hiérarchie de classes. On garde dans la super-classe *Personne* la méthode $affficher$ générique qui est appropriée pour la majorité des sous-classes. En outre, dans la sous-classe *Etudiant*, on offre une nouvelle définition bien spécialisée à la méthode $affficher$ afin de mieux rependre aux besoins de la sous-classe *Etudiant*. Pour une instance de la classe *Etudiant*, la nouvelle méthode $affficher$ spécialisée a la priorité de précédence par rapport à la méthode $affficher$ générique. Alors, au moyen de lien d'héritage, la méthode $affficher$ générique est applicable systématiquement de redéfinition plus spécifique dans les sous-classes.

Par exemple, si on a une instance de la classe *Enseignant*, et si on applique la méthode $affiche r$ pour cette instance, alors la méthode $affiche r$ générique héritée de la classe *Personne* qui va être invoquée.Par contre, si on a une instance de la classe *Etudiant*, et s'on lui applique la méthode $affiche r$, alors la méthode spécifique de la sous-classe *Etudiant* qui cette fois-ci va être appliquée, puisqu'il existe, dans ce cas, une redéfinition de la méthode $affiche r$ dans la sous-classe *Etudiant*. Alors, dans une hiérarchie de classes, on peut avoir plusieurs méthodes de même entête (de même signature) définissent sur plusieurs niveaux de l'hiérarchie. Il s'agit de ce qu'on appelle la redéfinition (overriding), où une méthode est redéfinie dans une sous-classe pour des raisons de spécialisation.

Similairement aux méthodes, on peut avoir plusieurs attributs avec le même identificateur sur plusieurs niveaux d'une hiérarchie. Par exemple, s'on a dans une super-classe *Enseignant* un attribut donné *Dpartement*, pour désigner le département dans lequel l'enseignant est recruté. En outre, il est possible d'avoir déclaré à nouveau un attribut *Dpartement* dans une sous-classe par exemple *EnseignantChercheur*, pour désigner cette fois-ci le département où se trouve le laboratoire dans lequel l'enseignant est rattaché. On est dans une situation, où un attribut de même identificateur est définit sur deux niveaux de l'hiérarchie. Alors, une instance de type *EnseignantChercheur* disposera alors de deux attributs nommés *Dpartement*, l'un de ces attributs qui est définit directement dans la classe *EnseignantChercheur*, et l'autre qui est hérité de la super-classe *Enseignant*. Si dans une méthode de la classe *EnseignantChercheur*, on utilise l'attribut *Dpartement*, alors l'attribut *Dpartement* de la classe *EnseignantChercheur* qui soit utilisé, et non pas celui hérité de la super-classe *Enseignant*. Dans ce cas, on dit que l'attribut qui est définit dans la sous classe *EnseignantChercheur* masque celui, qui porte le même nom, qui est défini dans la super-classe *Enseignant*. Il s'agit de ce qu'on appel le *masquage* (*shadowing*) des attributs. Les attributs dans une hiérarchie de classes sont alors

une source d'ambiguïté pénible.

Cependant, contrairement masquage des attributs, la redéfinition de méthodes est très pratique et beaucoup plus courant, puisque il a pour avantage de bien spécialiser des méthodes aux besoins très spécifiques d'une sous-classe. Alors, dans une hiérarchie de classes, La redéfinition de méthodes consiste à définir a nouveau dans des sous classes une méthode, avec le même entête, déjà définit dans une super-classe pour satisfaire des besoins bien spécialisées. En termes de jargon, on dit qu'une méthode héritée dont peut bénéficier d'autres éventuelles sous-classes d'une super-classe qui ne la redéfinit pas spécifiquement est une méthode par défaut (une méthode générale), et une méthode de sous classe qui redéfinit une méthode héritée de la superclasse est une méthode spécialisée qui s'adapte spécifiquement aux besoins de cette sous-classe. La méthode spécialisée a toujours la priorité de précédence sur une méthode par défaut. Par exemple, lorsqu'on adopte la méthode $afficher$ sur un objet de la classe *Etudiant*, alors la méthode $afficher$ spécialisée pour la sous classe *Etudiant* qui va être invoquée. En effet, un objet de la sous classe *Etudiant* dispose de deux méthodes $afficher$, une méthode spécifique propre, et une autre méthode générale hérité de plus haut. Parfois, dans une sous-classe, il est devient très utile d'utiliser une méthode générale pour la définition d'une méthode spécialisé. Par exemple, si on souhaite avoir dans la sous classe *Etudiant* une bonne manière de réutilisation de code. Alors, on peut utiliser la méthode générale dans la définition de la méthode spécifique de la sous classe *Etudiant*. Pour une personne qui n'est pas un étudiant, on a gardé la méthode générale pour afficher uniquement le nom et le prénom de cette personne. Par contre pour une personne qui est un étudiant, on a défini une méthode spécialisée pour afficher cette fois-ci le groupe en plus le nom et le prénom de cet étudiant en faisant appel à la méthode générale. Alors, on a utilisé la méthode générale depuis la méthode spécialisée pour ne pas dupliquer le code. Il est plus intelligent d'appeler une méthode générale en favorisant

parfaitement la réutilisation des lignes de code correspondantes. En *Java*, pour accéder aux attributs masqués, et aux méthodes redéfinies d'une super-classe, on doit avoir adopter le mot réservé *super* selon une syntaxe particulière, où en impliquant l'identificateur de l'attribut (*super.identificateurAttribut*) ou sel de la méthode (*super.identificateurMthode*) en question. Partant un exemple, où on souhaite utiliser la méthode *afficher* héritée de la classe *Personne* dans la méthode *afficher* de la sous-classe *Etudiant*. Évidemment, il est tout à fait possible d'enchaîner des actions plus spécifiques dans la sous classe. En outre, l'appel à une méthode générale est très utile, puisqu'elle permet de réutiliser dans une méthode spécialisée toutes les instructions de la méthode générale (voir figure 3.15). En *Java*, si une méthode géné-

```
class Personne {

 String Nom;
 String Prénom;
 String Adresse;
 // ...
 void afficher (){
 System.out.print (Nom + " " + Prénom);
 }
}//_____________________________________________________

class Etudiant extends Personne {

 String Section;
 String Groupe;
 // ...
 void afficher (){
 System.out.print ("Le groupe de l'etudiant ");
 super.afficher ();
 System.out.println ("est : " + Groupe);
 }
}
```

FIGURE 3.15 – Apelles de méthodes générales

rale ne soit pas redéfinie à un niveau intermédiaire, alors *super* permettre d'invoquer la méthode générale de la super-classe la plus proche. En revanche, il n'est pas autorisé, d'enchaîner plusieurs *super* pour y accéder à une super-classe. Par exemple, si on a une méthode $afficher$ dans la super-classe $Personne$, ainsi elle est redéfinie dans la sous classe $Enseignant$, et également elle est redéfinie à nouveau dans la classe $EnseignantChercheur$, alors la méthode $afficher$ de $EnseignantChercheur$ ne peut pas court-circuiter la méthode héritée de la classe $Enseignant$ en faisant un appel tel que $super.super.afficher$. Les concepteurs du langage $Java$ ont considéré que telle tournure engendre le danger qu'une instance de type $sous-sous-classe$ se comporte davantage comme une instance de type $super-super-classe$, que comme une instance de sa $super-classe$ immédiate. Il peut être perçu probablement comme un affaiblissement de la relation d'héritage (de la relation $est-un$). Ce genre de limitation n'est pas considéré pour tous les langages de programmation OO, certains langages support tout à fait ce genre de tournures (une sous-sous-classe fasse appel à une méthode de la super-super-classe directement).

Chapitre 4

Réutilisation, et Membres Statiques

Sommaire

Dans les chapitres précédents on a présenté les notions fondamentales de l'approche OO, plus précisément on a introduit la notion d'abstraction, d'encapsulation et la notion d'héritage. Ce chapitre a pour objectif d'adresser d'autres notions très intéressantes. Il introduit un très bon complément d'abstraction pour mieux spécifier des classes beaucoup plus génériques en exigeant la redéfinition des méthodes dans

des sous classes. En revanche, il présente le principe de restriction de réutilisation de codes, et également la manière de partages de données entres les objets d'une mêmes classe.

4.1 Exigences de réutilisation

Comme il est déjà présenté dans le chapitre précédent, le polymorphisme universel d'inclusion (et on parle aussi au polymorphisme d'héritage) offre aux objets d'une sous-classe la capacité de conserver leurs propriétés propres au lieu d'être réduites uniquement aux propriétés d'une super-classe. Particulièrement, des objets d'une sous-classe (qui sont substituables aux objets des super-classes) gardent leurs comportements propres. Ce qui permet de désigner au moment d'exécution du programme les méthodes impliquées selon la nature effectif des objets contenues. Par exemple, si on a une classe *Administrateur* qui est hérité de la classe générale *Personne*, alors une instance de la sous classe *Administrateur* est une instance de la super-classe *Personne*, puisque tout administrateur est une personne. Pareillement, pour les sous classes *Enseignant* et *Etudiant*, etc. Les instances des sous classes *Enseignants*, *Etudiants* et *Administrateurs*, même s'ils sont perçus en tant que des instances de la super-classe *Personnes*, ils peuvent se comporter si on les manipule en tant que chacun des instances propres. Alors, si on affiche une instance de la sous classe *Administrateur* en tant que une instance de classe *Personne* on peut impliquer en plus l'attribut *Service*. Pour un *Etudiant* on peut impliquer en plus *Note_ TD* et *Section*, etc. Si on a une instance de la classe *Personne* référencé par $p1$ qui hérite une autre instance de la même classe *Personne* référencé par $p2$, alors la méthode $afficher$ va s'adapter à la nature réelle de l'instance de $p1$. Par conséquence, elle va afficher l'instance de $p2$ (voir figure 4.1) :

- comme une instance de la classe *Administrateur* si l'instance de $p1$ est de la classe *Administrateur*,

```
class Personne{
 String Nom;
 String Prénom;
 void afficher(){
 System.out.println("Le nom de la personne est :" + Nom +
                 ", et son prénom est : "+ Prénom);
 }
}//-------------------------------------------------------------
class Etudiant extends Personne{
 String Section ;
 String Groupe ;
 void afficher(){
 System.out.println("La section de l'étudiant  " + Nom + ", "
                  + Prénom + " est : " + Section);
 }
}//-------------------------------------------------------------
class Enseignant extends Personne {
 String Grade ;
 void afficher(){
 System.out.println("Le grade de l'enseignant  " + Nom + ", "
                  + Prénom +" est : " + Grade);
 }
}//-------------------------------------------------------------
class Administrateur extends Personne {
 String Service ;
 void afficher(){
 System.out.println("Le service de l'étudiant  " + Nom + ", "
                   + Prénom + " est : " + Service);
 }
}
```

FIGURE 4.1 – Redéfinition de la méthode $affiche r$

- comme une instance de la classe *Enseignant* si l'instance de $p1$ est de la classe *Enseignant*,
- et comme une instance de la classe *Etudiant* si l'instance de $p1$ est de la classe *Etudiant*.

Évidemment, l'héritage et la résolution dynamique de liens permet d'avoir une classe générale par rapport à laquelle les traitements doivent s'adapter aux sous-classes spé-

cifiques. Concrètement, la résolution dynamique des liens sert à désigner au moment d'exécution la bonne méthode selon la nature effective de l'instance en question. Par exemple, si on considère un affichage des instances pour les classes *Etudiant* et *Administrateur* (voir figure 4.2). La méthode $affficher$ implique un affichage pour

```
class GestionArtificielle {
 public static void main(String[] args) {
 afficher(new Etudiant("Lamouri", "Amine"),
             new Administrateur("Beri", "AEK"));
 }
 void afficher (Personne P1, Personne P2){
  P1.afficher();
  P2.afficher();
 }
}
```

FIGURE 4.2 – Appelle de la méthode $affficher$

deux instances de la classe *Personne* reçu en arguments en affichant un message par le biais de la méthode $affficher$. Grace à la résolution dynamique des liens, lors de l'affichage d'un étudiant (le premier paramètre) alors la méthode $affficher$ de l'instance *Etudiant* (de la classe *Etudiant*) qui va dynamiquement invoquer, et non pas celle de la classe *Personne*. Le polymorphisme permet de bien spécifier à un niveau plus abstrait des programmes en favorisant des manipulations beaucoup plus abstrait. Cependant, à un niveau beaucoup plus élevé d'une hiérarchie, on ne peut pas définir une méthode pour toutes les sous-classes d'une même super-classe. Par exemple, il devient difficile voire impossible de définir une méthode pour traiter et réaliser l'affichage pour toutes les catégories de personnes (des étudiants, des enseignants, des administrateurs, etc.) de la classe générale *Personne*. On ne peut pas certainement le faire, malgré qu'on sache que toutes personnes possèdent des données à traiter et également les afficher. Pour, un étudiant on peut considérer le

nom, le prénom, la section et le groupe, etc. Similairement pour un enseignant et un administrateur. Qu'au niveau de ces personnes spécifiques, on peut quand même définir des méthodes spécifiques (comme la méthode $affiche r$). Ainsi, à tel niveau, on peut considérer l'adoption de deux méthode par exemple, l'une pour préparer les données sous une forme d'une chaine de caractères qui s'appelle *commeChain*, et l'autre (appeler $affiche r$) qui fait appelle cette méthode *commeChain* pour afficher les données préparées. On peut décider que toutes les instances de la classe $Personne$ disposent une méthode *commeChain*, malgré qu'on ne connaitre pas bien la définir dans un niveau plus abstrait. On outre, pour pousser le problème encore pour une nouvelle fois, on peut considérer que la méthode *commeChain* soit utilisée au niveau plus élevé (la classe $Personne$) dans une autre méthode par exemple $affiche r$. Pour réaliser un affichage sur une certaine instance de la classe $Personne$, on peut impliquer l'attribut Nom et l'attribut $Prénom$ pour cette instance, et également d'autres données beaucoup plus spécialisées. On peut ainsi utiliser la méthode *commeChain* même si on ne peut pas la définir à un niveau plus abstrait. Ce genre de méthodes doit être définie pour tous les instances concrets de la super-classe $Personne$ la plus abstraite. Pour ce faire, l'approche OO en $Java$ adopte ce qu'on appelle en termes de jargon de méthode abstraite.

4.1.1 Méthodes Abstraites

Si on revient à notre hiérarchie de personnes où on considère des étudiants, des enseignants, et des administrateurs. Si on essaye d'afficher les données des différents instances de la classe $Personne$, alors on ne peut pas afficher certaines données des instances de la classe $Personne$, malgré qu'on peut absolument afficher des instances pour les sous classes $Etudiant$, $Enseignant$, $Administrateur$. Dans chacune de ces sous-classes spécifiques on peut définir une méthode pour afficher les données de ces instances. En outre, pour le niveau le plus abstrait (la classe $Personne$) on doit

permettre l'affichage des données des instances de la classes *Personne*, mais malheureusement on ne sait pas commet le faire. Alors, on peut définir arbitrairement dans la classe *Personne* une méthode dédier pour afficher des données des instances génériques (un affichage générique est de ne rien afficher), pour permettre l'appelle à une méthode d'affichage sur une instance de la classe (voir figure 4.3). Cette solu-

```
class Personne{
 String Nom;
 String Prénom;

  void afficher(){  }
}//_____________________________________________________________
class Etudiant extends Personne{
  /...
}//_____________________________________________________________
class Enseignant extends Personne {
  /...
}//_____________________________________________________________
class Administrateur extends Personne {
  /...
}
```

FIGURE 4.3 – Définition arbitraire d'une méthode générale

tion est vraiment une très mauvaise modélisation de la réalité, surtout lorsque telle méthode d'affichage ne soit pas redéfinir dans des sous-classes, puisqu'on aura des données pour des personnes qui ne s'affichent pas. En plus, cette solution n'exige pas la redéfinition de telle méthode dans des sous-classes. Alors, on doit mettre une méthode d'affichage dans la classe *Personne* pour permettre l'appelle à telle méthode, et également on doit imposer à chacune des sous-classes *Etudiant* et *Enseignant* l'affichage de données avec leurs propres méthodes spécialisées. En fait, on ne peut pas définir une méthode d'affichage au niveau de la super-classe, mais souhaite nécessairement avoir une méthode polymorphique au niveau des sous classes. Supposons qu'on ne sache pas afficher une personne générique, et de plus, on doit imposer que

cette méthode $affiche r$ soit redéfinie dans les sous-classes. En d'autres termes, on doit signaler à un niveau plus abstrait que telle méthode d'affichage doit exister par redéfinition dans chacune des sous-classes. Il s'agit alors ce qu'on appelle en termes de jargon une méthode abstraite. En *Java*, pour définir une méthode abstraite, on adopte le modificateur *abstract* sans aucune définition de corps, puisqu'une méthode abstraite ne peut pas la définir dans la super-classe où elle se trouve, mais elle a pour but d'imposer sa redéfinition dans des sous-classes qu'on souhaite plus des classes abstraites.

4.1.2 Classes Abstraites

Similairement aux méthodes abstraites, on adopte le modificateur *abstract* pour définir les classes abstraites au travers de la tournure (***abstract class IdentificateurClasse***). Dans une classe abstraite, on peut définir plusieurs méthodes abstraites. Si on reprend notre exemple d'hiérarchie de personnes, où il est impossible de définir la méthode *commeChain* générale dans la classe *Personne*. Dans ce cas, la méthode *commeChain* doit être une méthode abstraite, et également la classe *Personne* une classe abstraite. On a ainsi, les sous-classes *Etudiant*, *Enseignant*, *Responsable*, etc. La classe *Personne* est une classe abstraite, où on rajoute le modificateur *abstract*. La classe *Personne* contient deux méthodes, la méthode *commeChain* dont laquelle on a rajouté le modificateur *abstract*. On a aussi la méthode abstraite $infoSpcifique$ qui offre des informations spécifique à une personne, où on ne peut pas la définir au niveau de la classe *Personne* (voir figure 4.4). Comme il est indiqué par la figure, on peut utiliser une méthode abstraite (*commeChain*) pour définir une autre méthode non-abstraite ($affiche r$). Cependant, il faut bien noter qu'une classe abstraite est une classe qui n'autorise pas des opérations d'instanciations, alors il est impossible de créer des objets au travers de ses constructeurs. Ainsi, une classe abstraite ne peut contient aucune méthode abstraite, alors qu'une

```
abstract class Personne{
 String Nom;
 String Prénom;

 abstract String commeChain();
 abstract String infoSpcifique();

 void affiche (){
 System.out.print(Nom + Prénom + commeChain);
 }

}//_____________________________________________________________
class Etudiant extends Personne{

}//_____________________________________________________________
class Enseignant extends Personne {

}//_____________________________________________________________
class Responsable extends Personne {

}
```

FIGURE 4.4 – La classe abstraite *Personne*

méthode abstraite ne peut être défini que dans une classe abstraite. En outre, les sous-classes d'une classe abstraite restent toujours abstraites tant qu'elles ne redéfinissent pas toutes les méthodes abstraites. En d'autre terme, elles restent abstraites tant qu'il y à une méthode abstraite héritée de plus haut qui n'a pas encore redéfinie. Par exemple, si on suppose qu'on a une méthode abstraite *commeChain* dans la classe abstraite *Personne*, alors la sous-classe *Etudiant* de la classe *Personne* reste une classe abstraite tant que la méthode abstraite *commeChain* n'est pas encore redéfini dans la classe *Personne*. Dans ce cas, on doit définir la classe *Etudiant* comme une classe abstraite au moyen du modificateur *abstract*, et également la création des instances pour les classes *Personne* et *Etudiant* est forcément impossible.

4.2 Restriction de réutilisation

Cette section a pour but de présenter la restriction de modification de valeurs de variables, la restriction de spécialisation de méthodes, et la restriction d'extension de classes. En d'autres termes, elle introduit l'utilisation du modificateur *final* en se focalisant aux attributs, aux méthodes, et aux classes. . En *Java*, Le modificateur *final* est adopter à des variables pour définir des constants. Il est aussi applicable à des méthodes pour empêcher leur redéfinition dans des sous classes. Le modificateur *final* est aussi supportable par classes pour interdire leur extension. Ce modificateur est très courant pour la définition des constantes, par contre il est moins utile pour les méthodes et les classes.

4.2.1 Restriction de modification des attributs

En programation OO, une variable peut être un attribut (statique ou d'instance) pour une classe, un paramètre de méthode, ou et une variable locale dans un corps d'une méthode. Lors de l'utilisation du modificateur *final* pour ladéclaration de telle variable, il devient alors impossible de lui affecter une valeur plus d'une fois. Par conséquence, elle devient une constante avec une seule valeur permanente. En revanche, un attribut peut être initialisé au moyen de constructeurs, et certainement aucune autre modification ne soit autorisée. Évidemment, si on souhaite définir un attribut au travers de modificateur *final*, alors il est possible de l'initialiser soit au niveau de la déclaration, soit aux niveaux des constructeurs de la classe (n'importe quel constructeur), mais pas à tous ces niveaux. Ainsi, si on souhaite l'initialiser au moyen des constructeurs, alors tout constructeur de la classe doit comporter une instruction d'initialisation de l'attribut, sinon on aura un message d'erreur du compilateur qui informe que la constante (l'attribut final) n'est pas encore initialisée. Une fois initialisée, elle ne peut plus être modifiée. Si on essaye pour une nouvelle

fois de lui affecter une autre valeur dans un constructeur ou bien dans une méthode, alors on aura un message d'erreur du compilateur. Tandis que, s'une variable finale contient comme valeur une référence vers un objet (variable de type classe), alors on ne peut pas de lui affecter une autre valeur, mais il est tout à fait possible de modifier l'objet référencé au travers d'une autre variable. Par exemple, pour la classe *Etudiant* où on a l'attribut *Groupe*. Dans la classe *Etudiant* on a le manipulateur *setGroupe* pour l'attribut *Groupe*. On a ainsi, dans la méthode *main* la variable *Groupe* dédier pour affecter une valeur à l'attribut entier (voir figure 4.5). On a dans la variable E une référence vers un objet de type *Etudiant* dont valeur de l'attribut *Groupe* égale à 1. L'appelle de la méthode $modifier$ a pour particularité que l'instance de la classe *Etudiant* passé en paramètre est déclaré comme $final$. On peut considérer que l'instance passé en paramètre ne soit pas modifiable localement dans la méthode $modifier$. En d'autres termes, l'argument lui-même n'est pas modifiable, alors on ne peut pas lui affectant une nouvelle référence. Par contre l'objet référencé est absolument modifiable au travers E, même si E est passé comme paramètre final. Alors, le paramètre $final$ n'autorise pas la variable passée en argument d'avoir pointée vers un autre objet (il est impossible de modifier la référence elle-même), par contre rien n'empêche de modifier l'objet référencé au travers l'argument final. Ce genre de modification au moyen de références est constamment possible pour toute variable finale (un attribut, une variable locale, ou un paramètre de méthode). Effectivement, lors de l'affectation d'un objet à une variable finale (la variable ne peut référencer que cet objet), alors cet objet ne soit pas protégé de toute modification depuis l'extérieur. En outre, puisque le langage *Java* adopte uniquement le passage par valeur, alors tout paramètre final ne peut être modifié à l'intérieur de la méthode tout en protégeant cette modification à l'extérieur de la méthode.

```
class Etudiant extends Personne {

 private String Nom;
 private String Prénom;
 private int Groupe = 1 ;

 void changerGroupe(String groupe){
 Groupe = groupe;
 }
}
```

```
class GestionArtificielle {

 public static void main (String[] args) {

 Etudiant E = new Etudiant();
 E.changerGroupe(2);

 modifier(E);
 }
 void modifier( final Etudiant etudiant){
  etudiant.changerGroupe(3); // modification de l'objet
   référencé !!
  etudiant = new Etudiant(); // erreur
 }
}
```

FIGURE 4.5 – Modification de l'objet référencé

4.2.2 Restriction de redéfinition de méthode

Le modificateur *final* peut être aussi adopté à des méthodes pour empêcher leur redéfinitions dans des sous classes. Une méthode finale est une méthode usuelle déclarée au niveau d'une classe en interdisant sa redéfinition éventuelle dans une sous-classe par le biais du modificateur *final*. Partant d'un exemple où on suppose maintenant que l'hiérarchie de classes dispose d'une méthode $affiche r$ (en impli-

quant le nom et le prénom de la personne) qui l'on souhaite constamment la même pour tous les instances de ces classes. Alors, la définition comme *final* de la méthode *afficher* dans la super-classe *Personne* empêche alors toute sous-classe de la super-classe *Personne* (ex sous-classe *Etudiant*) de redéfinir cette méthode de manière spécifique. Dans ce cas, il est impossible d'avoir affiché les données des étudiants par exemple d'une manière plus spécifique que les autres personnes, en redéfinissant la méthode *afficher* dans la sous-classe *Etudiant*, puisque la méthode *afficher* est définie maintenant comme *final* dans la super-classe *Personne* (voir figure 4.6).

```
class Personne{

 private String Nom;
 private  String Prénom;

 public Personne(String nom, String prénom){
  Nom=nom;
  Prénom=prénom;
 }

 final void afficher(){
  System.out.println("Le nom de la personne est : " + Nom + ",
    et son prénom est :"+ Prénom);
 }
}
```

FIGURE 4.6 – Restriction de la redéfinition de la méthode *afficher*

4.2.3 Restriction d'extension de classe

Similairement à la restriction de redéfinition de méthode, on adopte aussi, le modificateur *final* pour la restriction d'extension des classes. Concrètement, pour la déclaration d'une classe comme *final*, on doit mettre le modificateur *final* devant la déclaration usuel de la classe. Par exemple, si on veut pour une raison ou une

autre que la classe *Etudiant* ne peut être jamais étendue par des sous-classes, on doit la déclarer comme classe finale. Dans ce cas, si on essaye de créer par une relation d'héritage une classe en étendant la classe *Etudiant*, alors on aura un message d'erreur du compilateur. Alors l'application du modificateur *final* à des classes empêche leurs extension par des liens d'héritage, et leurs méthodes sont a priori un peu agaçantes (voir figure 4.7).

```
final class Etudiant extends Personne{
 private String Section;
 private String Groupe;
}
```

FIGURE 4.7 – Restriction d'extension de la classe *Etudiant*

En effet, ce genre restriction de définition des sous-classes bride la liberté du programmeur d'extensions qui voudrait étendre une hiérarchie préexistante, et également de redéfinir des méthodes héritées de plus haut. Si on suppose par exemple, qu'on a une sous-classe *MyString* qui étend par un lien d'héritage la classe prédéfinie *String*. Supposer qu'on souhaite redéfinir dans la sous-classe *MyString* la méthode *substring* préexistante de la classe *String* qui permet d'extraire une sous-chaîne de caractères d'une chaîne de caractères. Évidemment, il est tout à fait possible de ne pas respecter la sémantique usuelle de la méthode *substring*, malgré qu'il est déconseillé. On peut par exemple de définir la méthode *substring* avec un comportement différent de ceu de la méthode *substring* de la classe *String*. Par conséquence, on peut déclarer une variable de type *String*, ensuite lui affecter une instance de type *MyString* grâce du lien d'héritage. Lors de l'exécution, on peut invoquer la méthode *substring* de la classe *MyString* sur cet instance (puisque le langage *Java* supporte la résolution dynamique de liens), et on peut avoir un comportement imprévu. Alors, à partir du code qui utilise la classe *MyString*, on pout avoir l'impression d'être

entrain de travailler avec des instances de la classe *String* en s'attendant au comportement usuel de la méthode *substring*. Par contre, il s'agit d'un autre comportement relatif à la nouvelle redéfinition de la *substring* dans la sous classe *MyString*. Pour éviter ce genre de dérives, les concepteurs du langage *Java* ont définit en fait la classe *String* comme *final*. En d'autres termes, pour pouvoir fixer une fois pour toute le comportement de la classe *String*, et également de ses méthodes. Par conséquent, on ne pout jamais définir une sous classe par héritage de la classe *String* en adaptant certaines méthodes par redéfinition où en ajoutant d'autres nouvelles méthodes. Cependant, ce genre de garde-fou aboutit par fois à une perte de certaines libertés très intéressantes pour des programmeurs d'extensions. Par exemple, si on souhaite de redéfinir la méthode *substring* même en préservant la sémantique originale afin d'adapter l'algorithme d'extraction de chaine de caractères à un contexte bien particulier. Malheureusement, ce genre d'adaptation de la classe *String* est absolument insupportable par les concepteurs du langage objet *Java*.

4.3 Membres de Statiques

On a déja vu que la spécification des membres (des attributs et des méthodes) dans une classe est proprement spécifique à chaque instance de la classe. On a vu que toute instance (objet) de la classe *Etudiant* par exemple a son propre *Nom*, son propre *Prénom*, sa propre *Note_ TD*, sa propre *Note_ Examen*, etc. il s'agit des informations qui sont spécifiques caractérisant chaque objet. Dans cette présente section, on va présenter la manière de modélisation d'un ensemble des informations communes et partagées pour tous les instances (objets) d'une même classe, ce qu'on appels en OO des membres statiques ou (membres de classes).

4.3.1 Attributs statiques

Pour les déférents exemples de notre cours, on a adopté des variables d'instances (des attributs), des variables locales déclarées à l'intérieur du corps de méthode, et également des variables locales déclarées comme paramètres de méthodes. Dans cette section, on va adresser la manière de définition des attributs statiques (attributs de classe). Il s'agit en fait d'attributs particuliers dont la syntaxe de déclaration est certainement similaire à selle de déclaration des attributs usuels (d'instances), et en adoptant cette fois-ci un modificateur statique sous certaines modalités un peut particulière. Tandis que, les attributs de classe (les attributs statiques) doivent nécessairement être déclarés dans des classes en dehors de toutes méthodes. Similairement aux attributs d'instance, les attributs de classe sont visibles partout dans une classe. Ainsi, on peut également adopter les modificateurs (*private* et *public*) pour restreindre ou étendre leur visibilité au monde extérieur. Évidemment, on peut adopter le modificateur d'autorisation d'accès par défaut (accès *friendly*) carrément de même façon que les attributs d'instances. Contrairement à un attribut d'instance, un attribut de classe (un attribut statique) est utilisable sans aucune obligation de création d'instance. En effet, on adopte une des facettes carrément au travers l'identificateur de la classe. Puisque les attributs statiques sont communs à toutes les instances de la classe (ils sont des attributs des instances d'une même classe), on peut utiliser ainsi la même facette procédée pour les attributs d'instance. Pour ce faire, le langage de programmation OO *Java* offre le modificateur *static* pour définir de tels attributs. Évidemment, si on reprend notre classe *Etudiant* où on utilise des attributs d'instances déclarés à l'intérieur de la classe, il sera impossible de les accéder qu'après avoir créé une instance de la classe *Etudiant*. Comme on déjà vu, chaque instance à ses propres attributs dont chacun dispose sa propre zone mémoire. Alors, chaque instance de la classe *Etudiant* crée dans un programme à

ses propres zones mémoire pour les attributs *Nom*, *Prénom*. Par exemple, pour l'instance de $E1$ de la classe *Etudiant*, on a les zones mémoire stockant la valeur des attributs *Nom*, *Prénom* pour cette instance. Pour une autre instance de la même classe *Etudiant*, on a d'autres zones mémoires stockant d'autres valeurs pour les attributs *Nom* et *Prénom* de cette deuxième instance. Il est claire que chaque étudiant à son propre nom et son propre prénom, c'est-à-dire que chaque instance de la classe *Etudiant* à ses propre données. C'est pour cette raison d'ailleurs qu'on parle de données d'objet (d'attributs d'instance), puisque chaque instance dispose de ses propres attributs. Cependant, un attribut de classe (statique) dispose d'une zone mémoire unique pour toutes les instances de la classe, une selle zone mémoire partagé entre toutes les instances d'une même classe. Une zone mémoire unique qui est accessible par toutes les instances d'une même classe. En d'autres termes, pour un attribut d'instance, il y à une réservation d'une zone mémoire propre à chaque objet créé, alors que pour un attribut de classe (un attribut statique), il y à exactement une réservation d'une seule zone mémoire partagée pour toutes les instances d'une classe même si aucun objet ni créé. Une réservation de telle zone mémoire est faite une fois la classe est mentionnée dans un programme au moment de son chargement, et aucune d'autres allocation mémoire ni possible pour tel attribut lors de la création de nouvelles instances. Cette zone mémoire reste toujours accessible par tous les anciens et les nouveaux objets créés de la même classe. Partant d'un exemple plus concret où considérant cette fois-ci que les étudiants du département informatique. On considère comme d'habitude la classe *Etudiant* avec les deux attributs d'instances *Nom* et *Prénom*, ainsi que un nouveaux attribut *Spécialité* qui est cette fois-ci un attribut de classe dont la déclaration est précédée par le modificateur *static*. La classe *Etudiant* contient également une méthode usuelle *set*, qui accède à tous les attributs (de classe ou d'instances) de la classe (voir figure 4.8). En fait, une fois que la classe *Etudiant*, qui adopte une variable statique *Spécialité*, est

```
class GestionArtificielle {
 public static void main(String[] args) {
  Etudiant.Spécialité = "Math";
  Etudiant E;
  E = new Etudiant("Lamouri", "Amine");
  E.set();
  E.Spécialité = "Electronique";
 }
}//_____________________________________________________________
class Etudiant {
 String Nom, Prénom;
 int Groupe = 2; // Attribut d'instance
 static String Spécialité = "Informatique"; // Attribut de
   classe
 Etudiant (String nom, String prénom {
 Nom = nom;
 Prénom=prénom;
 }
 void set() {
  Groupe = 3;
  Spécialité = "Médecine";
 }
}
```

FIGURE 4.8 – Attribut statique

utilisée dans la méthode *main* de la classe *GestionArtificielle*, alors on aura une réservation d'un emplacement mémoire pour cette variable statique *Spécialité*. Dans ce cas, la variable statique . Comme il est indiqué par la figure, la variable de classe *Spécialité* est initialisée avec la valeur $Informatique$, et elle est accessible au travers de la classe *Etudiant* indépendamment de toute création d'instances. Lorsque l'on exécute la première lignela méthode *main*, on va affecter la valeur "$Math$" au variable de classe *Spécialité*, variable de classe de la classe *Etudiant*, laquelle a déjà été initialisée avec la valeur $Informatique$. On a, la déclaration d'une variable d'instance E de la classe *Etudiant*, et l'initialisent au moyen du constructeur usuel de deux arguments *nom* et *prénom*. Chaque instance de la classe *Etudiant* dispose

de ses propres attributs (attributs d'instance). En l'occurrence, l'attribut d'instance *Nom*, ce qui veut dire qu'au terme de l'exécution de cette ligne, la situation en mémoire sera la suivante, on aura une variable E contenant une référence vers un objet de type *Etudiant*, qui dispose des attributs *Nom*, *Prénom* et *Groupe* lesquels sont initialisés grâce à la tournure *E = new Etudiant("Lamouri", "Amine");* respectivement à "*Lamouri*", "*Amine*" et 2. La méthode *set* est ensuite appelée sur l'instance de E ce qui affecte la valeur de la variable d'instance *Groupe* à 3, ainsi la valeur de l'attribut statique *Spcialit* à "*Mdecine*". Ainsi, au moyen de la cinquième ligne la valeur de l'attribut statique *Spcialit* est modifiée à "*Electronique*" ; Alors, on notera que les méthodes de la classe *Etudiant* peuvent accéder aux attributs statiques exactementde la même façon qu'elles accèdent aux attributs non statiques (d'instances). On peut modifier la valeur de la variable de classe de l'attribut statique aussi bien en passant uniquement par l'identificateur de la classe qu'en passant au travers une instance. Évidemment, lorsqu'on modifie un attribut d'instance, la valeur change uniquement pour l'objet en question. Tandis que, lorsqu'on modifie un attribut de classe la valeur change pour tous les objets de la classe. En effet, l'attribut de classe est unique, et il est accessible par tous les objets de sa classe. Par conséquence, un attribut statique peut être manipulé au travers d'un identificateur de la classe sans passer par la création d'un objet, on peut alors adopter des membres statiques en outrepassant complètement des objets encontre l'approche OO. L'utilisation de tels attributs pour contourner l'approche OO est vraiment une mauvaise raison. Tandis que, la bonne raison est de représenter une valeur commune à tous les objets d'une classe. Partant d'un exemple plus illustratif, où on considère cette fois-ci la classe *Enseignant* en envisageant l'âge officiel de départ à la retraite pour tous les enseignants est 60 ans. On peut adopter un attribut d'instance pour modéliser l'âge de la retraite (voir figure 4.9). Comme il est indiqué par la figure, le constructeur de la classe la classe *Enseignant* permet d'initialiser chacun des attributs de la classe

```
class Enseignant {

 private String Nom;
 private int AgeRetraite;

 public Enseignant(String nom, int ageRetraite) {
  Nom = nom;
  AgeRetraite = ageRetraite;
 }
}
```

```
class GestionArtificielle {

 public static void main(String[] args) {

  Enseignant[] E = new Enseignant[600];
  E[0] = new Enseignant("Mostfiel", 60);
  E[1] = new Enseignant("Limame", 60);

  // La nécessité de parcours du tableau pour la modification
   de l'âge de la retraite, puisque chaque E a sa propre zone
   mémoir pour l'attribut AgeRetraite.
  for (int i = 0; i < E.length; i++)  E[i].AgeRetraite = 65;
 }
}
```

FIGURE 4.9 – La première version : variable d'instance

Enseignant. Ainsi, la classe *GestionArtificielle* utilise cette classe pour définir un tableau contenant un certain nombre d'éléments dont les valeurs sont des objets de la classe*Enseignant*. Le tableau est remplit en créant un ensemble d'instance cette classe*Enseignant*. Puisque l'âge officiel de la retraite est modélisé par le biais d'un attribut d'instance, alors pour chaque instance de la classe *Enseignant* on a une zone mémoire pour cet attribut *AgeRetraite* qui contient exactement la même valeur 60 ans pour tous les autres enseignants (toutes les autres instances de la classe

Enseignant). En conséquence, si on suppose qu'on a six cents d'enseignants (six cents instances de la classe *Enseignant*), alors on aura six cents fois la valeur 60 en mémoire. Malheureusement, il s'agit d'une duplication indésirable. En revanche, elle pose un problème très sérieux de maintenance, lorsde changement de la loi, où l'âge de départ à la retraite est devient 65 ans par exemple. Dans ce cas, on doit revoir convenablement toutes les six cents instances pour attribuer la nouvelle valeur 65 à l'attribut *AgeRetraite*. Pour pousser le problème encore une nouvelle fois, imaginer que les instances ont été créées de façon indépendante, et on n'a pas les stockées dans un tableau, alors on aura une grande difficulté de maintenance. Cependant, On peut également opter un attribut de classe pour (un attribut statique) pour modéliser l'âge de départ à la retraite (voir figure 4.10). En effet, on a une selle zone mémoire, au lieu de six cents zones mémoires, qui contient la valeur âge de la retraite partagée pour toutes les six cents instances. Cette zone mémoire est unique, et elle est accessible via l'identificateur de la classe. En outre, elle est accessible au travers toute instance, et sans aucune duplication d'informations. En revanche, on a plus de difficulté de maintenance, puisque il est plus nécessaire de revoir cette fois-ci toutes les six cents instances pour faire adapter la valeur de l'âge de la retraite à 65 ans. Il est en fait, très indispensable d'adopter des attributs de classe (des attributs statiques) pour modéliser les données communes à toutes les instances d'une classe. Après avoir présenté la notion d'attribut statique, il est temps de comprendre une instruction très courante en *Java*. Il s'agit de l'instruction familiale *System.out.println*. Bien qu'on utilise toujours *out* directement au travers le mot clé *System*. Alors il est bel et bien que *System* est une classe, et également *out* un attribut statique de cette classe. En outre, puisque'on fait appel la méthode *println* sur cette attribut statique, alors *out*, en plus d'être un attribut statique de la classe *System*, est certainement une référence vers un objet d'une autre classe. En fait, l'attribut *out* est une variable d'instances de la classe *PrintStream*, et la

```
class Enseignant {
 private String Nom;
 static private int AgeRetraite;
 public Enseignant(String nom, int ageRetraite) {
  Nom = nom;
  AgeRetraite = ageRetraite;
 }
}
```

```
class GestionArtificielle {
public static void main (String[] args) {
  Enseignant[] E = new Enseignant[600];
  E[0] = new Enseignant("Mostfiel", 60);
  E[1] = new Enseignant("Limame", 60);
  // Aucun parcours du tableau nécessaire pour la modification
    de l'âge de la retraite
 Enseignant.AgeRetraite = 65; //modification au moyen de l'
   identificateur de la classe.
 E[0].AgeRetraite = 65; // modification alternative au moyen
   de la première instance du tableau.
 // ....
 E[600].AgeRetraite = 65; // modification alternative au moyen
    de la première instance du tableau.
}
}
```

FIGURE 4.10 – La seconde version : attribut de classe

méthode *println* est une méthode d'instances de la de cette classe *PrintStream*.

4.3.2 Méthodes statiques

Après avoir présenté les attributs de classe, dans cette section on va adressé un autre membre statique ce qu'on appelle communément les méthodes de classe (les méthodes statiques). Similairement aux attributs statiques, le modificateur *static* est aussi adopter pour définir les méthodes de classe. Des méthodes qui peuvent être

invoquées sans aucune nécessité de création des objets. Si on reprend par exemple la classe *Enseignant* dans laquelle on définira une méthode de classe (méthode statique) $affichAgeRetraite$, ainsi qu'une méthode usuelle (méthode d'instance) $afficheNom$ (voir figure 4.11. Similairement aux attributs, on peut appeler la mé-

```
class Enseignant {

 private String Nom;
  static private int AgeRetraite = 60;

 void afficheNom() {
  System.out.println("Le nom de l'enseignant es : " + Nom);
 }
  static void afficheAgeRetraite() {
  System.out.println("L'âge de départ à la retraite est : "
   AgeRetraite);
}
}
class  GestionArtificielle {

 public static void main(String[] args) {

  Enseignant.afficheNom(); // Accès non autorisé !
  Enseignant.afficheAgeRetraite(); // Accès autorisé

  Enseignant E = new Enseignant();
  E.afficheNom(); // Accès habituel
  E.afficheAgeRetraite(); // Accès alternatif
 }
}
```

FIGURE 4.11 – Méthode statique

thode de classe $affichAgeRetraite$ carrément sans aucune création d'instances de la classe *Enseignant*. Pour ce faire, on adopte la syntaxe d'accès directe *identificateurclasse.identificateurMéthodeClasse*. Par exemple, pour appeler la méthode $afficheAgeRetraite$, on adopte l'instruction $Enseignant.afficheAgeRetraite$, ce qui permet d'afficher le message l'âge de départ à la retraite. Alors que, pour la

méthode d'instance $affichеNom$ (la méthode déclarée sans le mot clé *static*), on doit absolument créer au préalable une instance sur laquelle on va l'invoquée. En outre, on peut créer également une instance de la classe *Enseignant* pour une variable par exemple E, pour appeler la méthode statique avec la syntaxe $E.affichеAgeRetraite$. Il est constamment la même chose qu'avoir usé la syntaxe $Enseignant.affichеAgeRetraite$. Il s'agit alors d'une autre variante d'accès aux membres statiques. Tandis que, on adopter plutôt la première syntaxe (la syntaxe $Identificateurclasse.IdentificateurMéthodeClasse$) pour invoquer des méthodes de classe, puisque il est bien explicite de montrer l'intention afin de rappeler qu'il s'agit d'une méthode de classe en franchissant toutes ambiguës. Par opposition aux méthodes de classe, la méthode d'instance $affichеNom$, est invocable que sur l'existence d'une instance. Par exemple créant une instance pour le variable E, et également adopter l'instruction $Enseignant.affichеNom$ afin de pouvoir afficher le *Nom* de l'enseignant en question. De plus, Une méthode de classe n'utilise ni de variables d'instances, ni de méthodes d'instances. En outre, la référence *this* n'est plus disponible pour telles méthodes, puisque la disponibilité des objets pour des classe qui adoptes ce genre de méthodes n'est jamais garanti. Par conséquence, une méthode statique ne manipule que des attributs et des méthodes statiques. Portant un exemple où on considère deux méthodes d'instances, et une méthode statique. En outre, on utilise une attribut d'instance, ainsi que un attribut statique (voir figure 4.12). Dans une méthode d'instace (ex $getInf$), on peut appeler des méthodes statiques comme $getAgeRetraite$. On peut invoquer également des méthodes d'instances comme $getNom$. Évidemment, pour une méthode d'instance le tout est absolument autorisé et certainement possible. Cependant, pour une méthode de classe, on ne peut accéder jamais aux membres (aux attributs et aux méthodes) d'instances, mais uniquement aux membres de classe (aux attributs statiques et aux méthodes statiques). En conséquence, si jamais on essaie dans une méthode

```
class Enseignant {
 String Nom;
static int AgeRetraite = 67;

 String getNom (){
 return Nom;
 }

static int getAgeRetraite (){
 return AgeRetraite;
 }

 String  getInfo (){
 return getNom() + " " + getAgeRetraite();
 }

}
```

FIGURE 4.12 – Utilisation des méthodes statiques dans celles d'instances

de classe d'accéder à un attribut d'instance comme *Nom* par exemple, ou à une méthode d'instance comme *getNom*, alors on aura forcement une erreur de compilation, parce que aucune création d'une instance de la classe n'est assurée. Alors, dans une méthode statique, il est tout à fait possible d'appeler des méthodes statiques, et y compris elle-même, et on doit faire attention au problème de récursivité infinie en gérant les appelles par des contraintes d'arrêts. Évidemment, les méthodes statiques (méthodes de classe) ne sont pas des méthodes spécifiques à des instances bien particulières d'une classe, mais elles sont des méthodes très générales et communes pour toutes les instances d'une même classe. Très souvent, elles sont adoptées pour des mécanismes d'outillage. Des mécanismes pour les quelles des classes boîte à outils sont exploités en définissant des méthodes statiques, et en ignorant la nécessité de création des instances de telle classes. Par exemple, on peut créer une boîte à outils mathématiques, par le biais de la classe *MathLib* (voir figure 4.13)

en définissant la constante de classe PI, et la méthode statique *périmètreCercle* qui permet de calculer le périmètre d'une cercle dont le Rion R soit fournit comme paramètre. Alors, il est bel et bien, d'utilisé directement la constante de classe PI,

```
class MathLib {
 public final static double PI = 3.14159265358979323846;
 public static double auCube(double d) {
 return d*d*d;
 }
}
```

FIGURE 4.13 – La class MathLib

et également d'accéder à la méthode statique *périmètreCercle* (voir figure 4.14). En outre, l'instanciation de la classe *MathLib* est carrément artificielle, et il devient absolument inutile d'avoir créé des instances de la classe *MathLib*. Dans ce cas, la classe *MathLib* est adoptée comme un mécanisme d'assemblage des méthodes et des constantes d'ordres utilitaires. Il est exactement le but de la classe *Math* envisagé par les concepteurs du langage objet *Java*. Absolument comme l'exemple pédagogique *MathLib*, en utilisant l'API *Java* standard, on peut accéder à la constante PI au travers la syntaxe $Math.PI$. Cependant, il faut noter en toute rigueur d'avoir utilisé les méthodes et les données utilitaires fourni par l'API en évitant toute redéfinition

```
class Calcul {
 public static void main(String[] args) {
 double x = 5.7;
 double y = MathLib.PI * MathLib.auCube(x);
 System.out.println(y);
 }
}
```

FIGURE 4.14 – Utilisation de la classe MathLib

explicite. Généralement, les classes dont les membres sont statiques sont utilisées pour crée des boîtes à outils. Par conséquent, il faut qu'on fuit parfaitement d'avoir abusé et proliféré les membres statiques, et en les conservent bien spécifiquement pour des situations suivantes :

- pour définir des constantes.
- pour définir un attribut de classe, afin de pouvoir partager une valeur commune pour toutes les instances de même classe, mais qui peut évoluer (n'est pas un constatant). Il s'agit d'une situation assez rares à utiliser.
- pour définir des méthodes de classe, lorsqu'il est artificiel de créer des instances particulières en favorisant l'accès direct par le biais de l'identificateur de la classe. Il s'agit d'une situation beaucoup plus rare, puisque généralement on dispose celles prédéfini par le langage de programmation OO.

Chapitre 5

Héritage Multiple et Interfaces

Sommaire

Après avoir présenté dans les chapitres précédents les fondamentaux de l'approche OO (l'abstraction, l'encapsulation, l'héritage), ce chapitre à pour objectif de présenter un très bon complément du concept d'héritage, où impliquant un certains nombre de classes, ce qu'on appel communément l'héritage multiple. En outre, il introduit aussi un concept très important pour proprement lié à la programmation

OO en *Java*, ce que l'on appelle en termes de jargon *interface*.

5.1 Héritage Multiple

Reprenant notre célèbre exemple de gestion des individués d'une université. On considère cette fois-ci des étudiants, des enseignants, des chefs de département, des doyens. En OO, on doit organiser toutes ces entités au travers de la notion de classe. Évidemment, on peut conçue une classe dite *Responsable* pour gérer les chefs de département et les doyens. Une classe *Enseignant* pour gérer les enseignants. Une classe *Etudiant* pour la gestion des étudiants, etc (voir figure 5.1). Alors, selon cette conception, on aura ce que l'on appelle en termes de jargon l'héritage multiple. Par exemple, un doyen hérite de façon multiple d'un enseignant et d'un responsable. Un chef de département hérite ainsi à la fois d'un enseignant et d'un responsable. Malheureusement, certains le langage de programation ne supporte pas de tel héritage multiple, puisqu'il y a des situations difficile à comprendre le sens qu'on veut vraiment donner lieu. Par exemple, lorsqu'on a une classe qui hérite de façons multiples d'une super-classe, alors les instances de lasuper- classe peuvent être considérer comme une même instance dans quelques situations, et comme des instances dis-

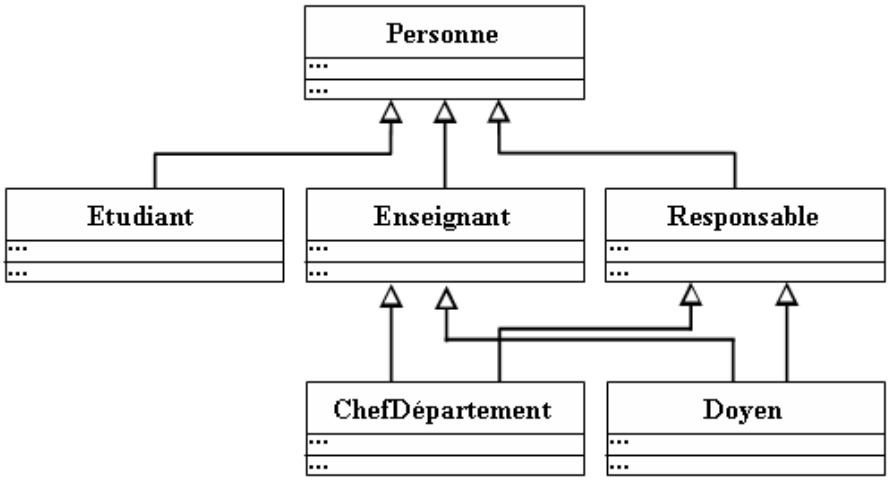

FIGURE 5.1 – Héritage Multiple

tinctes pour d'autres situations. Ce genre de situations est très difficile à comprendre par le compilateur, puisque il y à des décisions à des décisions d'ordre conceptuel à prendre. En outre, tel héritage n'est pas supporté a cause de l'ambiguïté rencontré lorsqu'un membre (méthode ou attribut) est déclaré dans plusieurs niveau de l'hiérarchie des classes, puisqu'on aura un problème de choix du membre à utiliser, et la manière de l'accéder. En revanche, si on souhaite mettre en oeuvre des méthodes communes pour certaines classes. Par exemple, on veut imposer aux classe *ChefDépartement* et *Doyen* de mettre en oeuvre une méthode communes dite *gestionRéunion* pour la gestion des réunions (voir figure 5.2). Évidemment, la méthode

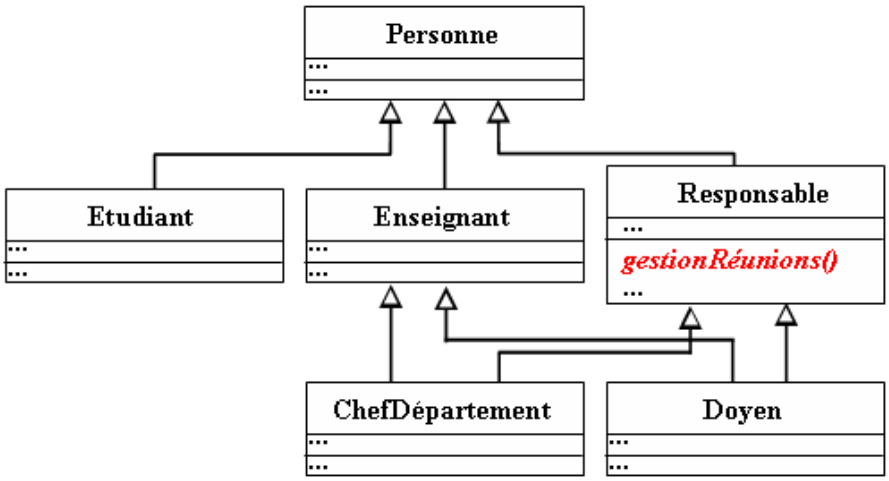

FIGURE 5.2 – Une méthode commune au niveau de la classe *Responsable*

gestionRéunion ne doit pas être une méthode pour leur super-classe (voir figures 5.3 et 5.4), puisque elle n'est pas toujours une méthode commune à tous les sous classe de leur super-classe. Par exemple, la sous classe *Etudiant* ne doit pas avoir de telle méthode de gestion de réunion. Similairement pour la sous classe *Enseignant*, puisqu'on a des enseignants qui ne gèrent pas des réunions. Par conséquent, certains langages de programmation décident comme une bonne solution de ne pas adopter l'héritage multiple, mais ils adoptent plutôt la notion d'interface pour l'assimiler seulement.

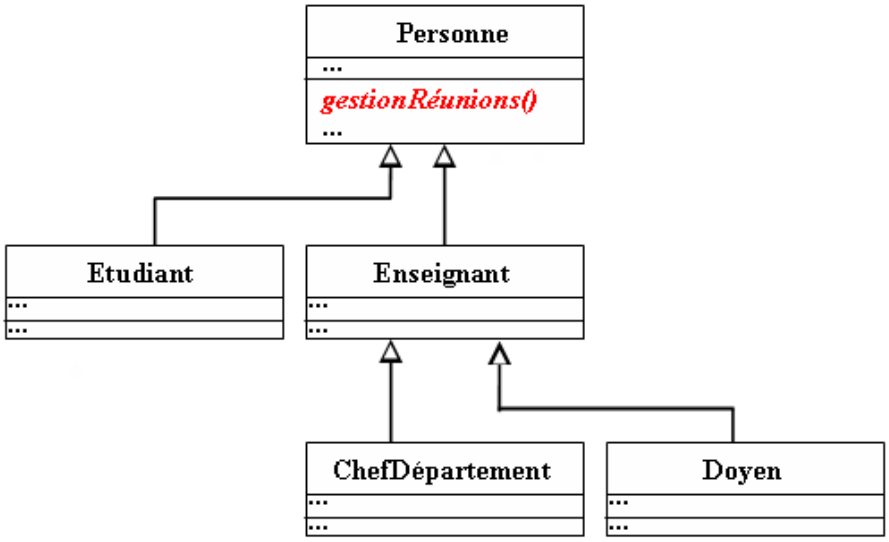

FIGURE 5.3 – Une méthode commune au niveau de la classe *Personne*

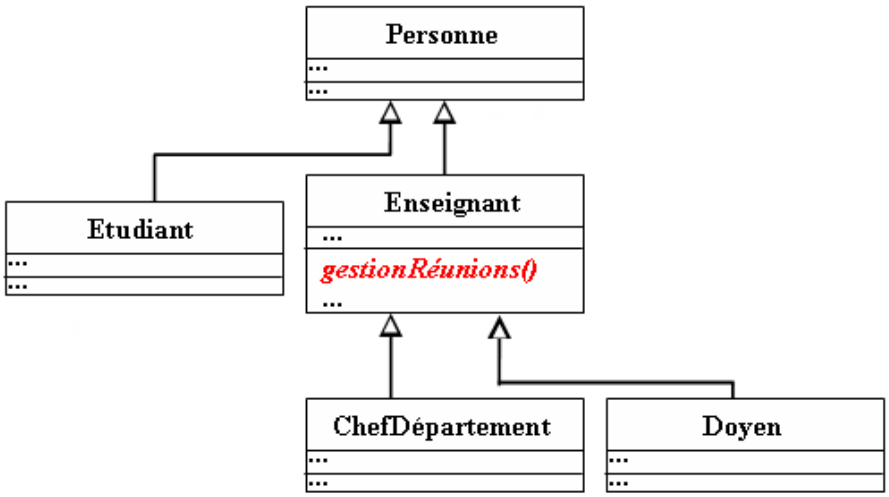

FIGURE 5.4 – Une méthode commune au niveau de la classe *Enseignant*

5.2 Interface

En java, par le biais de la notion d'interface, on peut imposer un contenu commun à certaines sous-classes en dehors de toute relation d'héritage, et indépendamment à toutes autres sous-classes. La notion d'interface est totalement différente de celle de classe. Elle a une existence propre, et elle permet d'imposer à certaines classes un contenu divers et particulier. Par exemple, les sous classe *Doyen* et *ChefDépartement*

qui héritent la classe *Enseignant* ont l'interface *Responsabilité* qui exige la gestion des réunions. En d'autres termes, les sous-classes *Doyen* et *ChefDépartement* ont, en plus de l'extension de la classe *Enseignant*, l'interface *Responsabilité* qui les impose d'avoir gérer les réunions. Ce genre d'obligation de gestion de réunions au niveaux des sous classes *Doyen* et *ChefDépartement* est conçu sans aucune implication de la méthode *gestionRéunion* dans la classe *Personne*, et par conséquence on a empêché les sous classe *Etudiant* et *Enseignant* d'avoir une méthode *gestionRéunion* dont on n'a certainement rien à faire (voir figure 5.5). Concrètement, la déclaration d'une interface est un peu similaire à celle d'une classe en adoptant le mot clé *interface* au lieu du mot clé *classe*. Similairement à tout identificateur en programmation, l'identificateur de l'interface est librement choisi (voir figure 5.6).

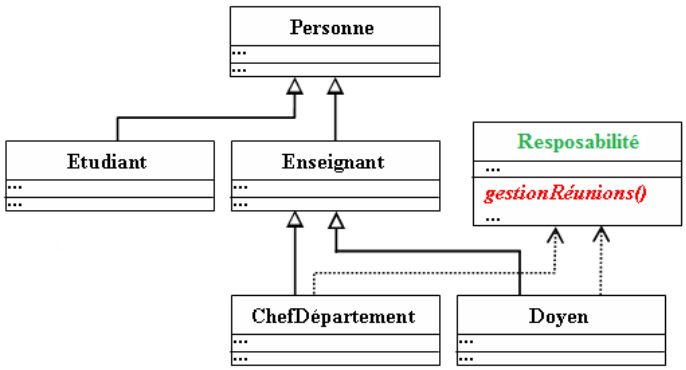

FIGURE 5.5 – Une méthode *gestionRéunion* au niveau de l'interface *Responsabilité*

```
interface Responsabilité {
  void gestionRéunion();
}
```

FIGURE 5.6 – L'interface *Responsabilité*

5.2.1 Implémentation d'Interface

Après avoir présenté la notion d'interface, et sa déclaration concrète en *Java*, on va s'intéresser maintenant à la manière d'établissement de lien entre une interface et une classe. En java, un lien entre une interface et une classe est établit au travers du mot clé *implements*. Lorsque de déclaration d'une classe on peut indiquer, au moyen du mot clé *implements*, que cette classe implémente un certain nombre d'interfaces (voir figure 5.7). Par exemple, pour exprimer le fait que la sous classe *Doyen* étend la sous classe *Enseignant* et implémente l'interface *Responsabilité* (voir figure 5.8). Dans ce cas, un doyen est perçu comme un enseignant responsable. En d'autres termes, un doyen est considéré comme un enseignant, et qui se comporte comme un responsable. Cependant, lorsqu'une classe implémente un certain nombre d'interfaces, et si on souhaite qu'elle soit instanciable, elle doit redéfinisse toutes les

```
class IdentificateurClasse implements IdentificateurInterface1
    , IdentificateurInterface2, ... {
  // ...
}
```

FIGURE 5.7 – Syntaxe d'implémentation des interfaces

```
interface Responsabilité {
  int afficheNombreConseilsPédagogiques();
}

class Doyen extends Enseignant implements Responsabilité {

}
```

FIGURE 5.8 – Implémentation de l'interface *Responsabilité*

méthodes abstraites déclarées dans les interfaces. Grace à ce genre d'obligation de redéfinition de méthodes abstraites, on offre à une interface la capacité d'imposer un contenu commune à certains classes qui l'implémentent (voir figure 5.9). Partant d'un exemple où on considère une classe qui implémente parfaitement plusieurs interfaces. Si on considére qu'un chef de département est un enseignant chercheur, et au même temps est un enseignant responsable, alors la classe *ChefDépartement* doit implémente les deux interfaces *Recherche* et *Responsabilité* (voir figure 5.10). Si on souhaite pouvoir créer des instances pour la classe *ChefDépartement*, alors on doit nécessairement définir les méthodes abstraites *nombreConseilsPédagogiques* et *nombrePapiersRédigés* (voir figure 5.11).

5.2.2 Extension d'Interface

Après avoir introduit ce que l'on peut mettre à l'intérieur d'une interface, et la manière d'établissement des liens entre les classes et les interfaces, on va s'intéresser dans cette section à la manière d'établissement des liens entre les interfaces elles mêmes. Similairement à l'extension des classes, il est tout à fait possible de définir une hiérarchie d'interfaces au travers du mot clé *extends* Par exemple, Si on considère l'extention de la super-interface *Responsabilité* avec les deux interfaces *RS-*

```
interface Responsabilité {
  int afficheNombreConseilsPédagogiques();
}

class Doyen extends Enseignant implements Responsabilité {
  public int afficheNombreConseilsPédagogiques(){
  return 7;
  };
}
```

FIGURE 5.9 – Redéfinition de la méthode abstraite de l'interface *Responsabilité*

```
interface Recherche {
  int afficheNombrePapiersRédigés();
}//_____________________________________________________________
interface Responsabilité {
  int afficheNombreConseilsPédagogiques();
}//_____________________________________________________________
class Enseignant {

}//_____________________________________________________________
class ChefDépartement extends Enseignant implements Recherche,
    Responsabilité {

}
```

FIGURE 5.10 – Implémentation des interfaces *Recherche* et *Responsabilité*

```
interface Recherche {
  int afficheNombrePapiersRédigés();
}//_____________________________________________________________
interface Responsabilité {
  int afficheNombreConseilsPédagogiques();
}//_____________________________________________________________
class Enseignant {
}//_____________________________________________________________
class ChefDépartement extends Enseignant implements Recherche,
    Responsabilité {
  public int afficheNombrePapiersRédigés(){
  return 4;
  };
  public int afficheNombreConseilsPédagogiques(){
  return 7;
  };
}
```

FIGURE 5.11 – Redéfinition des méthodes abstraites dans la classe *ChefDépartement*

pécialité et *RDomain*, alors on aura une hiérarchie d'interfaces. (voir figure 5.12). Contrairement, à l'extension des classes, en java, l'héritage multiple est supporté

pour les interfaces. En d'autres termes, il est tout à fait possible de définir une interface par extension de certaines autres interfaces préexistantes. Par exemple, si on considère les deux interfaces *Recherche* et *Responsabilité*. On peut définir un interface par exemple *RSpécialité* qui étant les deux autres interfaces *Recherche* et *Responsabilité* (voir figure 5.13).

5.3 Manipulation des Objets au moyen d'Interface

Similairement à une classe, la déclaration d'une nouvelle interface (en *Java*) permet de définir en fait un nouveau type. Il devient alors tout à fait possible de déclarer

```
interface Responsabilité {

}//_____________________________________________________________
interface RDomain extends Responsabilité {

}//_____________________________________________________________
interface RSpécialite extends Responsabilité {

}
```

FIGURE 5.12 – Extension des interfaces

```
interface Recherche {

}//_____________________________________________________________
interface Responsabilité {

}//_____________________________________________________________
interface RSpécialite extends Recherche, Responsabilité {

}
```

FIGURE 5.13 – Héritage multiple pour des interfaces

des variables au moyen des interfaces. Cependant, les interfaces sont abstraites (ne sont pas instanciables), alors on n'a pas des objets de type interface. En java, on peut affecter un objet de type classe pour une variable de type interface sachant qu'elle implémente cette interface. Par exemple, si on a une variable *Responsable* de type *Responsabilité*. Sachant que la classe *Doyen* implémente l'interface *Responsabilité*, alors on peut affecter à la variable *responsable* une instance de type *Doyen*. Pour certaines situations, il peut être très intéressant de manipuler des objets sous une étiquette d'interface, plutôt que sous une étiquette de classe. Par exemple, plutôt de manipuler un objet de type *Doyen*, on le manipule entant que objet de type *Responsabilité* en justifiant le type effectif d'affectation. Sachant qu'on peut convenablement affecter un objet de type sous-classe à une variable de type super-classe en affirmant le contenu dans la variable est un objet qui implémente l'interface en question par le biais de conversion de type (de transtypage). Par exemple, une variable contenant un objet de type *Doyen* qui implémente l'interface *Responsabilité*, il est toujours possible d'affecter à cette variable un objet de type *Enseignant* (voir figure 5.14). L'intérêt primordial des interfaces est de regroupé impérativement des membres communs à certains classes en d'hors de toutes relation d'héritage. Par exemple, à l'aide de l'interface *Responsabilité*, on impose un contenu commun, la méthode *nombreConseilsPédagogiques* aux classes *Doyen* et *ChefDépartement* qui ne sont pas liées entre elles par un lien d'héritage. Typiquement, on a apprécié que :

1. le concept héritage sert à modéliser une relation de type $est-un$. Par exemple un enseignant chercheur est un enseignant.

2. le concept attribut sert à modéliser une relation de type $a-un$, même s'il s'agit d'un objet d'une autre classe. Par exemple, si on suppose que les adresses des étudiants sont modélisé par une classe dit *Adresse*. Alors on dit un étudiant a une adresse. Dans ce cas, on a établit entre les deux classes *Etudiant* et *Adresse* une relation de type $a-un$. On dit ainsi un étudiant : a un nom,

```
interface Responsabilité {
 int afficheNombreConseilsPédagogiques();
}//-------------------------------------------------------------
class Enseignant {
 String Grade ;
 // ...
}//-------------------------------------------------------------
class Doyen extends Enseignant implements Responsabilité {
 public int afficheNombreConseilsPédagogiques(){
 return 7;
 };
}//-------------------------------------------------------------
class ChefDépartement implements Responsabilité {
 public int afficheNombreConseilsPédagogiques(){
 return 4;
 };
}//-------------------------------------------------------------
class GestionArtificielle {

 public static void main (){
 Responsabilité Responsable;
 Enseignant E = new Doyen ();
 Responsable = (Doyen)E; // ou bien
 Responsable = new ChefDépartement ();
 }
}
```

FIGURE 5.14 – Manipulation des objets au travers des interfaces

a un prénom, a une note de TD, et a une note d'examen. Évidemment, les attributs *Nom*, et *Prénom* sont de type classe (la classe prédéfinit *String*).

3. le concept interface sert à modéliser une relation de type *se-comporte-comme*, il permet en fait d'assurer qu'une classe se conforme à un certain protocole. Par exemple un doyen de faculté $est-un$ enseignant, et il se comporte comme un responsable. Un doyen de faculté $est-un$ enseignant, et il se comporte comme un chercheur. Un doyen de faculté $est-un$ enseignant, et il se comporte à la fois comme un responsable et comme un chercheur.

5.4 Membres d'Interface

En *Java*, une interface peut comporter des constantes, et des méthodes abstraites. On outre, à partir de la version *Java*8 qu'était publiée en avril 2015, elle peut comporter ainsi des méthodes statiques et des méthodes par défauts.

5.4.1 Constantes

Contrairement aux classes, l'initialisation d'une constante (attribut final) dans une interface doit être effectuée au moment de sa déclaration, puisqu'on n'a pas de constructeur éventuel pour attribuer une valeur initiale à cette constante. En java, il y a des modificateurs implicites pour les constantes des interfaces. Toute constante d'une interface est nécessairement *public, final, et static*. Alors, dans une interface on peut définir des constantes (des variables finales) statiques et publiques. En revanche, il n'est pas très courant de définir des constantes dans des interfaces, et il nécessite une certaine discipline bien particulière puisque la déclaration des constantes dans une interface peut être éventuellement une source d'ambiguïté prudente :

1. pour une classe qui comporte une constante, et qu'elle implémente une interface. Si l'interface comporte la définition de la même constante, alors on aura un problème d'accès à cette constante (la constante de la classe ou bien celle de l'interface). Par exemple, si on considère la classe *ChefDépartement* qui implémente l'interface *Recherche* dont chacune comporte la définition de la même constante *Nombre* (voire figure 5.15). Il est claire qu'il y a un problème d'accès à la constant *Nombre*. Alors, on considère la constante de l'interface *Recherche* ou bien celle de la classe *ChefDépartement* (voire figure 5.16). Tandis que, en *Java* la classe à la priorité de précédence, et c'est pour cette raison qu'il est autorisé par le compilateur.

2. pour une classe qui étend une super-classe, et qui implémente une interface où

on a une définition d'une même constante à la fois dans une super-classe et une interface, alors on aura un problème d'accès à la constante (la constante de la super-classe, ou bien celle de l'interface). Par exemple, si on considère maintenant, la classe *ChefDépartement* qui étant la classe *Personne*, et qui implémente l'interface *Recherche* dont la définition de la même constante *Nombre* est à la fois dans la super-classe *Personne* et dans l'interface *Recherche* (voire figure 5.17). Il est évident qu'il y a un problème d'accès à la constante *Nombre*. On considère la constante de l'interface *Recherche* ou bien celle de la super-

```
interface Recherche {
 int Nombre = 4;
}//_____________________________________________________________
class ChefDépartement implements Recherche {
 final static int Nombre = 1;
 }
}
```

FIGURE 5.15 – Situation conflictuelle : Classe - Interface

```
interface Recherche {

 int Nombre = 4;
}//_____________________________________________________________
class ChefDépartement implements Recherche {

 final static int Nombre = 1;
 void affiche() {
  System.out.println(Nombre); /* Situation conflictuelle */
  System.out.println(Recherche.Nombre);
 }
}
```

FIGURE 5.16 – Situation conflictuelle (Exemple) : Classe - Interface

classe *Personne*. Alors, a cause de ce genre de problèmes qu'il ne soit pas autorisé par le compilateur (voire figure 5.18).

3. une classe qui implémente deux interfaces dont chacune on a la définition

```
interface Recherche {

  int Nombre = 4;
}//_____________________________________________________________
class Personne {

  final static int Nombre = 1;
}//_____________________________________________________________
class ChefDépartement extends Personne implements Recherche {

  }
}
```

FIGURE 5.17 – Situation conflictuelle : Sous-Classe - Interface

```
interface Recherche {

 int Nombre = 4;
}//_____________________________________________________________
class Personne {

 final static int Nombre = 1;
}//_____________________________________________________________
class ChefDépartement extends Personne implements Recherche {

 void affiche(){
  System.out.println(Nombre); /* Situation conflictuelle */
  System.out.println(super.Nombre);
  System.out.println(Recherche.Nombre);
 }
}
```

FIGURE 5.18 – Situation conflictuelle (Exemple) : Sous-Classe - Interface

de la même constante. Alors on a ainsi un problème d'accès à la constante (la constante de quelle interface). Par exemple, si on a une classe *ChefDépartement* qui implémente les deux interfaces *Recherche* et *Responsabilité* dont chacune comporte la définition de la même constante *Nombre* (voire figure 5.19). Il est certain qu'il y a un problème d'accès à la constante *Nombre*. On considère la constante de l'interface *Recherche*, ou bien celle de *Responsabilité*. Alors, c'est pour cette raison qu'il soit rejeté par le compilateur (voire figure 5.20).

5.4.2 Méthodes abstraites

En d'hors parfaitement de toutes relation d'héritage, les interfaces imposent des contenus communs à certaines classes qui les implémentes. Le contenu imposé est précisément un certain nombre de méthodes abstraites. Puisque, une interface ne supporte jamais de constructeurs, alors il est illicite de créer une instance de type interface. Tandis que, il est tout à fait possible d'affecter à une variable de type interface une des instances des classes qui l'implémentent. En *Java*, une méthode abstraite d'une interface est définie sans le mot clé *abstract* (voir figure 5.21), puisque

```
interface Recherche {

 int Nombre = 4;
}//_____________________________________________________________
interface Responsable {

 int Nombre = 7;
}//_____________________________________________________________
class ChefDépartement implements Recherche,  Responsable {

 }
}
```

FIGURE 5.19 – Situation conflictuelle : Classe - Plusieurs Interfaces

```
interface Recherche {

 int Nombre = 4;
}//_____________________________________________________________
interface Responsable {

 int Nombre = 7;
}//_____________________________________________________________
class ChefDépartement implements Recherche,  Responsable {

 void affiche(){
  System.out.println(Nombre); /* Situation conflictuelle */
  System.out.println(Recherche.Nombre);
  System.out.println(Responsable.Nombre);
 }
 }
}
```

FIGURE 5.20 – Situation conflictuelle (Exemple) : Classe - Plusieurs Interfaces

les méthodes sans corps sont nécessairement abstraites et nécessairement publiques, et c'est pour cette raison que le langage *Java* nous dispense de le mentionner explicitement. Les interfaces servent d'exiger à certaines classes de redéfinir ses méthodes abstraites sans nécessairement avoir recours à la notion de classe et classe abstraite. Contrairement, aux constantes, les méthodes abstraites ne peuvent être jamais d'une source d'ambiguïté, puisqu'elles n'ont pas de corps (instructions). Si on a une méthode abstraite impliquée par une certaines interfaces, alors une classe peut les

```
interface Recherche {

 int afficheNombrePapiersRédigés();
}
```

FIGURE 5.21 – La méthode abstraite *afficheNombrePapiersRédigés*

implémente avec aucun problème. En effet, si on souhaite qu'elle soit instanciable, il suffit de définir dans telle classe la méthode en question quelque soit l'interface dont laquelle la méthode est imposée. Par exemple, si on a la méthode abstraite *étatAvancement* qui est impliquée par les deux interfaces *Recherche* et *Responsabilité*, alors la classe *ChefDépartement* peut parfaitement implémenter ces deux interfaces sans que cela ne fasse réagir le compilateur (voir figure 5.22). En outre, si la classe *ChefDépartement* offre une définition pour la méthode abstraite *étatAvancement*, quelque soit l'interface dont laquelle la méthode abstraite soit déclarée, alors elle peut être instanciée (voir figure 5.23).

5.4.3 Méthodes par défaut

Comme il est esquissé dans la section 5.4, le concept d'interface a fait l'objet depuis *Java*8 des enrichissements très importants. L'objectif de cette section est de présenter parmi ces enrichissements la notion de définition par défaut pour des méthodes dans une interface. On a déjà apprécié que, dans une interface on peut définir des constantes, ainsi que des méthodes abstraites. Les versions poste *Java*7 disposent d'une nouveauté certainement remarquable pour des interfaces afin de

```
interface Recherche {

 void étatAvancement();
}//-------------------------------------------------------------
interface Responsable {

 void étatAvancement();
}//-------------------------------------------------------------
class ChefDépartement implements Recherche,  Responsable {

}
```

FIGURE 5.22 – La méthode abstraite *étatAvancement*

```
interface Recherche {

 void étatAvancement();
}//_____________________________________________________________
interface Responsable {

 void étatAvancement();
}//_____________________________________________________________
class ChefDépartement implements Recherche,  Responsable {

 void étatAvancement(){
    System.out.println(" Nous avons rédigé cinq papiers, et
   nous sommes en trains de les soumissionnée pour des
   journaux internationaux.");
 }
}
```

FIGURE 5.23 – Redéfinition de la méthode abstraite *étatAvancement*

supporter la notion de corps avec une définition par défaut. Pour bien adresser la notion de méthode d'interface avec une définition par défaut, on va reprendre notre exemple de gestion de certaines personnes d'une université. En programmation OO, on peut immédiatement conçue certaines classes pour pouvoir définir des méthodes liées à la gestion de contenues des attributs et de leurs objets. Si on suppose par exemple que seuls les enseignants soient capables à publier des papiers. Dans ce cas, il ne fait pas de sens de définir des méthodes liées au fait de publication de papier dans la super-classe *Personne*. Pareillement, il ne ferait pas sens de mettre dans la super-classe *Personne* une méthode pour la gestion des réunions, puisqu'un enseignant qui n'est pas un responsable ne soit pas impliqué par un processus d'organisation des réunions. Ainsi, il ne fait pas de sens de lui offre par héritage une méthode pour la gestion des réunions. Plutôt, il est très évident de mettre des méthodes en questions dans des interfaces. On définit alors, dans l'interface *Recherche* des méthodes typiquement liées à la rédaction et à la publication des papiers, et

on spécifie dans l'interface *Responsabilité* des méthodes liées à l'organisation et à la gestion des réunions. Dans l'interface *Recherche*, on peut mettre par exemple, la méthode *nombrePapiersRédigés* pour fournir le nombre de papiers rédigés, et la méthode *nombrePapiersPubliés* pour offrir le nombre de papiers publiés (voir figure 5.24). Alors, on a la déclaration de deux méthodes abstraites. Par conséquence, toute classe (qui implémente l'interface *Recherche* et que l'on souhaite instanciable) faudra impérativement fournir une définition concrète pour ces deux méthodes (voir figure 5.25). Si on suppose qu'une personne qui fait de la recherche publie ordinairement deux papiers comme une contribution scientifique annuelle. Depuis *Java*8, il devient désormais possible de spécifier ce genre de comportement usuel dans une interface en fournissant une définition par défaut pour une méthode de rédaction. Concrètement, la méthode *nombrePapiersRédigés* peut désormais avoir un corps. Ce corps sert naturellement de fournir et d'afficher le nombre de papiers publiés par an. Toute méthode d'interface avec une définition concrète nécessitera impérativement un ajout du modificateur $default$. Pareillement à la définition d'une méthode usuelle (avec un corps) dans une classe, une méthode d'interface avec une définition par défaut est définit naturellement avec une entête et un corps, mais en précédant son entête cette fois-ci avec le modificateur $default$ (voir figure 5.26). Dans ce qui suit, on va introduit les différentes règles d'utilisation des méthodes d'interface avec une définition par défaut. En d'autres termes, on va présenter le droit de redéfinition et la

```
interface Recherche {

 int nombrePapiersRédigés();
 int nombrePapiersPubliés();
}
```

FIGURE 5.24 – La déclaration des méthodes abstraites dans l'interface *Recherche*

```
interface Recherche {

 int nombrePapiersRédigés();
 int nombrePapiersPubliés();
}//-------------------------------------------------------------
class Enseignant implements Recherche {

 int nombrePapiersRédigés() {
 return 4;
 }

 int nombrePapiersPubliés() {
 return 3;
 }
}
```

FIGURE 5.25 – La redéfinition des méthodes abstraites dans la classe *Enseignant*

```
interface Recherche {

 default int nombrePapiersRédigés()  {
  System.out.println("Nous publions deux papiers par an.
   Veillez contacter le chef de Labo, si vous voulez faire un
   coup d'oille. Merci " );
  return 2;
 }
}
```

FIGURE 5.26 – La déclaration d'une méthode par défaut dans l'interface *Recherche*

manière d'utilisation de telles méthodes d'interfaces dans des classes implémentant ces interfaces. En outre, on va illustrer la gestion des ambiguïtés possibles relative à ce genre de définition par défaut. Par exemple, si on suppose qu'on a dans une classe qui implémente une interface une redéfinition (pour un autre corps potentiel) d'une méthode (de cette interface) avec une définition par défaut, on aura alors une situation conflictuelle. Pareillement, si on suppose cette fois-ci qu'on a (dans deux

interfaces) deux méthodes de même en-tête avec des définitions par défauts, alors on aura potentiellement aussi une autre situation conflictuelle lorsqu'on souhaite dans une classe qui implémente ces deux interfaces d'utiliser ou de redéfinir telles méthodes. Alors pour définir la manière d'utilisation et de redéfinition de telles méthodes, et également pour résoudre ce genre de situations conflictuelles, les versions poste *Java*7 proposent quatre règles fondamentales. Les deux premières règles déterminent la manière d'écriture et d'utilisation des méthodes d'interface avec définition par défaut. Les deux dernières règles régissent des situations de conflits potentiels.

La règle 1 On ne doit plus fournir dans une classe qui implémente une interface des redéfinitions concrètes pour des méthodes (de cette interface) ayant des définitions par défaut. Par exemple, si on suppose qu'on a dans l'interface *Recherche*, qui est implémenté par la classe *Enseignant*, deux méthodes avec des définitions par défauts *nombrePapiersPubliés* et *nombrePapiersRédigés* (voir figure 5.27). Dans ce cas, la classe *Enseignant* est instanciable en l'état, et on n'est pas dans l'obligation de redéfinition de ces deux méthodes. Il

```
interface Recherche {

  default int nombrePapiersRédigés(){
    return 4;
  }

  default int nombrePapiersPubliés(){
    return 3;
  }
}//-------------------------------------------------------------
class Enseignant implements Recherche {

}
```

FIGURE 5.27 – La redéfinition de méthodes par défaut est facultatif

s'agit, en fait, de la principale raison du concept de méthode d'interface avec définition par défaut.

La règle 2 les méthodes d'interfaces s'héritent. Par exemple, si on considère dans l'interface *Recherche* la méthode abstraite *nombrePapiersPubliés*, ainsi que la méthode *nombrePapiersRédigés* avec définition par défaut qui retourne 4 pour indiquer le nombre annuel des papiers rédigés par un enseignant. Rien on n'empêche d'étendre l'interface *Recherche* au travers de l'interface *ContributionScientifique* sans explicitement redéfinir les méthodes de l'interface (voir figure 5.28). Puisque, si un enseignant a pour rédaction par défaut de 4 papiers par une année, alors on peut parfaitement se contenter de la définition héritée de la super interface *Recherche*. Par conséquence, on peut convenablement garder la définition héritée de plus haut. En outre, dans la sous interface *ContributionScientifique* (voir figure 5.29), il est tout a fait possible d'envisagé une définition par défaut pour la méthode abstraite *nombrePapiersPubliés* héritée de plus haut depuis l'interface *Recherche*. Dans ce

```
interface Recherche {

 int nombrePapiersPubliés();
 default int nombrePapiersRédigés() {
 return 4;
 }
}//_____________________________________________________________
interface ContributionScientifique extends Recherche {

}//_____________________________________________________________
class Enseignant implements ContributionScientifique {

}
```

FIGURE 5.28 – Les méthodes d'interfaces s'héritent

cas, un enseignant aura une définition concrète de la méthode *nombrePapiersPubliés*, ce qui n'est pas le cas dans la super interface *Recherche*. Alors, toute classe implémente l'interface *ContributionScientifique* dispose systématiquement la méthode par défaut *nombrePapiersRédigés* qui retourne 4 (le nombre de papiers rédigés), et la méthode *nombrePapiersPubliés* qui a cette fois-ci un comportement par défaut qui retourne 3 (le nombre de papiers publiés). Évidemment, il n'était pas nécessaire de redéfinir la méthode *nombrePapiersRédigés* dans l'interface *ContributionScientifique* si on est satisfait de la définition par défaut héritée de plus haut. Par contre, si on n'est pas satisfait, rien on n'empêche de la redéfinir à nouveau exactement comme on le fait pour les classes (voir figure 5.30). En d'autres termes, on peut envisagé une conception alternative pour l'interface *ContributionScientifique* en redéfinissant non seulement la méthode abstraite *nombrePapiersPubliés*, mais également la méthode par défaut *nombrePapiersRédigés* où en considérant un comportement

```
interface Recherche {

 int nombrePapiersPubliés();
 default int nombrePapiersRédigés(){
 return 4;
 }
}//_____________________________________________________________
interface ContributionScientifique extends Recherche {

 default int nombrePapiersPubliés(){
 return 3;
 }
}//_____________________________________________________________
class Enseignant implements ContributionScientifique {

}
```

FIGURE 5.29 – La redéfinition de méthodes par défaut dans des interfaces

alternatif (pour indiquer comme un comportement par défaut, par exemple, qu'un enseignant chercheur a pouvoir rédiger annuellement 5 papiers.

La règle 3 une méthode de classe est prioritaire par rapport à une méthode (d'interface) par défaut. Par exemple, si on considère la méthode par défaut $affiche$ défini dans la sous classe *Enseignant* qui implémente l'interface *Recherche*, dans laquelle on a aussi une définition alternative pour la méthode $affiche$. En outre, la sous classe *Enseignant* dispose déjà, par héritage depuis la superclasse *Personne*, une autre variante pour la méthode $affiche$ (voir figure 5.31). On est alors dans une situation conflictuelle un peu particulière, où on a une conception dans laquelle la méthode $affiche$ générale est définit immédiatement dans la super classe *Personne*. Dans ce cas, on a une ambiguïté entre la méthode $affiche$ générale hérité de la super classe *Personne*,

```
interface Recherche {

 int nombrePapiersPubliés();
 default int nombrePapiersRédigés() {
 return 4;
 }
}//-------------------------------------------------------------
interface ContributionScientifique extends Recherche {

 default int nombrePapiersPubliés(){
 return 3;
 }//
   -------------------------------------------------------------
 default int nombrePapiersRédigés() {
 return 5;
 }
}//-------------------------------------------------------------
class Enseignant implements ContributionScientifique {
}
```

FIGURE 5.30 – La redéfinition de méthodes d'interface dans des interfaces

et celle du même identificateur de l'interface *Recherche*. Alors, on affiche le nom et le prénom d'un enseignant au travers de la méthode $affiche$ hérité de la super-classe *Personne*, ou bien on affiche le nombre de papiers rédigé annuellement par enseignant au moyen de la méthode par défaut $affiche$ de l'interface. Selon la règle 3 qui est adoptée par le langage *Java*, si on déclare une instance de type *Enseignant*, et lors de l'invocation de la méthode $affiche$ sur cette instance, alors la méthode défini dans la classe qui sera adopté puisqu'elle a la priorité de précédence. Dans ce cas, c'est le nom et le prénom d'un enseignant qui serrant affichés, et il ne soit pas le nombre de papiers rédigés. Tandis que, si l'on souhaite qu'il soit plutôt d'adopter celle de l'interface, il est tout à fait possible de le spécifier. Pour ce faire, on doit

```
interface Recherche {

 default void affiche() {
  System.out.println("Le nombre de papiers publiés est 4");
 }
}//_____________________________________________________________
class Personne {

 String Nom, Prénom;
 Personne(String nom, String prénom){
 Nom=nom;
 Prénom=Prénom;
 }
 void affiche() {
 System.out.println("Le nom de la personne est : "+ Nom + ",
   et le prénom est : "+ Prénom );
 }
}//_____________________________________________________________
class Enseignant extends Personne implements Recherche {

}
```

FIGURE 5.31 – Les méthodes de classe ont la priorité de précédence

redéfinir telle méthode dans la classe en question de telle sorte on favorise l'adoption explicite de celle de l'interface au moyen de la syntaxe *IdentificateurInterface.super.IdentificateurMéthodeParDéfaut*. Dans ce cas, on doit redéfinie la méthode $affiche$ dans la classe *Enseignant* en spécifiant explicitement l'appelle de la méthode par défaut $affiche$ de l'interface *Recherche* selon la syntaxe précédente (voir figure 5.32). En outre, l'utilisation du mot clé *super* permet de lever l'ambigüité d'invocation entre une méthode d'interface avec une définition par défaut, et une méthode d'interface statique. Par exemple si on a dans les deux interfaces *Responsable* et *Recherche* une définition par défaut pour la méthode $affiche$ avec bien sure le même entête. Alors, si on veut

```
interface Recherche {

 default void affiche() {
  System.out.println("Le nombre de papiers publiés est 4");
 }
}//_____________________________________________________________
class Personne {
 String Nom, Prénom;
 Personne(String nom, String prénom){
 Nom=nom;
 Prénom=Prénom;
 }
 void affiche() {
 System.out.println("Le Nom de la personne est : "+ Nom + ",
   et le Prénom est : "+ Prénom );
 }
}//_____________________________________________________________
class Enseignant extends Personne implements Recherche {
 void affiche(){
 Recherche.super.affiche();
 }
}
```

FIGURE 5.32 – Situation conflictuelle : SuperClasse - Interface

utiliser dans la classe *Enseignant* qui implémente ces deux interfaces la méthode $affiche$ sur une instance de la classe *Enseignant*, on aura une situation de conflit. Selon la règle adoptée par le langage *Java*, afin de résoudre cette situation conflictuelle, les classes qui implémentent les interfaces conflictuelles sont en charge de lever l'ambiguïté au travers de la notion de redéfinition de telles méthodes. Dans ce cas, la classe *Enseignant* qui implémente les deux interfaces conflictuelles *Responsable* et *Recherche* doit redéfinir la méthode $affiche$ en spécifiant la méthode (d'interface) par défaut à utiliser. Pour notre exemple, on a choisit la définition par défaut issue de l'interface *Recherche*. En d'autres termes, la méthode $affiche$ dans la classe *Enseignant* est redéfinie de façon à invoquer la méthode $affiche$ telle que définie par défaut dans l'interface *Recherche* (voir figure 5.33). Cependant, rien on n'empêche de faire le choix alternatif de la méthode $affiche$ issue de *Responsable*. Alors comme

```
interface Recherche {

 default int nombrePapiersPubliés() {
 return 4;
 }
}//_____________________________________________________________
interface Responsabilité {

 default int nombrePapiersPubliés() {
 return 7;
 }
}//_____________________________________________________________
class Enseignant implements Recherche, Responsabilité {

 int nombrePapiersPubliés() {
 return Recherche.super.nombrePapiersPubliés();
 }
}
```

FIGURE 5.33 – Situation conflictuelle (Exemple 1) : Classe - Interfaces

une nouvelle variante, la méthode est redéfinie dans la classe *Enseignant* de sorte à utiliser la méthode $affiche$ telle que définie par défaut dans l'interface *Responsable* (voir figure 5.34). En outre, d'autres implémentations sont évidemment possibles. Par exemple, rien on n'empêche la redéfinition dans la classe *Enseignant*, la méthode $affiche$ de sorte à invoquer les deux méthodes (des interfaces) par défauts. Lors qu'on souhaite d'afficher la personne ainsi que, le nombre annuel de papiers rédigés par lui, on pout envisager une définition de la méthode $affiche$ en faisant appel aux méthodes issue des interfaces *Recherche* et *Responsabilité* (voir figure 5.35).

La règle 4 les interfaces servent strictement à la modélisation des aspects comportementaux certainement indépendamment de tout état. Par exemple, si on suppose que l'on souhaite de savoir la liste, au lieu du nombre, de papiers rédigés par un enseignant au niveau de l'interface *Recherche*. Alors, on a besoin

```
interface Recherche {

 default int nombrePapiersPubliés() {
 return 4;
 }
}//-------------------------------------------------------------
interface Responsabilité {

 default int nombrePapiersPubliés() {
 return 7;
 }
}//-------------------------------------------------------------
class Enseignant implements Recherche, Responsabilité {

 int nombrePapiersPubliés() {
 return Responsabilité.super.nombrePapiersPubliés();
 }
}
```

FIGURE 5.34 – Situation conflictuelle (Exemple 2) : Classe - Interfaces

```
interface Recherche {
  default int nombrePapiersPubliés(){
    return 4;
  };
}
interface Responsabilité {
  default int nombrePapiersPubliés(){
  return 7;
  }
}
class Enseignant implements Recherche, Responsabilité {
  int nombrePapiersPubliés(){
  Recherche.super.nombrePapiersPubliés();
    Responsabilité.super.nombrePapiersPubliés();
  };
}
```

FIGURE 5.35 – Situation conflictuelle (Exemple 3) : Classe - Interfaces

de définir un attribut de type liste (tableau) dont les objets sont des papiers dans l'interface *Recherche*. Il impossible dans une interface de spécifier de tel état, puisque le fait de pouvoir déclarer un attribut, alors on a systématiquement spécifié un état (un objet du papier dépendamment de la personne qu'il rédige). Les interfaces ne peuvent contenir jamais d'attributs sauf que les constantes, parce qu'elles n'y ont pas de constructeurs qui ont pour vocation d'initialiser des attributs d'instances. Alors, l'adoption des interfaces est exactement privilégiée lors qu'on souhaite modéliser un aspect comportemental (fonctionnel) indépendant de toute instance. Selon le point de vue conception, le lien d'implémentation des interfaces (*implements*) est en effet plus flexible (et moins contraignant) que celui de l'héritage (*extends*). En fait, l'objectif primordiale d'admission des méthodes d'interfaces avec définition par défaut est de ne pas largement mimiquer l'héritage multiple en java, et également d'engendrer une conception alternative au moyen de classes abstraites, mais

plutôt de favoriser l'enrichissement des interfaces de nouvelles méthodes sans aucune pénalisation de l'existant, et également sans aucun influence sur des classes implémentant telles interfaces.

5.4.4 Méthodes statiques

Dans une interface, il est également possible de définir une certaines méthodes statiques. La syntaxe de définition d'une méthode statique pour une interface est constamment similaire à celle qu'on a adopté pour définir des méthodes statiques dans une classe (voir la section 4.3.2). Dans une méthode d'une classe qui implémente une interface, Si on veut appeler une méthode statique d'une interface, alors on adopte la syntaxe d'appelle d'une méthode statique de classe. En d'autres termes, pour invoquer une méthode statique d'une interface, on utilise l'identificateur de l'interface suivi de celui de la méthode statique de cette interface. Par contre, pour une méthode d'interface avec définition par défaut, on doit adopter en plus le mot clé *super*. Par exemple, dans la méthode *nombrePapiers* de la classe *Enseignant* qui implémente l'interface *Recherche*, on a adopté mot clé *super* pour appeler la méthode (l'interface *Recherche*) avec définition par défaut *nombrePapiersPubliés* (*Recherche.super.nombrePapiersPubliés*). Tandis qu'on a carrément utilisé l'identificateur de l'interface suivi directement de celui de la méthode statique (*Recherche.nombrePapiersRédigés*) pour invoquer la méthode statique *nombrePapiersRédigés* (voir figure 5.36). Cependant, il faut noter en toute rigueur que, il est illicite d'avoir deux méthodes différentes (l'une est statique, et l'autre a une définition par défaut) avec un même identificateur dans une même interface. Il n'est pas toléré par le compilateur *Java* pour un même entête de définir à la fois une méthode d'interface statique et en même temps une méthode d'interface avec une définition par défaut. En revanche, la raison principale d'introduction des méthodes d'interfaces statiques est certainement pour supporter parfaitement la flexibilité d'enrichisse-

```
interface Recherche {

 static int nombrePapiersRédigés () {
 return 4;
 }//
   ------------------------------------------------------------
 default int nombrePapiersPubliés() {
 return 3;
 }
}//------------------------------------------------------------
class Enseignant implements Recherche {
 int nombrePapiers(){
 Recherche.nombrePapiersRédigés();
 Recherche.super.nombrePapiersPubliés();
 }
}
```

FIGURE 5.36 – Utilisation des méthodes statiques

ment des interfaces de nouvelles méthodes en protégeant des classes implémentant telles interfaces.

Bibliographie

[Barry et Daniel, 2001] Barry J. Holmes, Daniel T. Joyce, Object-Oriented Programming With Java, Second Edition, ISBN 0-7637-1435-6, Jones And Bartlett ;

[Danny et al, 2008] Danny Poo, Derek Kiong, Swarnalatha Ashok, Object-Oriented Programming and Java, Second Edition, ISBN 13 : 978-1-84628-962-0 , Springer-Verlag.

[Jamila et Jean-Cédric, 2014] Jamila Sam, Jean-Cédric Chappelier. Introduction à la programmation orientée objet (en Java). École Polytechnique Fédérale de Lausanne. CoursEra Accéder en 2014.

[Jose, 2003] Jose M. Garrido, Object-Oriented Programming : From Problem Solving to Java, ISBN 1584502878, Charles River Media.

[Michel, 2006] Michel Divay, La Programmation Objet En Java, ISBN 2 10 049697 2, Dunod.

[Richard, 2001] Richard Wiener, Fundamentals of OOP and Data Structures in Java, ISBN 0 -521-66220 -6, Cambridge University Press (Virtual Publishing).

[Robert, 2002] Robert Lafore, Object-Oriented Programming in C++, Fourth Edition, ISBN 0-672-32308-7, Sams.