Support de cours

# Programmation orientée objet

(En JAVA)

2e Année Licence Informatique

Assia Brighen @ univ-Jijel

Université Mohammed Seddik Benyahia, Jijel
Faculté des sciences exactes et informatique
Département d'Informatique


a.brighen@univ-jijel.dz

*Mai 2024*

## Préface

L'orienté objet est un paradigme très puissant pour modéliser des phénomènes du monde réel dans un modèle de calcul. Ce paradigme est naturel et intuitif, car il permet de réfléchir en termes d'objets. Mais, il ne signifie pas seulement l'utilisation des objets dans un programme. L'abstraction, l'encapsulation, le polymorphisme et l'héritage sont quelques-unes des caractéristiques importantes du paradigme orienté objet.

Pour ne pas citer que quelques-uns, C++, Java et C# (prononcé "C Sharp") sont des langages de programmation qui prennent en charge le paradigme orienté objet. Java est l'un des langages de programmation les plus utilisés pour le développement de divers types de logiciels tels que des applications de bureau, d'entreprise, Web et mobiles.

L'objectif de ce cours est l'introduction des concepts de base de la programmation orientée objet (POO) par la pratique en utilisant le langage Java. Chaque chapitre comporte certaines notions qui sont traduite à sa fin en Java pour que l'étudiant puisse traduire les concepts théoriques acquis en pratique.

Connaissances préalables : Langage C, Structure de données et algorithmes

# Table des matières

# 1. POO & langages de programmation

Sommaire



## 1.1 Introduction

La programmation consiste à fournir une solution à un problème du monde réel à l'aide de modèles informatiques pris en charge par les langages de programmation. Cette solution s'appelle un programme. Avant de proposer une solution à un problème sous la forme d'un programme, on a toujours une vision mentale du problème et de sa solution. Cependant, les perceptions du problème et de sa solution ne sont pas les mêmes. La manière dont les gents regardent la réalité constitue leur paradigme. Un paradigme est considéré comme un état d'esprit avec lequel une réalité est perçue dans un contexte particulier. Il est courant d'avoir plusieurs paradigmes, qui permettent de voir différemment la même réalité.

Dans le monde informatique, il ne suffit pas de trouver une solution à un problème. La solution doit être pratique et efficace. Puisque la solution à un problème est

toujours liée à la façon dont le problème et la solution sont pensés, le paradigme devient primordial. Le paradigme de programmation entre en scène bien avant l'écriture d'un programme à l'aide d'un langage de programmation. C'est dans la phase d'analyse qu'on utilise un paradigme particulier pour analyser un problème et sa solution d'une manière particulière. Un langage de programmation fournit un moyen de la mise en œuvre de manière appropriée un paradigme de programmation particulier.

De nombreux paradigmes sont utilisés en programmation, tels que : paradigme impératif, orienté objet, procédural, etc. Chaque paradigme offre sa manière de résolution de problème, comme il offre aussi des manières de la modularité et la réutilisation de codes. Dans ce chapitre, nous allons examiner quelques paradigmes de programmation y compris le paradigme orienté objet qui fait l'objet de ce cours.

## 1.2 Langages de programmation

Les langages de programmation sont généralement classés en trois niveaux : les langages machine, les langages d'assemblage et les langages de haut niveau. Le langage machine est le seul langage de programmation compris par le CPU. Chaque type de CPU a sa propre langage machine. Les instructions en langage machine sont codées en binaire et de très bas niveau, par exemple, une instruction machine peut transférer le contenu d'un emplacement mémoire dans un registre CPU ou ajouter des nombres dans deux registres. Ainsi, nous devons fournir de nombreuses instructions en langage machine pour réaliser une tâche simple, par exemple "trouver la moyenne de quelques nombres".

Un niveau au-dessus du langage machine est le langage assembleur, qui permet une programmation symbolique "de niveau supérieur". Au lieu d'écrire des programmes sous la forme d'une séquence de bits, le langage d'assemblage permet aux programmeurs d'écrire des programmes en utilisant des codes d'opération symboliques. Par exemple, au lieu d'écrire *10110011*, on utilise *MV* pour déplacer le contenu d'une cellule mémoire dans un registre. On peut également utiliser des noms symboliques ou mnémoniques pour les registres et les cellules de mémoire.

Étant donné que les programmes écrits en langage assemblage ne sont pas reconnus par le CPU, un assembleur est utilisé pour traduire les programmes écrits en langage assemblage en équivalents en langage machine. Par rapport à l'écriture de programmes en langage machine, l'écriture de programmes en langage assemblage est beaucoup plus rapide, mais pas assez rapide pour écrire des programmes complexes.

Des langages de haut niveau ont été développés pour permettre aux programmeurs d'écrire des programmes plus rapidement qu'en utilisant des langages d'assemblage.

Par exemple : FORTRAN et COBOL, qui ont été développés à la fin des années 1950 et au début des années 1960 et sont toujours utilisés. BASIC (Beginners All-purpose Symbolic Instructional Code) était le premier langage de haut niveau disponible pour les micro-ordinateurs. D'autres célèbres langages de haut niveau sont : Pascal, C, C++, et Java. Le langage Pascal a été conçu comme un langage académique. Le langage de programmation C a été développé au début des années 1970 chez AT&T Bell Labs. Le langage de programmation C++ a été développé comme successeur de C au début des années 1980 pour ajouter la prise en charge de la programmation orientée objet. La programmation orientée objet est un style de programmation de plus en plus accepté aujourd'hui.

Étant donné que les programmes écrits dans un langage de haut niveau ne sont pas reconnus par le processeur, on doit utiliser un compilateur pour les traduire en équivalents en langage d'assemblage ou en langage machine.

### 1.2.1 Paradigmes de programmation

Le mot « paradigme » est défini comme suit :

**Définition 1.1** Un paradigme est une théorie ou un groupe d'idées sur la façon dont quelque chose doit être fait, fabriqué ou pensé.

La programmation consiste à fournir une solution à un problème du monde réel à l'aide de modèles de calcul pris en charge par le langage de programmation. Dans ce contexte, un paradigme de programmation est définit comme suit :

**Définition 1.2** Un paradigme de programmation est une façon de conceptualiser ce que signifie effectuer un calcul et comment les tâches qui doivent être effectuées sur un ordinateur doivent être structurées et organisées.

Le paradigme de programmation est la façon de penser et conceptualiser le problème du monde réel et sa solution dans les modèles de calcul sous-jacents. Un langage de programmation peut fournir des fonctionnalités qui le rendent adapté à la programmation en utilisant un paradigme de programmation et pas l'autre.

Un programme comporte deux composants : des données et un algorithme. Les données sont utilisées pour représenter des éléments d'information. Un algorithme est un ensemble d'étapes qui opèrent sur des données pour arriver à une solution d'un problème. Différents paradigmes de programmation impliquent d'avoir la solution d'un problème en combinant des données et des algorithmes de différentes manières. Voici quelques paradigmes de programmation couramment utilisés :

**Paradigme impératif :** Le paradigme impératif est également appelé paradigme algorithmique. Dans le paradigme impératif, un programme se compose de données et d'un algorithme (séquence de commandes) qui manipule les données. Les données à un instant donné définissent l'état du programme. L'état du programme change au fur et à mesure que les commandes sont exécutées dans une séquence spécifique. FORTRAN, COBOL et C sont quelques exemples de langages de programmation qui prennent en charge le paradigme impératif.

**Paradigme procédural :** Le paradigme procédural est similaire au paradigme impératif avec une différence, il combine plusieurs commandes dans une unité appelée procédure. Un programme dans un langage procédural se compose de données et d'une séquence d'appels de procédure qui manipulent les données. L'utilisation de procédures se traduit par un code modulaire et augmente la réutilisabilité des algorithmes. Même s'ils sont différents, le paradigme procédural implique toujours le paradigme impératif. Dans le paradigme procédural, l'unité de programmation n'est pas une séquence de commandes, mais une séquence de commandes dans une procédure, et le programme consiste, plutôt, en une séquence de procédures. C, C++, Java et COBOL sont quelques exemples de langages de programmation prenant en charge le paradigme procédural.

**Paradigme déclaratif :** dans ce paradigme, un programme consiste en la description d'un problème, et l'ordinateur trouve la solution. Le programme ne précise pas comment arriver à la solution au problème. Dans le paradigme impératif, on s'intéresse à la partie « comment » résoudre le problème. Dans le paradigme déclaratif, on s'intéresse à la partie c'est « quoi » le problème plutôt que de savoir comment le résoudre. L'écriture d'une requête de base de données SQL est considérée comme un exemple de la programmation déclaratif, où le programmeur spécifie les données voulus et le moteur de base de données détermine comment récupérer les données demandés. Contrairement au paradigme impératif, les données sont permanentes et l'algorithme est transitoire dans le paradigme déclaratif.

**Paradigme fonctionnel :** ce paradigme est basé sur le concept de fonctions mathématiques. Une fonction est considéré comme un algorithme qui calcule une valeur à partir de certaines données d'entrées. Une nouvelle valeur est dérivée en appliquant une fonction à la valeur d'entrée. La valeur d'entrée ne change pas. Les langages de programmation fonctionnels n'utilisent pas les variables et les affectations, qui sont utilisées pour modifier les données. En programmation fonctionnelle, une tâche répétée (une boucle par exemple) est effectuée à l'aide de la récursivité. Haskell,

Erlang et Scala sont quelques exemples de langages de programmation prenant en charge le paradigme fonctionnel.

**Paradigme logique :** dans ce paradigme, un programme se compose d'un ensemble d'axiomes et d'un énoncé d'objectif. L'ensemble des axiomes est l'ensemble des faits et des règles d'inférence qui composent une théorie. Le programme utilise des déductions pour prouver le théorème dans la théorie. La programmation logique utilise un concept mathématique appelé relation de la théorie des ensembles. Une relation en théorie des ensembles est définie comme un sous-ensemble du produit cartésien de deux ensembles ou plus. Prolog est un exemple de langage de programmation prenant en charge le paradigme logique.

**Paradigme orienté objet :** Dans ce paradigme, un programme se compose d'objets interactifs. Un objet encapsule des données et des algorithmes. Les données définissent l'état d'un objet. Les algorithmes définissent le comportement d'un objet. Un objet communique avec d'autres objets en leur envoyant des messages. Lorsqu'un objet reçoit un message, il répond en exécutant un de ses algorithmes, ce qui peut modifier son état. Dans les paradigmes impératif et fonctionnel, les données et les algorithmes sont séparés, alors que dans le paradigme orienté objet, les données et les algorithmes ne sont pas séparés ; ils sont combinés en une seule entité, appelée un objet. Les classes sont les unités de base de la programmation dans le paradigme orienté objet. Les objets similaires sont regroupés dans une définition appelée classe. La définition d'une classe est utilisée pour créer un objet. Un objet est également connu comme une instance de la classe. Une classe se compose de variables d'instance et de méthodes. Les valeurs des variables d'instance d'un objet définissent l'état de l'objet. Une méthode est comme une procédure (ou une sous-routine) dans le paradigme procédural. Les méthodes peuvent accéder/modifier l'état de l'objet. Java, C++ et Kotlin sont des exemples de langage de programmation prenant en charge le paradigme logique.

### 1.2.2 Programmation procédurale VS programmation orientée objet

La programmation procédurale est un moyen de résoudre des problèmes informatiques en identifiant l'ensemble des étapes permettant de les résoudre et l'ordre exact d'exécution pour atteindre le résultat ou l'état souhaité. En programmation orientée objet, l'accent est mis sur la réflexion sur le problème à résoudre en termes d'éléments du monde réel et sur la représentation du problème en termes d'objets et de comportement. L'objet est une structure de données qui ressemble étroitement à un objet du monde réel.

| | Paradigme procédurale | Paradigme orienté objet |
|---|---|---|
| **Programmes** | Le programme principal est divisé en petites parties selon les procédures | Le programme principal est divisé en petits objets en fonction du problème |
| **Données** | Les attributs et les procédures sont séparés | Les attributs et les méthodes sont contenus dans un seul objet |
| **Masquage de données** | Il n'y a pas un moyen idéal pour masquer les données | Permet de restreindre l'accès à certains attributs et/ou méthodes (Masquage) |
| **Accès aux données/procédures** | Données locales, globales, passages de paramètres | Utilisation des spécifications d'accès public, private, et protected aux données et aux méthodes |
| **Communication** | Passage de paramètres | Communication via des messages |
| Résolution de problème | Ensemble des étapes dans un ordre d'exécution bien définit | Représentation du problème en termes d'objets et de comportement |

TABLE 1.1 – Procédurale VS Orienté objet

La principale différence entre la programmation orientée objet et la programmation procédurale réside dans le fait que la tâche de programmation se décompose en un ensemble de variables et de sous-routines, tandis que la programmation orientée objet permet de décomposer la tâche de programmation en objets, qui encapsulent les données et les méthodes. La Table 1.1 résume quelques différences entre ces deux paradigmes de programmation.

## 1.3 Petit historique de la POO

Bien que de nombreux programmeurs ne s'en rendent pas compte, le développement de logiciels Orientés Objet (OO) existe depuis le début des années 1960. Ce n'est qu'entre le milieu et la fin des années 1990 que le paradigme orienté objet a commencé à pénétrer le monde de la programmation, malgré le fait que les langages de programmation orientés objets populaires tels que Smalltalk et C++ étaient déjà largement utilisés.

L'orienté objet né à la fin des années soixante avec SIMULA (Simple Universal Language), un langage de programmation développé durant les années soixante-dix par Ole-Johan Dahl et Kristen Nygaard à partir d'Algol 60. Simula est souvent considéré à tort comme le premier langage orienté objet, malgré que le concept orienté objet introduit pour la première fois dans les années 1970. Le paradigme orienté objet a commencé à pénétrer le monde industriel durant les années 80, notamment avec le langage SMALLTALK, pour connaître un succès très important

à partir des années 90, avec C++ et JAVA. L'idée originelle consiste à proposer un modèle de programmation qui soit le plus proche possible du monde réel (des objets interagissent entre eux).

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés, citons comme exemples : Java, C++, C#, Python, Ruby, PHP, VB.NET,Objective-C, Kotlin, etc. Une connaissance minimale des principes de la POO est donc indispensable à tout informaticien, qu'il soit développeur ou non.

## 1.4 Réutilisation de code

En informatique le terme réutilisation de code consiste à utiliser un logiciel existant, de connaissances sur ce logiciel, ses composants logiciels ou son code source, pour créer de nouveaux logiciels sans avoir à réécrire à nouveau. Celà peut se faire par la création des fonctions qui peuvent être appelées ou invoquées dans le même programme, ou regroupées en bibliothèques qui peuvent, à son tour, être utilisées aux besoins dans des différents programmes ou API sans avoir à les réécrire.

La réutilisation de code permet donc de réduire de manière considérable le temps de développement d'un programme par le programmeur, réduction de l'effort de test du code, faciliter la tache de programmation et d'améliorer la lisibilité et la structuration du code. Comme il permet aussi de réduire le code de tel sorte qu'il devient plus court et synthétique. La réutilisation du code est disponible dans la quasi-totalité des langages de programmation. Elle est basée sur la notion de "modularité".

## 1.5 Introduction à la modularité

Le mot "module" est l'un des mots les plus employés en programmation moderne. Dans ce contexte, un module consiste à une entité de données et d'instructions qui fournissent une solution à une partie bien définie d'un problème plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules reliés doit alors être capable de résoudre le problème global.

La plupart des langages de programmation permettent de décomposer les programmes en sous-programmes, fonctions ou procédures plus simples et plus compacts. A l'aide de ces structures, les programmes peuvent être modularisés afin d'obtenir des solutions plus élégantes et plus efficaces.

Parmi les avantages de la programmation modulaire on peut citer les suivantes :

- ☆ Meilleure lisibilité du programme

- ☆ Diminution du risque d'erreurs
- ☆ Possibilité de tests sélectifs de chaque module
- ☆ Dissimulation des méthodes
- ☆ Réutilisation de modules déjà existants
- ☆ Simplicité de l'entretien et la modification du code
- ☆ Favorisation du travail en équipe par affectation de la programmation des modules à différentes personnes ou groupes de personnes.

## 1.6 Conclusion

La programmation orientée objet est utilisée pour structurer un programme en éléments de code simples et réutilisables, généralement appelés classes, qui sont utilisés pour créer des instances individuelles d'objets. Elle est un paradigme de programmation fondamental pour de nombreux langages de programmation, dont Java et C++. Ce paradigme de programmation sera détaillé dans le Chapitre 3. Les codes illustratifs et les exemples seront présentés en Java. Une introduction au langage Java fait l'objet du Chapitre 2.

# 2. Introduction à Java

Sommaire



## 2.1 Introduction

Java est l'un des langages de programmation qui support le paradigme orienté objet. Ce langage de programmation reçoit une grande attention de la part de l'industrie et du monde académique. Il est un langage de programmation populaire et est utilisé à grande échelle dans le monde entier pour le développement d'applications.

Java est un langage objet qui s'appuie sur la syntaxe du langage C et du C++. Ce

chapitre présente la syntaxe des instructions de base de Java permettant de définir des variables de types dits "primitifs" et de contrôler le déroulement du programme. Les tableaux, les chaines de caractères et les tableaux dynamiques sont abordés aussi dans ce chapitre.

## 2.2 Syntaxe de base de Java

Le Pseudo-code 2.1 présente un programme réalisant la somme de deux nombres entiers. JavaApp est une classe, qu'on peut la considérer comme un module comportant une fonction main. Les mots-clés public et static sont définis dans le chapter 3. *String [ ] args* déclare un tableau de chaînes de caractères nommé args qui contient les arguments de la ligne de commande lors du lancement du programme. Les instructions suivantes déclarent et lisent deux nombres entiers, calculent et affichent la somme de ces deux nombres.

```
1  package javaapp;
2  import java.util.Scanner;
3  //JavaApp.java: Somme de deux nombres entiers
4  public class JavaApp {
5  public static void main(String[] args) {
6      // TODO code application logic here
7      System.out.println("Premier programme Java !");
8      Scanner in = new Scanner(System.in);
9      int a,b,som;
10     System.out.print("a = ");
11     a = in.nextInt();
12     System.out.print("b = ");
13     b = in.nextInt();
14     som=a+b;
15     System.out.println("a+b = "+som);
16     }
17 }
```

Pseudo-code 2.1 – Structure d'un programme Java

## 2.3 Lecture/écriture

L'affichage de données à écran sous Java est réalisé par la fonction `System.out.print` ou `System.out.println`. Cette fonction permet d'afficher des chaines littérales entre guillemets (par exemple `System.out.println("Hello world!");`). ou des expression Java simples ou complexes (par exemple `System.out.println(a+b);`).

Concernant la lecture de données aux clavier, il faut d'abord importer la classe `Scanner`, pour celà on écrit au début du programme : `import java.util.Scanner;`.

| Type | Description | Valeurs |
|---|---|---|
| byte | un entier signé sur 8 bits | de –128 à +127 |
| short | un entier signé sur 16 bits | de –32 768 à +32 767 |
| int | un entier signé sur 32 bits | de –2147483648 à 2147483647 |
| long | un entier signé sur 64 bits | de l'ordre de (±) 9 milliards de milliards |
| float | un réel sur 32 bits | de l'ordre de 1.4E-45 à 3.4E38 |
| double | un réel sur 64 bits | de l'ordre de 4.9E-324 à 1.79E308 |
| boolean | vrai ou faux | true ou false |
| char | un caractère Unicode entier positif sur 16 bits | entre 0 et 65535 |

TABLE 2.1 – Types de bases Java

Ensuite on peut créer une variable de type `Scanner` comme suit :

```
1  Scanner input=new Scanner(System.in)
```

La variable input peut être utilisée autant de fois que nécessaire pour demander des valeurs au clavier et selon le type de données désiré on utilise la méthode appropriée, comme illustré par le Pseudo-code 2.2.

```
1  Scanner input = new Scanner(System.in);
2  //import java.util.Scanner;
3  int n=input.nextInt();
4  double d=input.nextDouble();
5  float f=input.nextFloat();
6  String s=input.nextLine();
7  char c=input.next().charAt(0);
8  //and so on ...
```

Pseudo-code 2.2 – Lecture de données au clavier

## 2.4 Types et structures de contrôle en Java

### 2.4.1 Types de données primitifs (de base)

Les types primitifs (de base) correspondent à ceux du langage C auxquels s'ajoutent les types byte et boolean. Les chaînes de caractères sont gérées (différemment du C) sous forme de classes. La Table 2.1 indique la liste des types de base Java.

R pour chaque type de données primitifs, des constantes indiquant les maximums et les minimums sont définies, par exemple : `Integer.MIN_VALUE, Integer.MAX_VALUE, Byte.MIN_VALUE, Byte.MAX_VALUE, Short.MIN_VALUE, //etc.`

| Opérateur | Signification | Exemple |
|---|---|---|
| + | addition | n=a+b ; |
| - | soustraction | n=a-b ; |
| * | multiplication | n=a*b ; |
| / | division | n=a/b |
| % | modulo | n=a%b ; |
| ++ | incrémente la variable | i++ ; |
| - - | décrémente la variable | i- - ; |
| += | - | n+=b ; ⇒ $n = n + b$; |
| -= | - | n-=b ; ⇒ $n = n - b$; |
| = | affectation | $a = b$; |

TABLE 2.2 – Opérateurs arithlétiques Java

| Opérateur | Signification | Exemple |
|---|---|---|
| > | supèrieur | $a > b$; |
| < | soustraction | $a < b$ |
| >= | supèrieur ou égale | $a >= b$ |
| <= | inférieur ou égale | $<= b$ |
| == | égale | $a == b$ |
| ! = | différent à | $a! = b$ |
| && | et logique | $a\&\&b$ |
| \|\| | ou logique | a \|\| b |
| ! | négation | $!trouve$ |

TABLE 2.3 – Opérateurs logiques Java

R En Java, une variable déclarée et non initialisée ne peut pas être utilisée et une erreur de compilation va survenir.

### 2.4.2 Opérateurs arithmétiques et logiques

La Table 2.2 et Table 2.3 , ci-dessus, résument les différents opérateurs arithmétiques et logiques Java.

### 2.4.3 Transtypage implicite et explicite (casting)

Les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais on peut écrire des expressions mixtes dans lesquelles interviennent des opérandes de types différents. Les conversions de types implicites (automatiques) autorisés en Java avec des exemples sont illustrées par le Pseudo-code 2.3. En Java, on peut affecter un *byte* à un *short*, ou un *float* à un *double*.

```
1 //converting a smaller type to a larger type size
2 byte -> short -> char -> int -> long -> float -> double
3 //example:
```

```
4  int n = 1, p = 5;
5  float x = 1.5f;
6  float b= n * x + p; // Automatic casting: float to double
7  long a=n;           // Automatic casting: int to long
8  double d =n+x;      // Automatic casting: int to double
9  char c='a';
10 int e=c;            // Automatic casting: char to int
```

Pseudo-code 2.3 – Conversion automatique de types en Java

Par contre, l'inverse n'est pas autorisé et il faut explicitement forcer la conversion de types. Pour forcer la conversion d'une expression quelconque dans un type de son choix, on utilise un opérateur un peu particulier nommé *cast*. Sa dénotation consiste à placer entre parenthèses, devant la valeur à convertir, le type dans lequel elle doit être converti. Les conversions de types explicites (manuelles) en Java avec des exemples sont illustrées par Pseudo-code 2.4.

```
1  // converting a larger type to a smaller size type
2  double -> float -> long -> int -> char -> short -> byte
3
4  //Example:
5  double d = 9.78d;
6  int myInt = (int) d; // Manual casting: double to int
7  float f=3;
8  f=(float)(d+3);      // Manual casting: double to float
9  n= (int) f ;         // Manual casting: float to int
10 char c='a';
11 c=(char) e;          // Manual casting: int to char
```

Pseudo-code 2.4 – Conversion manuelle de types en Java

R Un *casting* entre types primitifs peut générer une perte de données sans aucun avertissement ni message d'erreur.

### 2.4.4 Constante en Java

En Java, les constantes sont déclarées à l'aide du mot clés `final`. Le mot-clé `final` permet de déclarer que la valeur d'une variable (attribut) ne doit pas être modifiée pendant l'exécution du programme. En Java, on est pas obligé d'initialiser une variable déclarée finale lors de sa déclaration. En effet, Java demande simplement qu'une variable déclarée finale ne reçoive qu'une seule fois une valeur. Cette affectation doit être effectuée, au plus tard, dans le constructeur de la classe.

R Si la variable finale est un objet, c'est la référence vers l'objet qui devient constante et non sa valeur. Ceci s'applique également aux tableaux qui sont aussi des références.

**Exemple 1** La Figure 2.1 montre un exemple d'un tableau $z$ déclaré final avec une tentative de modification du tableau et ses valeurs dans la fonction $A$. Une erreur est apparue lors la modification de $z$. Par contre, la modification des éléments du tableau ne signale aucune erreur.

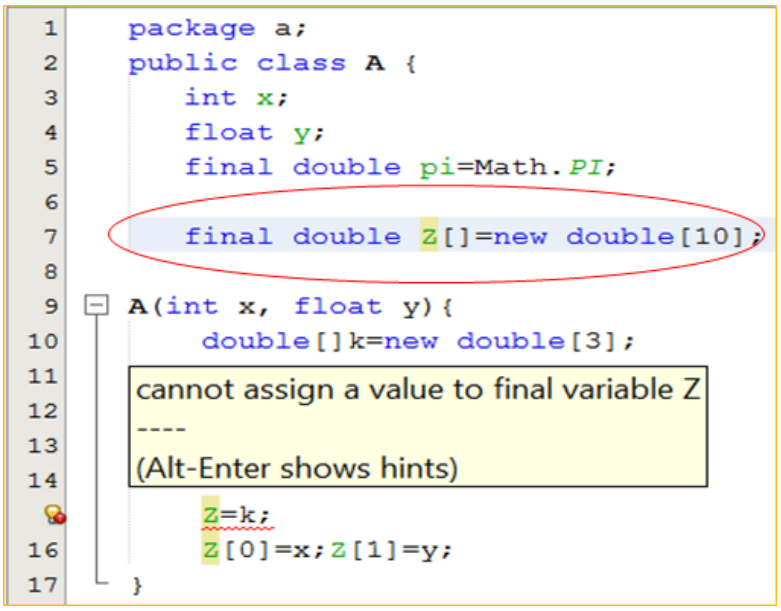



FIGURE 2.1 – Exemple d'un tableau déclaré final.

### 2.4.5 Structures de contrôles

Les instructions de contrôle du déroulement lors de l'exécution des instructions ont la même syntaxe que pour le langage C. La syntaxe des alternatives et la syntaxe des boucles sont présentées, respectivement, par Pseudo-code 2.5 et Pseudo-code 2.6.

```
1  //if without else
2  if(test) //test is a logic expression
3  {  block_of_instructions }
4
5  //if with one instruction
6  if(test) one_intruction;
7  //if with else
8  if(test) {  block_of_instructions }
9  else
10 {  block_of_instructions }
11 //if with else if
12 if(test)
13 {  block_of_instructions  }
14 else if {  block_of_instructions }
```

Pseudo-code 2.5 – Les alternatives en Java

```
//for with one instruction
for(int i=0;i<n;i++) one_intruction;

//for with several instructions
for(int i=0;i<n;i++)
{
  block_of_instructions ;
}

//while with one instruction
while(condition) one_intruction;

//while with one instruction
while(condition)
{
  block_of_instructions ;
}

//do ... while
do
{
  block_of_instructions ;
} while(condition) ;
```

Pseudo-code 2.6 – Les boucles en Java

## 2.5 Tableaux

Plusieurs éléments de même type peuvent être regroupés en tableau. On peut accéder à chaque élément à l'aide d'un d'indice précisant le rang de l'élément dans le tableau. Le premier élément porte l'indice 0. Ils sont initialisés par défaut à 0 pour les nombres et à $false$ pour les tableaux de booléens. En Java, on peut déclarer un tableau de différentes manières (voir le Pseudo-code 2.7 pour les tableaux d'un seul dimension et le Pseudo-code 2.8 pour les tableux de plusieurs dimensions).

```
int tab1[] = {1, 2, 3};// identique au langage C
int[] tab2 = {1, 2, 3};
int[] tab3 ;
tab3=new int[6];/tab3 fait reference a un tableau de 6 entiers
int []t1,t2;//t1 et t2 sont des references a des tableaux d'
 entiers
int t[10] ; // erreur: on ne peut pas indiquer de dimension ici
```

Pseudo-code 2.7 – Déclaration des tableaux simples en Java

```
// ces trois déclarations sont équivalentes
int t [] [] ;
int [] t [] ;
int [] [] t ;

//un tableau de deux lignes et un nombre varie des colonnes
int t [] [] = {new int [3], new int [2]};
```

Pseudo-code 2.8 – Déclaration des tableaux de plusieurs dimensions en Java

R Les éléments d'un tableau peuvent être de type quelconque, et pas seulement d'un type primitif. Il est possible en Java de déclarer des tableaux des objets, par exemple :

```
    Person [] t ;
    t = new Person[3] ;//Person est une classe  définit par le programmeur

    String[] prenoms = new String [2]; //tableau de 2 String
    prenoms[0]="Michel";//prenoms[0] référence la chaine "Michel"
    prenoms[1]="Josette";//prenoms[1] référence la chaine "Josette"
```

La même chose pour les tableaux à deux dimensions, on peut déclarer des tableaux d'objets à deux dimensions.

**Exemple 2** La déclaration du tableau *tab* ci-dessous, déclare que *tab* est une référence à un tableau, dans lequel chaque élément est lui-même une référence à un tableau d'entiers (voir la Figure 2.2). Les éléments du tableaux sont initialisés à 0 par défaut. Par la suite, le tableau est remplit élément par élément.

```
    int [][]tab=new  int [3][3];
    // i=0
    tab[0][0]=25;tab[0][1]=12;tab[0][2]=3;
    // i=1
    tab[1][0]=33;tab[1][1]=4;tab[2][2]=46;
    // i=2
    tab[2][0]=9;tab[2][2]=9;tab[2][2]=6;

    //une autre maniere d'initialisation du tab
    int [][]tab={{25,12,3},{33,4,46},{9,9,6}};
```

Pour itérer ou parcourir un tableau, on utilise une boucle *for* (ou *while*). Aussi, il est nécessaire de connaitre la taille d'un tableau. Pour cela, en java, on utilise l'instruction *tableau.length*. Un exemple du calcul de la somme des éléments d'un

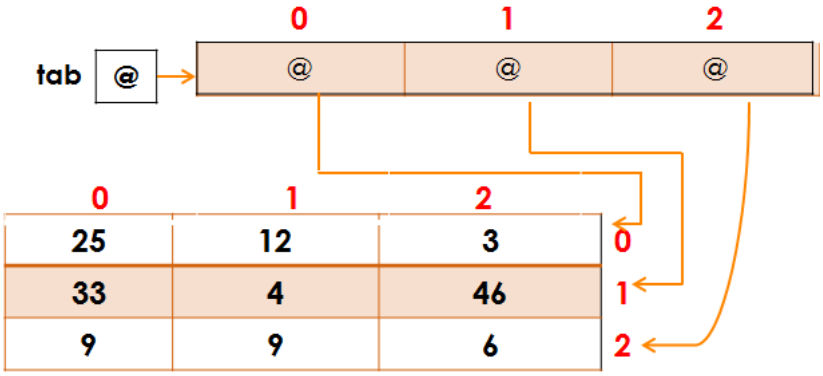



FIGURE 2.2 – Exemple d'un tableau des entiers de deux dimensions

tableau d'une seule dimension $t$ et la somme des éléments d'un tableau de deux dimensions $tt$ est illustré par le Pseudo-code 2.9. Dans l'exemple du tableau à deux dimension $tt$, la première boucle itère sur les lignes et la deuxième sur les colonnes.

```
1  int t[] = new int[5] ;//tableau de 1 dimension
2  int somt=0, somtt;
3  for (int i =0; i<t.length; i++) somt+=t[i] ;
4  System.out.println("La somme des éléments de t="+som);
5
6  int tt[] = new int[5][7] ;//tableau de 2 dimensions
7  for (int i =0 ; i<tt.length ; i++)
8  {
9    for (int j=0; j<tt[i].length; j++) somtt+=t[i][j] ;
10 }
11 System.out.println("La somme des éléments de tt="+somtt);
```

Pseudo-code 2.9 – Exemple de parcours d'un tableau

R Une fois qu'un tableau est créé, sa capacité ne peut pas être modifiée. Par exemple, si on déclare un tableau de 10 éléments, nous sommes limités à stocker au plus 10 éléments dans ce tableau. Si on veut ajouter des éléments, on doit créer un nouveau tableau. On peut généralement éviter un débordement de tableau en le déclara avec une grande capacité. Cependant, si on déclare une trop grande capacité d'un tableau, on peut finir par un sous-utilisation de l'espace.

## 2.6 Chaines de caractères

Pour créer et manipuler des chaînes de caractères, l'environnement JAVA définit deux classes, *String* et *StringBuilder*. Pour initialiser une variable de type String, on utilise les guillemets `""`, par exemple : `String m="Bonjour";`. Comme les tableaux, une variable de type *String* contient une référence vers une chaine de caractères.

Donc, la sémantique des opérateurs d'affectation et d'égalité est la même que les tableaux. La classe offrent une multitude de méthodes pour la manipulation des chaînes de caractères (voir le Pseudo-code 2.10). La Table 2.4 résume quelques fonctions classiques de la classe String.

| **Méthode** | **Résultat renvoyé** |
|---|---|
| *length*() | Nombre de caractères contenus dans la chaîne courante |
| *charAt* | Le caractère dont la position dans la chaîne courante est passée en paramètre |
| *compareTo* | Compare la chaîne courante avec une chaîne passée en paramètre. renvoie zéro si les chaînes sont identiques, une valeur entière négative si l'objet courant est inférieur au paramètre, et une valeur entière positive si l'objet courant lui est supérieur |
| *endsWith* | Teste si la chaîne courante se termine par le suffixe passé en paramètre. Renvoie *true* ou *false* |
| *equals* | Compare cette chaîne à l'objet spécifié. Le résultat est vrai si et seulement si l'argument n'est pas *nul* et est un objet String qui représente la même séquence de caractères que cet objet |
| equalsIgnoreCase | Compare cette chaîne à une autre chaîne, en ignorant la casse (majuscule et minuscule). Renvoie *true* ou *false* |
| *contains* | Renvoie *true* si et seulement si cette chaîne contient une sous chaîne passée en paramètre |
| *indexOf* | renvoie la première position du caractère passé en paramètre dans la chaîne courante |
| *isEmpty* | Renvoie *true* si, et seulement si, *String.length*() vaut 0 |
| *toLowerCase* | Convertit tous les caractères de cette chaîne en minuscule |
| *toUpperCase* | Convertit tous les caractères de cette chaîne en majuscule |

TABLE 2.4 – Quelques méthodes de la classe String

```
1  String c1 = new String ( "bonjour " ) ; //ou String c1="bonjour "
2  String c2 ;
3  c2 = c1 + "á tous " ;
4  String c3 ="BonJour á Tous ";
5  System.out.println( c2 ) ;// bonjour á tous
6  System.out.println(c3) ;//BonJour á Tous
7  System.out.println( c2.length());
8  System.out.println ( c2.charAt( 3 ));
9  System.out.println ( c2.indexOf('o'));
10 System.out.println( c1.contains("á tous "));
11 System.out.println( c3.equalsIgnoreCase(c2));
12 System.out.println( c3.s.charAt(2));
```

Pseudo-code 2.10 – Exemple de parcours d'un tableau

R Les chaînes de caractères d'un objet de type String ne peuvent pas être modifiés. Une même chaîne de caractères de la classe String peut être référencée plusieurs fois puisqu'on est sûr qu'elle ne peut pas être modifiée. Par contre, la classe *StringBuffer* permet la création et la modification d'une chaîne de caractères.

## 2.7 Tableaux dynamiques

Les tableaux dynamiques (liste) ont la particularité de pouvoir changer de taille pendant l'exécution du programme. En Java, on les définit à l'aide de type *ArrayList* de la bibliothèque *java.util.ArrayList*. Pour ce faire, en écrit :

```
1  ArrayList <Type> myList;
2  myList = new <Type> ArrayList();
```

Le type des éléments d'un tableau dynamique ne peut être un type simple (int ou double par exemple). Il doit être l'un des types évolués correspondant aux types de données simples. Par exemple : *Integer* pour *int*, *Float* pour *float*, *Double* pour *double*, etc. Comme il peut être une classe spécifique (*Person* par exemple).

On peut aussi déclarer un tableau dynamique comme suit :

```
1  ArrayList  myList;
2  myList = new  ArrayList();
```

Dans cette déclaration, il n'y a aucune restriction sur le type d'objets qu'on peut ajouter au tableau.

Une fois un tableau dynamique est créée, on peut appliquer les différents méthodes appropriées. L'utilisation de ces méthodes se fait avec la syntaxe suivante : $nomDeTableau.nomDeMethode(arg1, arg2...)$;. Les principales méthodes sont les suivantes :

- ☆ $tableau.add(valeur)$ ajoute *valeur* à la fin de tableau
- ☆ $tableau.size()$ : renvoie la taille du tableau (de type int) ;
- ☆ $tableau.set(i, valeur)$ : affecte valeur à la case i du tableau
- ☆ $tableau.get(i)$ : renvoie l'élément à l'indice $i$ dans le tableau ($i$ doit être un entier compris entre 0 et $tableau.size()$-1)
- ☆ $tableau.isEmpty()$ : détermine si le tableau est vide ou non
- ☆ $tableau.clear()$ : supprime tous les éléments du tableau
- ☆ $tableau.remove(i)$ : supprime l'élément à l'indice i du tableau

**Exemple 3** Le Pseudo-code 2.11 suivant présente un exemple d'utilisation de la méthode *add*, *size* et *get*.

```
1    ArrayList <String> sample = new ArrayList<String>();
2    sample.add("One Java");
3    sample.add("Two Java");
4    sample.add("Three Java");
5    System.out.println(sample.size());
6    System.out.println(sample.get(2));
7    System.out.println(sample.get(3));
8
```

Pseudo-code 2.11 – Exemple des méthodes sur les tableaux dynamiques

## 2.8 Conclusion

Parmi les caractéristiques qui expliquent la popularité et l'acceptation de Java dans l'industrie du logiciel, sa simplicité, grande variété d'environnements d'utilisation, et sa robustesse. Java est un langage orienté objet : tout appartient à une classe sauf les variables et les données de types primitives. Pour accéder à une classe, il faut en déclarer une instance de classe (ou objet). Le chapitre suivant est dédié à la présentation des concepts de base de l'orienté objet illustrés en Java.

# 3. Bases de POO

## Sommaire

## 3.1 Introduction

Nous avons vu précédemment que dans la programmation procédurale, les données et les traitement apparaissent comme des entités séparées. Par exemple, si on veut créer un programme qui permet de calculer la surface d'un rectangle, on doit déclarer deux variable *hauteur* et *largeur*, puis définir une fonction *surface* avec deux paramètres qui désignent la largeur et la hauteur. A la fin, la fonction renvoie comme valeur la surface du triangle (voir le Pseudo-code 3.1). Dans ce cas, les données, qui sont stockées dans des variables, sont séparées des traitements (la fonction surface) et n'ont aucun lien sémantique.

Par contre, dans la POO, les données et les traitements sont regroupés en une seule et même entité appelé "objet" qui est instancié d'une classe. La programmation orientée objet (POO) est basée sur ces deux concepts : Classe et Objet, qui font l'objet principal de ce chapitre.

```
1  package exemples;
2  public class Exemples {
3
4  public static void main(String[] args) {
5     double hauteur=4;
6     double largeur=3.5;
7     System.out.println("La surface= "+ surface(hauteur, largeur));
8      }
9     static double surface(double h, double l)
10     {
11     return l*h;
12     }
13 }
```

Pseudo-code 3.1 – Calcul de la surface d'un rectangle utilisant la programmation procedurale

## 3.2 Notion d'object

Un programme écrit dans un style orienté objet sera composé d'objets en interaction. Un objet est définit comme suit :

**Définition 3.1** Un objet est une chose, à la fois tangible et intangible, qu'on peut imaginer. Chaque objet a son propre état, comportement et identité.

Dans la poo, on sépare le problème en objets. Par exemples, pour un programme permettant de suivre les étudiants résidents dans une cité universitaire, on peut avoir

de nombreux objets *Student*, *Room* et *Floor*. Pour un autre programme permettant de suivre les clients et l'inventaire d'un magasin de vélos, on peut avoir *Client*, *Vélo* et de nombreux autres types d'objets.

Un objet est composé de données et d'opérations qui manipulent ces données. Par exemple, un objet *Étudiant* peut avoir des données telles que le nom, le sexe, la date de naissance, l'adresse, le numéro de téléphone et l'âge, ainsi que des opérations permettant d'affecter et de modifier ces valeurs de données.

R Dans un même programme on peut avoir de nombreux objets du même type. Par exemple, dans le programme de magasin de vélos, on s'attend à voir de nombreux vélos et autres objets.

R Un objet doit être instanciée d'une classe. Donc, une classe doit être définie avant la création d'une instance (objet) de la classe.

## 3.3 Notions de classe

Dans un programme, on écrit des instructions pour créer des objets. Pour que l'ordinateur puisse créer un objet, on doit fournir une définition, appelée *classe*.

**Définition 3.2** Une classe est une sorte de forme ou de modèle qui décrit et impose ce que les objets peuvent et ne peuvent pas faire. En d'autres termes, une *classe* est un nouveau type de données dont les instances sont des objets.

Pour déclarer une nouvelle classe il suffit d'utiliser le mot clé "class" suivi du nom de la classe :

```
1 class MyClasse {
2 ...
3 }
```

Un objet est appelé une *instance* d'une classe. Un objet est une instance d'exactement d'une seule classe. Une classe se compose de variables d'instance et de méthodes. Un exemple de définition d'une classe Rectangle et la création d'un objet $r$ de type Rectanle sont illustrées par la Figure 3.2. On dit alors que $r$ est une instance de la classe *Rectangle*.

```
1 package exemples;
2 public class Exemples {
3 public static void main(String[] args) {
4     //Create new object r of class Rectangle
5     Rectangle r=new Rectangle();
6     r.hauteur=4.3;
7     r.largeur=5.2;
8     System.out.println(r.hauteur);
9     System.out.println(r.largeur);
```

```
10      System.out.println(r.surface());
11  }
12  }
13  // Definition of a new class Rectangle
14  class Rectangle{
15      double hauteur;
16      double largeur;
17
18      double surface()
19      {   double s=hauteur*largeur;
20          return s;
21      }
22  }
```

Pseudo-code 3.2 – Définition d'une classe *Rectangle* et un objet $r$ de type *Rectangle*

## 3.4 Création des objets

Une fois une nouvelle classe est définit, supposons que le nom de la classe soit $MyClass$, il est possible de créer des objets de ce type, qui sont des instances de la classe. Chaque objet créé possède tous les attributs de la classe dont il est issu. Par exemple, pour déclarer un objet obja de la classe $MyClass$ on utilise l'instruction suivante :

```
1  MyClass obja=new MyClass();
```

En fait, la ligne ci-dessus peut être décomposée en deux lignes comme suit :

```
1  MyClass obja;//Line 1
2  obja=new MyClass();//Line 2
```

A la fin de la première ligne, *obja* est une référence. Jusqu'à ce point, il n'y a pas d'allocation mémoire. Mais une fois le mot *new* s'exécute (Line2), la mémoire est allouée. Le Pseudo-code 3.2 est un exemple de définition d'une classe *Rectangle* est d'une variable (objet) $r$ de type *Rectangle* et dont les attributs sont initialisés à la valeur réelle 0.0 par défaut.

R Dans la deuxième ligne, le nom de la classe est suivi des parenthèses. Elles servent pour les constructeurs (constructors). Les constructeurs sont utilisés pour décrire ce qui se passe lorsqu'un objet est créé. Les attributs de l'objet prennent des valeurs initiales données par un constructeur. Pour chaque classe, il existe un constructeur par défaut qui initialise les attributs à des valeurs initiales par défaut.

Le Pseudo-code 3.3 est un exemple de création d'une classe *Person* et une variable *Alice* de type *Person* dont les attributs sont par défaut initialisés, respectivement,

aux valeurs *null*, 0 et *null*. Par la suite, l'objet est utilisé en modifiant son état par l'attribution des nouvelles valeurs à ces attributs. En fin, la méthode (fonction) $Affiche r$ est invoqué. Dans cette exemple, la méthode afficher est désignée pour $Affiche r$ les valeurs des attributs.

```
1  package newclasse;
2  public class NewClasse {
3      public static void main(String[] args) {
4      Person Alice=new Person();//create new object Alice of Person
5          Alice.name="Alice Dunod";
6          Alice.adress="Madrid";
7          Alice.age=23;
8          Alice.Afficher();
9      }
10 }
11 class Person
12 {
13     String name;
14     String adress;
15     int age;
16 void Afficher()
17 {
18 System.out.println(name+" "+age+" ans habit a "+adress);
19 }
20
21 }
```

Pseudo-code 3.3 – Exemple de déclaration d'une classe *Person* et une instance *Alice* de ce type

R La variable qui désigne l'objet créé est une référence à l'objet, et non pas l'objet lui-même. Supposant la création des deux objets $r1$ et $r2$ de types Rectangle, comme suit :

```
1  Rectangle r1=new Rectangle();//Create r1
2  Rectangle r2=new Rectangle();//Create r2
3  //r1 and r2 are tow different objects
4  r1=r2;//Here, r1 and r2 designate the same object
```

L'affectation $r1 \leftarrow r2$ affecte à $r1$ la référence à l'objet désigné par $r2$. Après l'affectation, les références $r1$ et $r2$ désignent le même objet.

## 3.5 Attributs

Les données stockées dans un objet représentent l'état de l'objet. Dans la terminologie de la POO, ces données sont appelées *attributs*. Les attributs contiennent les

informations qui permettent de différencier un objet d'un autre. Par exemple, les attributs des employés peuvent être le numéro de sécurité sociale, la date de naissance, le sexe, le numéro de téléphone, etc. Chaque classe doit définir les attributs qui vont stocker l'état de chaque objet instancié à partir de cette classe. Dans l'exemple de classe *Person* (voir le Pseudo-code 3.3), la classe *Person* définit des attributs pour le nom, l'adresse et l'age. Ainsi, Dans l'exemple de classe *Rectangle* (voir le Pseudo-code 3.3), la classe *Rectangle* définit deux attributs : *hauteur* et *largeur*.

Pour déclarer des attributs d'une classe, on utilise la syntaxe suivant :

```
class My_class {
type_attribut1 name_attribut1;
type_attribut2 name_attribut2;
type_attribut3 name_attribut3;
...
}
```

L'accès aux valeur des attributs d'une instance se fait comme suit :

```
object_name.name_attribut; //Access to the attribute value
```

Par exemple, pour accéder aux valeurs des attributs d'une instance *Alice*, de la classe *Person*, on a écrit (voir le Pseudo-code 3.3) :

```
Alice.name    //Access to the attribute name's value
Alice.adress //Access to the attribute adress's value
Alice.age   //Access to the attribute age's value
```

de la même chôse, pour accéder aux valeurs des attributs d'une instance $r$, de la classe *Rectangle*, on a écrit (voir le Pseudo-code 3.2) :

```
r.hauteur    //Access to the attribute "hauteur" value
r.largeur //Access to the attribute "largeur" value
```

Une classe peut avoir deux types d'attributs : les variables d'instance et les variables de classe, également appelées respectivement, variables non statiques et statiques.

1. **Attribut d'instance :** les variables d'instance représentent l'état d'un objet de la classe. Une copie de toutes les variables d'instance existe pour chaque objet de la classe. Ils sont déclarés sans le mot-clé *static* et sont uniques pour chaque instance de la classe. Un attribut d'instance n'est accessible qu'à l'objet qui le possède et stockera le nom de l'objet.

2. **Attribut de classe :** les variables de classe (static variable) représentent l'état de la classe elle-même. Une seule copie des variables de classe existe pour une classe qui s'initialise une seule fois au début de l'exécution. Si une variable

est déclarée statique (static), alors la valeur de la variable est la même pour toutes les instances, et nous n'avons pas besoin de créer un objet pour appeler et ou modifier cette variable.

**Exemple 4** Considérant le Pseudo-code 3.4. La classe $B$ a deux attributs $b$ et $y$. L'attribut $b$ est déclaré *static* et $y$ *non – statique*. La création des deux instances $a1$ et $a2$ de la classe $B$ engendre ce qui suit (voir la Figure 3.1 ) :

- ☆ L'objet $a1$ a une copie de l'attribut $y$. Dans ce cas, $y$ de l'objet $a1$ n'est accessible qu'à l'objet $a1$ ;
- ☆ L'objet $a1$ a une copie de l'attribut $y$. Dans ce cas, $y$ de l'objet $a2$ n'est accessible qu'à l'objet $a2$ ; ;
- ☆ Les deux objets $a1$ et $a2$ partagent le même attribut $b$. Puisque $b$ est statique, quelque soit le nombre d'instanciation de la classe $B$, $b$ existe en un et un seul exemplaire.

```
1  class B{
2    staic int b;
3    float y;
4  }
5
6    B a1=new B(), a2=new B();
7
8    a1.y=10; a2.Y=12; a1.n=3; a2.n=5;
9    System.out.println(a1.y + " " +a2.y+" "+ a1.n + " " +a2.n);
10   //10 12 5 5
11   a1.y++; a2.Y++; a2.n++;
12   System.out.println(a1.y + " " +a2.y+" "+ a1.n + " " +a2.n);
13   //11 13 6 6
14   B.n=7;
15   System.out.println(a1.y + " " +a2.y+" "+ a1.n + " " +a2.n);
16   //11 13 7 7
```

Pseudo-code 3.4 – Exemple de variables statiques vs non statiques

## 3.6 Méthodes

Les fonctions (Méthodes en Java) en programmation sont des traitements, qui agissent sur des données. Les méthodes sont des morceaux de code réutilisables, que l'on définit à un endroit du programme et que l'on peut appeler depuis un ou plusieurs autre(s) endroit(s). Les méthodes permettent de :

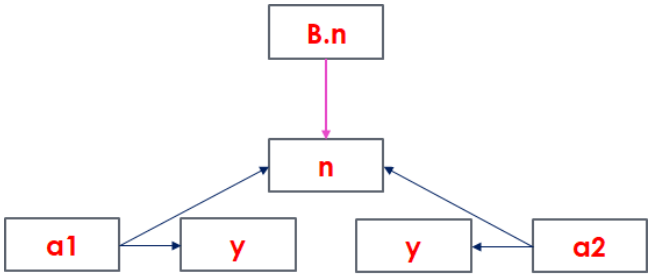



FIGURE 3.1 – Exemple de variables statiques vs non statiques.

- ☆ éviter la duplication de code
- ☆ facilement maintenir du code (si on veut changer le code, il suffit de changer le code de la fonction)
- ☆ avoir un programme bien structuré et facile à comprendre.
- ☆ faciliter la réutilisation du code

### 3.6.1 Généralités sur les méthodes

Les méthodes (les fonctions dans la programmation procédurale) implémentent le comportement requis d'une classe. Chaque objet instancié à partir de cette classe inclut des méthodes telles que définies par la classe. Les principaux caractéristiques des méthodes sont les suivants :

- ☆ Les méthodes peuvent implémenter des comportements appelés à partir d'autres objets (messages) ou fournir le comportement interne fondamental de la classe. Les comportements internes sont des méthodes privées qui ne sont pas accessibles par d'autres objets. Par exemple, dans la classe *Person* (Pseudo-code 3.3), les comportements sont seulement la méthode $Afficher$, qui permet d'afficher les valeurs de tous les attributs. Dans la classe *Rectangle* (Pseudo-code 3.2), le seul comportement définit est la méthode $surface$, qui retourne un double et qui ne prend pas de paramètres. Donc, les méthodes sont des fonctions qui sont déclarées dans une classe. L'entête d'une méthode indique son type de retour suivi par son nom et la liste des paramètres requises.
- ☆ Une méthode peut avoir zéro paramètre. Si une méthode n'a aucun paramètre, on dit que la méthode n'a aucun paramètre ou que la méthode ne prend/accepte aucun paramètre. Une méthode peut prendre des valeurs d'entrée (paramètres) de son appelant et un paramètre est utilisé pour prendre une valeur d'entrée de l'appelant. Une méthode est toujours définie dans le corps d'une classe.
- ☆ Le type de retour d'une méthode est le type de données renvoyé par la méthode

lorsqu'elle sera invoquée. Ce type de données peut être un type de données primitif (par exemple, int, double, booléen, etc.) ou d'un type de référence (par exemple, Person, String, etc.).

R Parfois, une méthode ne renvoie pas de valeur. Le mot clé *void* est utilisé comme type de retour si une méthode ne renvoie aucune valeur.

**Exemple 5** Par exemple, dans la classe *Rectangle* du Pseudo-code 3.2, la méthode $surface$ n'a aucun paramètre et renvoie une valeur de type *double* qui représente la surface d'un rectangle. Par contre, dans l'exemple de la classe *Person*, la méthode $Afficher$ ne renvoie aucune valeur.

### 3.6.2 Définitions des méthodes

Une méthode doit avoir les éléments suivants :

- ☆ un corps : la portion de code à exécuter ;
- ☆ un nom : celui par lequel on désignera cette fonction ;
- ☆ zéro ou plusieurs paramètres (des arguments) : ensemble de variables (extérieures à la fonction) que la fonction prend en entrée, et dont le corps a besoin pour fonctionner ;
- ☆ un type et une valeur de retour : la sortie de la fonction, ce qu'elle renvoie au reste du programme ;
- ☆ un appel qui consiste à l'endroit où l'on utilise la méthode. Il contient le nom de la méthode et les arguments que l'on veut lui passer.

La syntaxe de définition des méthodes en Java est la suivante :

```
return_type name_method (type p1, type p2, ...)
{
  // Body of the method goes here
}
```

**Exemple 6** Le Pseudo-code 3.5 présente un exemple de définition et d'utilisation (appel) d'une fonction simple, nommée *somme*. La fonction *somme* prend comme arguments deux entiers, $a$ et $b$, calcule et renvoie un résultat de type entier qui est la somme de ces deux nombres.

```
public static void main(String[] args)
{
  int x = 10, y = 15;
  System.out.print(somme(x,y));//appel de la fonction somme
}

```

```
7  //définition de la fonction somme
8  static int somme(int a, int b) {
9    return a+b;
10 }
```

Pseudo-code 3.5 – Exemple de définition et utilisation d'une fonction en Java

R Parfois, une méthode ne renvoie pas de valeur. Le mot clé *void* est utilisé comme type de retour si une méthode ne renvoie aucune valeur.

### 3.6.3 Signature d'une méthode

Une méthode a une signature, qui identifie de manière unique la méthode dans un contexte particulier. La signature d'une méthode est la combinaison des quatre parties suivantes :

- ☆ Nom de la méthode
- ☆ Nombre de paramètres
- ☆ Ordre de paramètres
- ☆ Types de paramètres

Dans le Pseudo-code 3.6, quatre méthodes sont définies. Ces méthodes portes le même nom mais se diffèrent par le type et le nombre des arguments.

```
1  int somme(int a){
2    return a+a;
3  }
4
5  int somme(int a, int b){
6    return a+b;
7  }
8
9  int somme(int a, int b, int c){
10   return a+b+c;
11 }
12
13 double somme(double a, double b){
14   return a+b;
15 }
```

Pseudo-code 3.6 – Signature d'une méthode

En Java, il est de ce fait possible de définir plusieurs méthodes de même nom si ces méthodes n'ont pas les mêmes listes de paramètres (nombre ou types de paramètres différents). Ce mécanisme, appelé *surcharge des méthodes*, est très utile pour écrire

des méthodes sensibles au type de leurs arguments (des méthodes correspondant à des traitements de même nature mais s'appliquant à des entités de types différents).

### 3.6.4 Évaluation d'un appel de méthode

Dans le cas où une méthode nécessite pour des paramètres entrants pour renvoyer en sortie une valeur concrète, l'appel de méthode se passe par cinq étapes :

1. Les expressions passées en argument sont évaluées (en peut passer, lors de l'appel d'une fonction, comme arguments soit des valeurs concrètes, soit des variables soit des expressions)
2. Les valeurs correspondantes sont affectées aux paramètres de la méthode (les valeurs sont copiées dans les paramètres de la méthode)
3. Le corps de la méthode s'exécute avec ces valeurs
4. L'expression suivant la première commande return rencontrée est évaluée
5. La valeur obtenue est retournée comme résultat de l'appel : cette valeur remplace l'expression de l'appel

R Si une méthode est sans arguments, les étapes 1 et 2 n'ont pas lieu. Ainsi, si la méthode est sans valeur de retour (type de retour *void*), les étapes 4 et 5 n'ont pas lieu.

### 3.6.5 Références et passage de paramètres

En programmation, de façon générale, on dira que :

- ☆ Un argument est passé par valeur si la méthode ne peut pas le modifier : la méthode créée une copie locale de cet argument (on travaille avec une copie)
- ☆ Un argument est passé par référence (adresse) si la méthode peut le modifier (on connaît son adresse)

En Java, il n'existe que le passage par valeur, mais cela a des conséquences différentes selon que le type du paramètre est simple (int, double, etc.) ou évolué (objet).

```
1  // Qu'affiche ce code:
2
3  public static void main(String[] args) {
4    int x = 10, y = 15;
5    System.out.print(x + " " + y + " ");
6
7    f1(x,y);
8    System.out.println(x + " " + y + " ");
```

```
9
10    String s1 = "abcd", s2 = "efgh";
11    System.out.print(s1 + s2);
12
13    f2(s1,s2);
14    System.out.print(s1 + s2);
15  }
16
17  static void f1(int a, int b) {
18    int tmp = a; a = b; b = tmp;
19  }
20
21  static void f2(String s, String t) {
22    String tmp = s; s = t; t = tmp;
23  }
```

Pseudo-code 3.7 – Passage de paramètres par valeur

**Exemple 7** Pour les entiers le passage est par valeur (voir le Pseudo-code 3.7). Par contre, pour les tableaux, le passage des paramètres s'effectue par adresse (par référence). On peut accéder et modifier les valeurs du tableau passé en paramètre (voir le Pseudo-code 3.8).

```
1  // Qu'affiche le code suivant:
2  public static void main(String[] args)  {
3    int[] tab = {1, 2, 3, 4};
4    System.out.println(tab[0] + " " + tab[1]);
5    f(tab, 0, 1);
6    System.out.println(tab[0] + " " + tab[1]);
7  }
8  static void f(int[] t, int i, int j)
9  {
10   int tmp = t[i];
11   t[i] = t[j];
12   t[j] = tmp;
13   t = new int[4];
14 }
```

Pseudo-code 3.8 – Passage de paramètres par référence

R Les modifications faites dans la méthode sur la référence elle-même ne sont pas visibles à l'extérieur de la méthode. Par exemple, dans le Pseudo-code 3.9, l'appelle de la fonction *add* peut modifier la chaine de caractères référencées par *st*, mais l'appelle de la fonction *replace* ne peut pas modifier la valeur de *st* (qui est une adresse).

```
1  public static void main(String[] args) {
2    StringBuilder st=new StringBuilder ("Bonjour");
3
4    add(st);
5    replace(st);
6    System.out.println(st);
7  }
8
9  static void add(StringBuilder s)
10 {
11   s.append(" tout le monde");
12 }
13
14 static void replace(StringBuilder s)
15 {
16   StringBuilder ch=new StringBuilder("Hello every one");
17   s=ch;
18 }
```

Pseudo-code 3.9 – Exemple de passage de paramètres par valeur et par référence

### 3.6.6 Méthodes d'instance et méthodes de classe

Une classe peut avoir deux types de méthodes : les méthodes d'instance et les méthodes de classe. Les méthodes d'instance et les méthodes de classe sont également appelées, respectivement, méthodes non statiques et méthodes statiques. Une comparaison entre les deux types de méthodes est illustré dans la Table 3.1.

Une méthode d'instance (d'objet) est utilisée pour implémenter le comportement des instances (des objets) de la classe. Une méthode d'instance ne peut être invoquée que dans le contexte d'une instance de la classe. La syntaxe pour invoquer une méthode d'instance est la suivante :

```
1 objectname.instanceMethodName(actual-parameters);
2
3 // Exemple:
4
5 String st=new String("Hello");
6 st.length();//length() est une méthode d'instance
```

Une méthode de classe est utilisée pour implémenter le comportement de la classe elle-même. Une méthode de classe s'exécute toujours dans le contexte d'une classe. Le mot clé *static* est utilisé pour définir une méthode de classe. L'absence du mot *static* dans la déclaration d'une méthode rendre la méthode une *méthode d'instance*. Voici des exemples de déclaration de méthodes statiques et non statiques :

| | **Méthode de classe** | **Méthode d'instance** |
|---|---|---|
| Déclaration | Mot clé *static* | Absence du Mot clé *static* |
| Appelle (syntaxe) | $ClassName.MethodeName(args)$ | $ObjectName.MethodName(args)$; |
| Accès aux attributs | Attributs statiques | Tous les attributs |
| Appelle (condition) | Peut être appeler sans avoir instancié la classe | Un objet doit être instancié avant l'appelle de la méthode |

TABLE 3.1 – Méthodes de classe vs. Méthodes d'instance

```
// A static or class method
static void aClassMethod() {
// Method's body goes here
}

// A non-static or instance method
void anInstanceMethod() {
// Method's body goes here
}
```

Une méthode de classe ou une méthode *statique* est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Une méthode de classe :

- ☆ peut être appelée même sans avoir instancié la classe
- ☆ peut accéder uniquement à des variables et méthodes statiques.
- ☆ dans la déclaration, le nom de la méthode est précédé du mot clé *static*
- ☆ peut être appelée avec la notation *classe.methode()* au lieu de *objet.methode()*

**Exemple 8** Le Pseudo-code 3.10 présente des exemples des méthodes de classes.

```
int e = Math.abs(j);
boolean m=Character.isLowercase('Z');

int tab[]={12,8,16,10,14};
System.out.println(max(tab));

int e = Math.abs(j);
boolean m=Character.isLowercase('Z');

int tab[]={12,8,16,10,14};
System.out.println(max(tab));
```

Pseudo-code 3.10 – Exemple des méthodes de classes

| | Méthode | Constructeur |
|---|---|---|
| Nom | Nom au choix | Même nom de la classe |
| Type de retour | N'importe quel type +void | Aucun |
| arguments | plusieurs ou zéro | plusieurs ou zéro |
| Signature | Nom, type et ordre et nombre de paramètres | type et ordre et nombre de paramètres |
| Mot clé *static* | Oui possible | Non |
| Nivaux d'accès | Public, private, protected | Public, private, protected |

TABLE 3.2 – Méthode vs. Constructeur

## 3.7 Constructeurs

La notion de constructeur permet d'automatiser le mécanisme d'initialisation d'un objet. Cette initialisation ne sera pas limitée à la mise en place de valeurs initiales ; il pourra s'agir de n'importe quelles actions utiles au bon fonctionnement de l'objet.

Un constructeur n'est rien d'autre qu'une méthode, sans valeur de retour, portant le même nom que la classe. Comme il peut disposer d'un nombre quelconque d'arguments (éventuellement aucun). La différence entre une méthode et un constructeur sont présentés dans la Table 3.2.

R Généralement, le constructeur servait à donner des valeurs initiales aux attributs d'un objet. Mais, il faut bien comprendre qu'un constructeur peut très bien réaliser d'autres actions, par exemple : vérification d'existence de fichier, ouverture d'une connexion Internet, vérification d'une valeur, etc.

R L'instanciation d'un objet, réalisée par un appel, réalise deux opérations :
- ☆ allocation d'un emplacement mémoire pour cet objet ;
- ☆ appel éventuel du constructeur pour cet objet.

Ces deux opérations sont indissociables. Le constructeur ne peut pas être appelé directement (sur un objet existant), en court-circuitant la première opération.

### 3.7.1 Caractéristiques d'un constructeur

Lors la définition d'un constructeur d'une classe on doit prendre en considération ce qui suit :

- ☆ Le nom du constructeur doit toujours être le même que celui de la classe.
- ☆ Il ne doit pas y avoir de type de retour du constructeur.
- ☆ Un constructeur peut avoir plusieurs arguments ou zéro arguments
- ☆ Un constructeur est toujours invoqué lorsqu'un objet est instancié
- ☆ Un constructeur ne doit pas déclarer avec les mots clés abstract, static et final

### 3.7.2 Constructeur par défaut et autres constructeurs

Lorsque aucun constructeur explicite n'est définit dans une classe, le compilateur crée un constructeur par défaut. Le constructeur par défaut est un constructeur sans arguments et permet de créer un objet même si aucun constructeur n'est défini.

Lorsque un constructeur explicite paramétré est définit, l'utilisation d'un constructeur sans argument génère une erreur de compilation. Dans ce ca, un constructeur sans argument doit être crée manuellement, avant de l'invoquer.

**Exemple 9** Un exemple d'un constructeur paramétré est illustré par le Pseudo-code 3.11. Le constructeur par défaut de la classe $A$ est éliminé une fois qu'un constructeur avec des arguments est crée et on ne peut pas crée un objet de la classe $A$ utilisant le constructeur par défaut : $A\ a1 = new\ A()$;. Dans ce cas, pour le faire, on doit créer un constructeur sans argument, comme illustré par le Pseudo-code 3.12.

```
class A {
    int a;
    String b;
    char c;

    A(int x, String s) {
        a = x;
        b = s;
   }}

// Main
public class Main{
    public static void main(String[] args) {
        A a = new A(1, "One");
        System.out.println(a.a + " " + a.b);
//      A a1= new A(); ERROR
  }}
```

Pseudo-code 3.11 – Exemples d'un constructeur paramétré

```
class A {
    int a;
    String b;
    char c;

    A(int x, String s) {
        a = x;
        b = s;
    }

```

```
    A(){
        a = 0;
        b = " ";
    }}

// Main
public class Main{
    public static void main(String[] args) {
        A a = new A(1, "One");
        System.out.println(a.a + " " + a.b);

        A c= new A();
      System.out.println(c.a + " " + c.b);
    }
}
```

Pseudo-code 3.12 – Exemples d'un constructeur sans paramètres et un paramétré

R Une classe peut avoir plusieurs constructeurs (des constructeurs surchargés). La surcharge des constructeurs est effectuée pour initialiser les attributs d'une classe de différentes manières. Les constructeurs surchargés doivent différer par le nombre de paramètres ou le type de données des paramètres qui leur sont transmis. La signature de chaque constructeur doit être différente des autres.

### 3.7.3 Appel d'un constructeur à l'intérieur d'un autre

L'une des utilisations du mot clé *this* est de faire un appel d'un constructeur à partir d'un autre constructeur, utilisant la syntaxe suivante :

```
  this( arg1, arg2, ...);
```

Dans ce cas,le constructeur qui correspond à la liste de paramètres sera appelé.

**Exemple 10** On peut, par exemple, ajouter un constructeur à la classe $A$ du Pseudo-code 3.12 comme suit :

```
    A(int x, String s, char e)
    {
      this( x, s);
      c=e;
    }
```

R On peut ajouter plus d'instructions après l'instruction *this* dans un constructeur, mais pas avant. En d'autres termes, l'appel à *this* dans un constructeur doit être la première instruction du constructeur.

### 3.7.4 Constructeur de copie

En Java, il est possible de créer une copie d'une instance au moyen de ce que l'on appelle le constructeur de copie. Ce constructeur s'utilise de la même façon que les autres et prend un seul argument, soit l'instance à copier :

```
1 ClassName Instance1 = new ClassName(arguments-list) ;
2 ClassName Instance2 = new ClassName(Instance1) ;
```

Ici, $Instance1$ et $Instance2$ sont deux instances distinctes mais ayant des mêmes valeurs pour leurs attributs (au moins juste après la copie). En plus, dans la classe $ClassName$, un constructeur avec un argument de type $ClassName$ doit être définit, qui sert à copier « champ par champ » tous les attributs.

Une autre manière de créer des copies consiste à définir une méthode en charge de la copie. Généralement, il s'agit de la méthode $clone()$ en Java.

## 3.8 Conclusion

Nous avons vu dans ce chapitre les concepts de base de la programmation orientée objet : classe et objet. Nous avons vu aussi comment créer et instancier une classe en Java, comment déclarer et appeler une méthode, et comment définir et utiliser des constructeurs de classes. La programmation orientée objet est basée aussi sur d'autres concepts, tels que niveaux d'accessibilité, l'héritage, le polymorphisme, les classes abstraites, et les interfaces, qui font l'objet des chapitres suivants.

# 4. Encapsulation et niveaux de visibilité

Sommaire



## 4.1 Introduction

Nous avons vu dans le chapitre précédent que la programmation orientée objet permet le regroupement des traitements et des données en une seule et même entité. Les données sont désignées par le terme "attribut" et les traitements par le terme "méthode". Le tout sera regroupé au sein d'une classe, qui constituera un nouveau type de données, une catégorie d'objets. Une variable de ce type sera aussi désignée par le terme d'objet ou d'instance.

En effet, il y a deux niveaux de perception d'un objet : le niveau externe, visible, utile au programmeur utilisateur, celui qui utilise l'objet. Cet utilisateur n'a pas besoin de connaître les détails techniques des différentes méthodes, tout comme nous n'avons pas besoin de savoir comment un moteur fonctionne pour pouvoir conduire une voiture. Le second niveau est le niveau interne, qui concerne le programmeur

concepteur, celui qui se charge des détails d'implémentation.

Le programmeur peut définir différents niveaux de visibilité et d'accessibilité de la classe, ce qui donne une grande robustesse au programme face aux changements et aux erreurs de manipulation. Dans ce chapitre, nous allons examiner les concepts de la programmation orienté objet liés à la visibilité et l'accessibilité de données et traitements.

## 4.2 Encapsulation

Encapsulation est l'un des meilleurs concepts de POO. Le terme encapsulation est utilisé pour désigner deux choses différentes : un processus ou une entité. En tant que processus, il s'agit d'un acte consistant à regrouper un ou plusieurs éléments dans un conteneur. Le conteneur peut être physique ou logique. En tant qu'entité, il s'agit d'un conteneur contenant un ou plusieurs éléments.

Les langages de programmation prennent en charge l'encapsulation de plusieurs manières. Une procédure est une encapsulation des étapes de réalisation d'une tâche, un tableau est une encapsulation de plusieurs éléments du même type, etc. Dans la POO, l'encapsulation consiste à regrouper des données et des opérations sur les données dans une entité appelée *classe*.

Dans la POO, on ne permet pas aux données de circuler librement à l'intérieur du système. Au lieu de cela, on regroupe les données et les méthodes dans une seule unité (c'est-à-dire dans une classe). L'objectif de l'encapsulation est au moins l'un des suivants :

1. Mettre en place des restrictions pour que les composants d'un objet impossible d'y accéder directement ;
2. Lier les données avec des méthodes qui agirent sur ces données (c'est-à-dire former une capsule).

En Java, on peut implémenter l'encapsulation de différentes manières. Par exemple : on regroupe les données et les méthodes qui opèrent sur les données dans une entité appelée *classe*, on peut utiliser les spécificateurs d'accès (ou modificateurs) et les méthodes getter-setter, on peut regrouper une ou plusieurs classes logiquement liées dans une entité appelée package, etc.

## 4.3 Notion de package

Pour des raisons de simplicité, on place toutes les classes d'un programme définies par le programmeur dans un même dossier. Cette approche fonctionne bien lorsque

on apprenne la programmation et ne traite pas de nombreuses classes.

Un package en Java est une collection logique d'une ou plusieurs classes liées. Un package crée une nouvelle portée de dénomination dans laquelle toutes les classes doivent avoir des noms uniques. Deux classes peuvent porter le même nom en Java à condition qu'elles soient regroupées (ou encapsulées) dans deux packages différents.

## 4.4 Niveaux de visibilité

Selon leur niveau de visibilité (accessibilité), les classes, les attributs et les méthodes peuvent être accessibles ou non accessibles depuis des classes du même paquetage ou d'autres paquetages. Pour définir les niveaux de visibilité, on utilise les modificateurs de visibilité *private*, *protected* et *public*. Ces modificateurs sont placés devant l'entête d'une classe, d'une méthode ou devant le type d'un attribut.

### 4.4.1 Classe publique et visibilité package

Si aucun modificateur n'est signalé, il s'agit de la visibilité par défaut qui est la visibilité package, c'est-à-dire que la classe (l'attribut ou la méthode) est visible dans le package où elle est définie.

R Afin d'utiliser une classe (déclarée public) dans un package externe, il faut l'importer. Pour cala, il suffit d'inclure en début de fichier le chemin d'accès à la classe qu'on veut importer en utilisant l'instruction *import*, utilisant la syntaxe suivante :

```
1    import PackageName.ClassName ;
```

Une classe possède par défaut la visibilité package. Si le modificateur public n'est pas indiqué, elle est accessible par toute classe qui est défini dans le même package et elle n'est pas accessible dans les packages externes (les autres). On peut attribuer à une classe le modificateur *public* qui la rend accessible de n'importe quel autre package (package externe), c'est le plus haut niveau de visibilité.

Une seule classe publique doit être définit par fichier. Cette classe publique doit avoir exactement le même nom que celui du fichier. Mais, outre la classe publique, les autres seront considérés comme des classes restreintes au package dans lequel est placé le fichier. Elles ne devront porter aucun qualificateur de visibilité.

### 4.4.2 Classe interne et autres membres de classe

Une classe interne est une classe définie à l'intérieur d'une autre classe. Si une classe est définie sous forme de membre d'une autre classe, on peut utiliser un des niveaux de visibilité classiques : *public*, *protected*, et *private*.

Un membre d'une classe C (classes, attributs, méthodes) peut être :

- ☆ *public* : accessible à toutes les classes.
- ☆ *private* : accessible seulement à la classe où le membre est définit ;
- ☆ *protected* : accessible aux classes dans le même paquetage et à toute classe dérivée en dehors du paquetage ;
- ☆ par défaut (package friendly ) : valeur par défaut, accessible aux classes dans le même paquetage ;

**Exemple 11** Considérant les classes de la Figure 4.1.

- ☆ La classe publique $C$ du package $P$ est accessible partout (accessibles à partir des classes $C, D, P1, P2$). Tandis que la classe $D$ du même package, a une visibilité par défaut, n'est visible que par la classe $C$ ;
- ☆ L'attribut $a$ de la classe $C$ est définit publique. Donc elle est visible partout. Par contre, l'attribut $d$ du même classe et l'attribut $k$ de la classe $D$ sont définit privés, et dans ce cas ne sont visibles que dans leurs classes de définition.
- ☆ L'attribut $b$ de la classe $D$ et l'attribut $f$ de la classe $D$ ont une visibilité par défaut. Donc, ils sont visibles seulement dans les classes du même package, qui sont la classe $C$ et la classe $D$.
- ☆ Concernant les méthodes, la méthodes $m$ est définit publique et elle est accessible à partir des classes $C, D, P1$, et $P2$. Par contre, la méthode $s$, qui a une visibilité par défaut, n'est accessible qu'à partir les classes du même package, $C$ et $D$.

```java
package p ;

public class C {
  public int a;
  int b;
  private int d;

  int s(){
      return a+b;
  }

  public int m(){
      return a*b*d;
  }

}
class D {
    int f;
    private int k;
}
```

```java
package p1;

public class P1 {
  // code
}
```

```java
package p2;
import p1.*;
public class P2 {
    // code ..
}
```

| Visibilité | Classes & Membres de classes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | C | D | a | b | s | m | f | k | d |
| C | √ | √ | √ | √ | √ | √ | √ | √ | X |
| D | √ | √ | √ | √ | √ | √ | √ | X | √ |
| P1 | √ | X | √ | X | X | √ | X | X | X |
| P2 | √ | X | √ | X | X | √ | X | X | X |

FIGURE 4.1 – Exemple de visibilité de classes, méthodes, et attributs

R En pratique, il est recommandé de définir :
- ☆ Toutes les classes publiques afin d'augmenter la réutilisation
- ☆ Toutes les méthodes publiques (ou protégées) afin de pouvoir les invoquer dans des packages externes
- ☆ Tous les attributs privés (ou protégés) afin de garder le maximum de contrôle sur leur manipulation

## 4.5 Accesseurs et et manipulateurs (get et set)

Pour accéder aux attributs privés à l'extérieur de leur classe, on doit fournir des méthodes spécifiques appelées méthodes d'accès ou accesseurs et manipulateurs : les méthodes *get* et *set*.

1. La méthode *get* permet de renvoyer la valeur de l'attribut et permet de connaître la valeur d'un attribut privé ;
2. La méthode *set* permet de modifier la valeur de d'un attribut privé. De telles méthodes prennent donc en paramètre la valeur à affecter et ne retournent rien.

parfois, on fournit juste une méthode *get* pour un accès en lecture seulement et parfois, un attribut privé peut être caché (n'est pas accessible ni en en lecture ni en écriture, aucune méthode).

**Exemple 12** Reprenant la classe *Rectangle* du Pseudo-code 3.2. Dans le Pseudo-code 4.1, les deux attributs de la classe *Rectangle* sont modifiés pour être privé. Pour permettre l'accès à ces deux attributs à l'extérieur de la classe Rectangle, on a définit deux *getters*, *getH* et *getL* et deux *setters*, *setH* et *setL*.

```
1  public class Rectangle {
2    private double hauteur;
3    private double largeur;
4
5    double getH(){return hauteur;} // getter
6    double getL(){return largeur;} // getter
7
8    void setH(double h){hauteur=h;} // setter
9    void setL(double l){largeur=l;} // setter
10
11   double surface()
12   {   double s=hauteur*largeur;
13     return s;
14   }
15 }
```

Pseudo-code 4.1 – Exemple des accesseurs et des manipulateurs

**Exemple 13** Un exemple d'accès aux attributs de la classe *Rectangle* pour modification et lecture dans une méthode *main* est donnée dans le Pseudo-code 4.2.

```
1 public static void main(String[] args) {
2  //Create new object r of class Rectangle
3  Rectangle r=new Rectangle();
4  r.setH(4.3); // modifier la hauteur de r
5  r.setL(5.2); // modifier la largeur de r
6  System.out.println(r.getH());// afficher la valeur de la hauteur
7  System.out.println(r.getL());// afficher la valeur de la largeur
8  System.out.println(r.surface());
9 }
```

Pseudo-code 4.2 – Appelle aux accesseurs et manipulateurs de la classe *Rectangle*

## 4.6 Conclusion

Nous avons vu dans ce chapitre les différents indicateurs de visibilité (accessibilité) des membres d'une classe. Un membre d'une classe peut avoir une visibilité par défaut, ou peut être déclaré *private*, *public*, ou *protected*. Un membre d'une classe déclaré *protected* a une visibilité package et il est accessible aussi par toutes les sous classes héritées de sa classe de déclaration. Ce dernier concept, *héritage*, sera détaillé dans le chapitre suivant.

# 5. Héritage

Sommaire



## 5.1 Introduction

L'héritage est l'un des mécanismes les plus puissants de la POO. Il permet la réutilisation des fonctionnalités d'une classe (appelée superclasse ou classe mère), tout en apportant des variations spécifiques aux classes héritant (appelée sous-classe, classe fille, classe enfant, ou classe dérivée). Une classe enfant obtient les caractéristiques de sa superclasse (classe mère). Cette nouvelle classe hérite des fonctionnalités de la super classe (attributs et méthodes) qu'elle pourra modifier ou compléter à volonté,

sans qu'il soit nécessaire de remettre en question la classe mère.

L'objectif principal de l'héritage est de favoriser la réutilisabilité des classes et d'éliminer la redondance de code. Cette technique offre la possibilité de développer à partir d'une classe de base, autant de classes dérivées qu'on le désire. De même, une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée. Ce chapitre est dédié à présenter le concept d'héritage et ses différents concepts.

## 5.2 Notion d'héritage

Pour illustrer la notion d'héritage, on peut imaginer vouloir représenter les véhicules d'une entreprise. On pourrait créer une classe spécifique à chaque type de véhicule, qui aurait comme attributs, un *matricule*, une *marque*, une *couleur* mais aussi des éléments propres à son identité comme une capacité pour un camion. Ces classes partageraient également des méthodes, telles que $affiche r$ pour afficher les caractéristiques d'un véhicule. Une telle solution dupliquerait beaucoup de code et poserait des problèmes de maintenance.

Il convient bien dans le cas de notre exemple d'établir une super-classe $Vehicule$ ayant comme attribut un *matricule*, une *marque*, et une *couleur* , et une méthode $affiche r$. Les sous-classes $Voiture$ et $Camion$ sont des versions spécifiques de la classe $Vehicule$, dont elles gardent les caractéristiques (voir la Figure 5.1). On dit que ces sous-classes héritent de la classe $Vehicule$. Les sous-classes peuvent avoir des éléments spécifiques comme *capacit* et *volume* pour le $Camion$ ou le fait de *rouler* pour la $Voiture$.

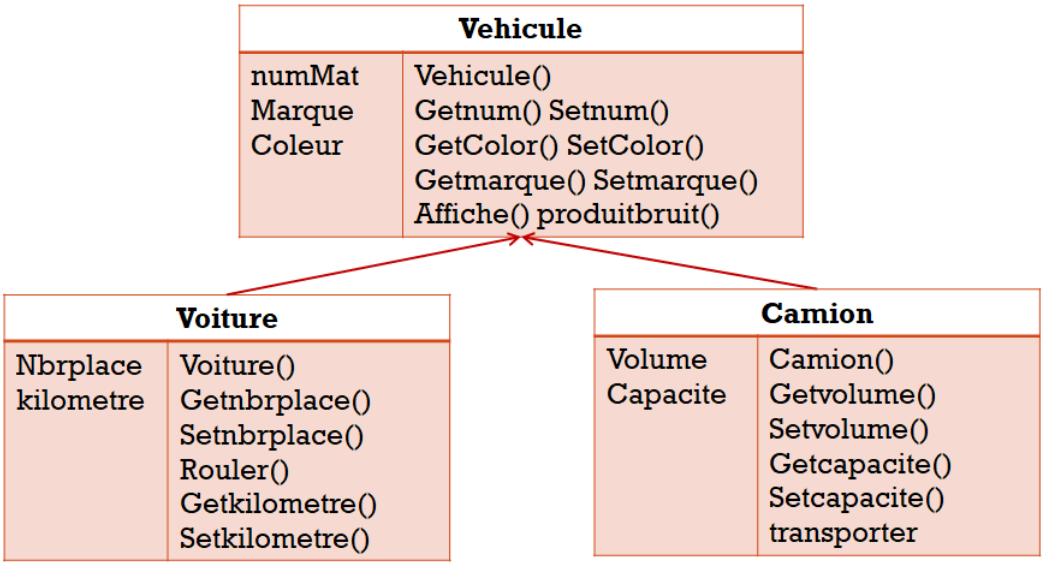



FIGURE 5.1 – Exemple de visibilité de classes, méthodes, et attributs

En POO, l'héritage permet de regrouper des caractéristiques communes dans une super-classe dont héritent des sous-classes qui en sont des versions enrichies, étendues. Ce mécanisme permet d'établir une relation "est-un" : si $B$ hérité de $A$, alors ont dit que "$B$ est un $A$", c.a.d, toute instance de $B$ est aussi une instance de $A$. De ce fait, l'ensemble des attributs et méthodes de $A$ (sauf les constructeurs) sont disponibles dans $B$. $B$ est enrichie par des attributs et des méthodes supplémentaires et spécialisées si elle redéfinit des méthodes héritées de $A$.

## 5.3 Types d'héritage

Quatre types d'héritages sont distingués :

1. Héritage unique (Single inheritance) : Une classe enfant est dérivée d'une classe de base (voir la Figure 5.2(a)).
2. Héritage hiérarchique : Plusieurs classes enfants peuvent être dérivées d'une super-classe (voir la Figure 5.2(b)).
3. Héritage à plusieurs niveaux : la classe parente peut avoir des sous-sous classes (voir la Figure 5.2(c)).
4. Héritage multiple : la classe dérivé peut dériver de plusieurs super-classes (voir la Figure 5.2(d)). Ce type d'héritage n'est pas autorisé par Java.

## 5.4 Héritage simple (extends)

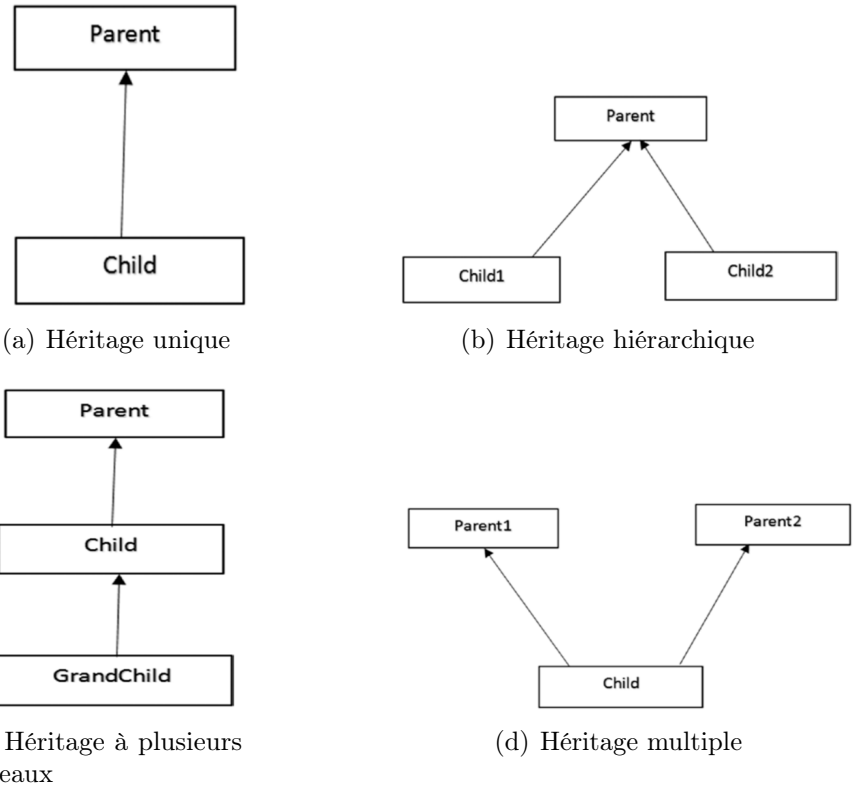
Pour hériter une classe d'une autre classe en Java on utiliser le mot clé `extends` suivi du nom de la super-classe dans la déclaration de la classe dérivée. La syntaxe générale est la suivante :

```
1 public class subclassName extends superclassName {
2
3 }
```

**Exemple 14** Soit les classes de la Figure 5.3. Ci-dessous l'implémentation de ces classes en Java :

```
1   public class A  {
2
3 }
```

```
1   public class B extends A {
2
3 }
```




(a) Héritage unique

(b) Héritage hiérarchique

(c) Héritage à plusieurs niveaux

(d) Héritage multiple

FIGURE 5.2 – Types d'héritage [8]

```
1  public class C extends A {
2
3  }
```

```
1  public class D extends C {
2
3  }
```

**Exemple 15** Reprenant les classes de la Figure 5.1. L'implémentation de ces classes est présentée par le Pseudo-code 5.1 pour la classe *Vehicule*, le Pseudo-code 5.2 pour la classe *Voiture*, et le Pseudo-code 5.3 pour la classe *Camion*.

```
1  public class Vehicule {
2    private int numMat;
3    private String Mark;
4    private int couleur;
5
6    public Vehicule() {numMat=1; Mark="MW"; couleur=0;}
7
8    public Vehicule(int numMat, String Mark, int couleur) {
9      this.numMat = numMat;
10     this.Mark = Mark;
```

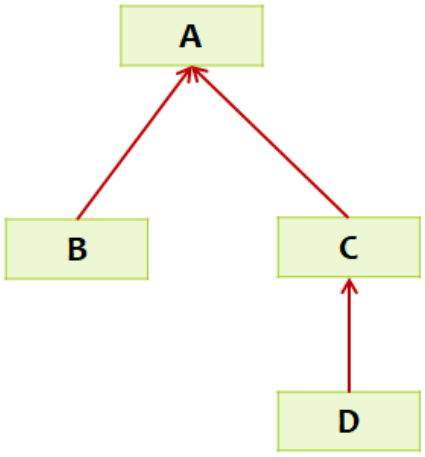



FIGURE 5.3 – Exemple d'une super-classe et des classes filles

```
    this.couleur = couleur;
  }

  public int getNumMat()  { return numMat;}
  public int getCouleur() { return couleur;}
  public String getMark() { return Mark;}

  public void setNumMat(int numMat)   {this.numMat = numMat;}
  public void setMark(String Mark)    {this.Mark = Mark; }
  public void setCouleur(int couleur) {this.couleur = couleur;}

  public void afficher(){
    System.out.print( numMat+" "+Mark+" "+couleur);
  }

  public boolean produitBruit(){return true;}
}
```

Pseudo-code 5.1 – Implémentation de la classe *Vehicule*

```
public class Voiture extends Vehicule{
  private int nbrPlace;
  private double kilometre;

  public Voiture() {nbrPlace=5;}
  public Voiture(int nbrPlace) { this.nbrPlace = nbrPlace;}

  public int getNbrPlace() {return nbrPlace;}
  public void setNbrPlace(int nbrPlace) {this.nbrPlace = nbrPlace;}

  public double getKilometre() {return kilometre;    }
```

```
12   public void setKilometre(double kilometre) {this.kilometre =
      kilometre;}
13
14   public void rouler(double k){kilometre=kilometre+k;}
15 }
```

Pseudo-code 5.2 – Implémentation de la classe *Voiture*

```
1   public class Camion extends Vehicule {
2     private int volume;
3     private int capacite;
4
5     public Camion() {volume=100;capacite=100;}
6
7     public Camion(int volume, int capacite) {
8       this.volume = volume;
9       this.capacite = capacite;
10    }
11
12    public int getVolume() {return volume;}
13    public int getCapacite() {return capacite;}
14
15    public void setVolume(int volume) {this.volume = volume;}
16    public void setCapacite(int capacite){this.capacite=capacite;}
17
18    public void transporer(){
19      System.out.print("Camoin transporte des produits ...");}
20  }
```

Pseudo-code 5.3 – Implémentation de la classe *Camion*

## 5.5 Héritage et niveaux de visibilité

Nous avons vu précédemment (dans la section 4.4) qu'une classe ou un membre d'une classe (classes, attributs, méthodes) peut être déclaré : `private`, `public`, ou peut être `protected`. On a veut aussi que `protected` signifie que l'élément est accessible aux classes dans le même paquetage et à toute classe dérivée en dehors du paquetage et il n'est pas accessible depuis les classes non liées dans d'autres packages. Donc, le modificateur `protected` est significatif lorsqu'il est utilisé avec l'héritage.

**Exemple 16** Soit la classe $A$ du package $P1$ de la Figure 5.4. Dans la classe $A$, l'attribut $z$ est déclaré *protected* et il est visible dans toutes les classe du package $P1$ où la classe $A$ est déclaré. L'attribut $z$ est visible aussi dans la classe $E$ du package $P2$, puisque celle-ci est une classe héritée de la classe $A$.

| Accessible? | x | y | g | z |
|---|---|---|---|---|
| B | X | √ | √ | √ |
| C | X | √ | √ | √ |
| D | X | √ | √ | √ |
| E | X | √ | X | √ |
| F | X | √ | X | X |

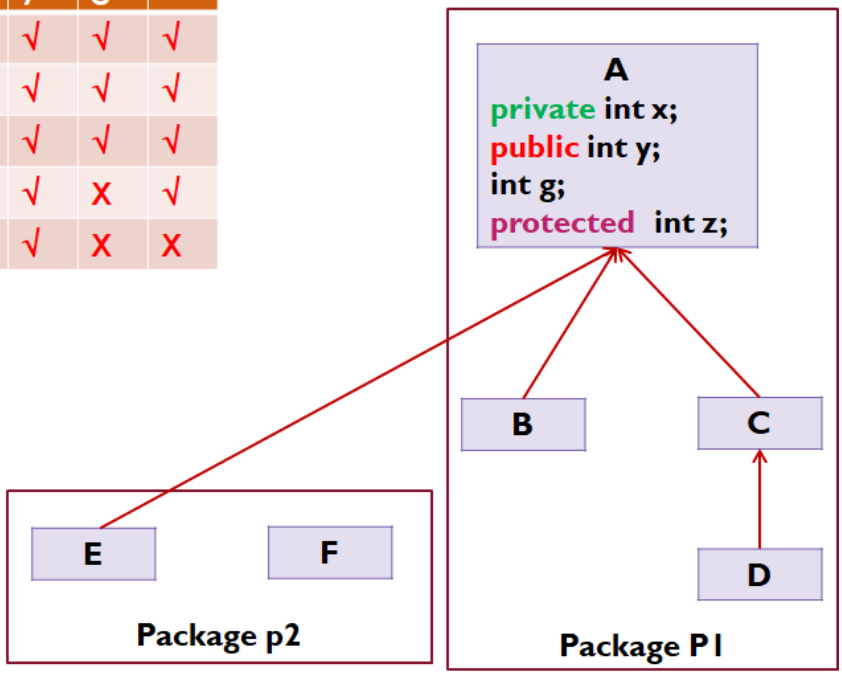



FIGURE 5.4 – Exemple de visibilité *protected*

## 5.6 Spécialisation et masquage

### 5.6.1 Redéfinition des méthodes (Overriding)

Dans certaines situations, on souhaite redéfinir ou modifier le comportement de la classe parent. La spécialisation ou la redéfinition de méthode entre en scène dans un tel scénario. La redéfinition d'une méthode est nécessaire si on désire adapter son action à des besoins spécifiques.

La redéfinition d'une méthode, aussi appelée "*overriding*", consiste à donner une nouvelle implémentation à une méthode héritée sans changer sa signature (même nom de méthode, même type de retour, même paramètre (nombre, ordre, et type)). La méthode héritée est alors dite la méthode générale, appelée par les sousclasses qui ne la redéfinissent pas. La méthode spécifique à la sous-classe est la méthode spécialisée.

R Ne pas confondre la redéfinition et la surcharge des méthodes :

- ☆ On surcharge une méthode quand une nouvelle méthode avec le même nom, mais pas la même signature qu'une autre méthode de la même classe ;
- ☆ 0n redéfinir une méthode quand on donne une nouvelle implémentation, dans la classe fille, d'une méthode héritée (de la super-classe) sans changer sa signature.

**Exemple 17** On veut redéfinir la méthode *produitBruit* de la classe *Vehicule* du Pseudo-code 5.1 dans la classe *Voiture* (Pseudo-code 5.2), pour que la méthode renvoie *true* au lieu de *false* (voir le Pseudo-code 5.4).

```
1  public class Voiture extends Vehicule {
2    private int nbrPlace;
3    private double kilometre;
4
5    ... // Code reste tel qu'il est
6
7    @Override
8    public boolean produitBruit() {return false;}
9  }
```

Pseudo-code 5.4 – Redéfinition de la méthode *produitBruit* dans la classe *Voiture*

R On peut aussi redéfinir une méthode en gardant la même implémentation que celle de la super-classe et en y ajoutant d'autres spécifications. Pour ce faire, et pour éviter la duplication du code, on utilise le mot clé `super`. Dans ce cas ce me clés est utilisé pour appeler la méthode héritée : `super.methodeName();`. Cette écriture établit le lien entre le nom du membre (méthode ou attribut) et la classe à laquelle il appartient, et permet d'éviter des duplications de code (évidement, réutilisation du code) lors de la spécialisation de méthodes.

**Exemple 18** On veut redéfinir la méthode *afficher* de la classe *Vehicule* du Pseudo-code 5.1 dans la classe *Voiture* (Pseudo-code 5.2), pour que la méthode affiche aussi le nombre de place et le kilométrage d'une voiture (voir le Pseudo-code 5.5). Dans ce code, la méthode *afficher*, affiche le matricule, la marque, la couleur, nombre de place, et le kilométrage d'une voiture. On peut aussi faire la même chose dans la classe *Camion* (voir le Pseudo-code 5.6).

```
1  public class Voiture extends Vehicule {
2    private int nbrPlace;
3    private double kilometre;
4    ... // Code reste tel qu'il est
5    @Override
6    public boolean produitBruit() {return false;}
7
8     @Override
9    public void afficher() {
10     super.afficher();
11   //appelle de la méthode afficher de la super-classe  Vehicule
12     System.out.print(nbrPlace+" "+kilometre);
13   }
14 }
```

Pseudo-code 5.5 – Redéfinition de la méthode *afficher* dans la classe *Voiture*

```
1  public class Camion extends Vehicule {
2    private int volume;
3    private int capacite;
4  ... // Code reste tel qu'il est
5
6  @Override
7  public void afficher() {
8    super.afficher();
9    System.out.print(capacite+" "+volume);
10 }}
```

Pseudo-code 5.6 – Redéfinition de la méthode $affiche r$ dans la classe $Camion$

**Exemple 19** Un autre exemple de redéfinition des méthodes et le mot clés *super* est illustré dans la Figure 5.5. Dans cette figure on trouve ce qui suit :

- ☆ Une classe $B$ qui comporte une méthode $f$ ;
- ☆ Une classe $C$ héritée de la classe $B$. Dans la classe $C$, la méthode $f$ est redéfinie pour qu'elle renvoie la valeur de la méthode $f$ du $B$ additionnée à 5 $(super.f + 5)$ ;
- ☆ Une classe $D$ héritée de la classe $B$. Dans la classe $D$, la méthode $f$ est redéfinie pour qu'elle renvoie la valeur de la méthode $f$ du $B$ multipliée par 5 $(super.f \times 5)$ ;
- ☆ Une méthode *main*, où trois instances $a$, $c$, et $d$ sont déclarées de type, respectivement, $B$, $C$, et $D$. Chaque déclaration d'instance est suivie par un appelle et affichage du résultat de la méthode $f$ correspondante ;
- ☆ Le résultat d'affichage de la méthode $f$ de chaque instance.

### 5.6.2 Masquage de données

Il existe aussi ce que l'on appelle le *masquage*, qui est la déclaration d'un attribut dans une classe fille ayant le même nom qu'un attribut d'une classe mère. Le masquage est souvent source de confusion et est peu courant : il consiste à nommer un attribut par le même nom (pas forcément le même type) qu'un attribut hérité, qui masquera ce dernier. Au contraire, redéfinir des méthodes est courant et permet de les spécialiser.

Pour résoudre la confusion des attributs de même de nom de la classe fille et sa super-classe, on utilise le mot clé `super.attributName` dans la classe fille. Cette écriture faite référence à l'attribut de la super-classe et établit un lien entre le nom de l'attribut et la classe à laquelle il appartient. Pour faire référence à l'attribut de la classe fille on utilise juste le nom de l'attribut.

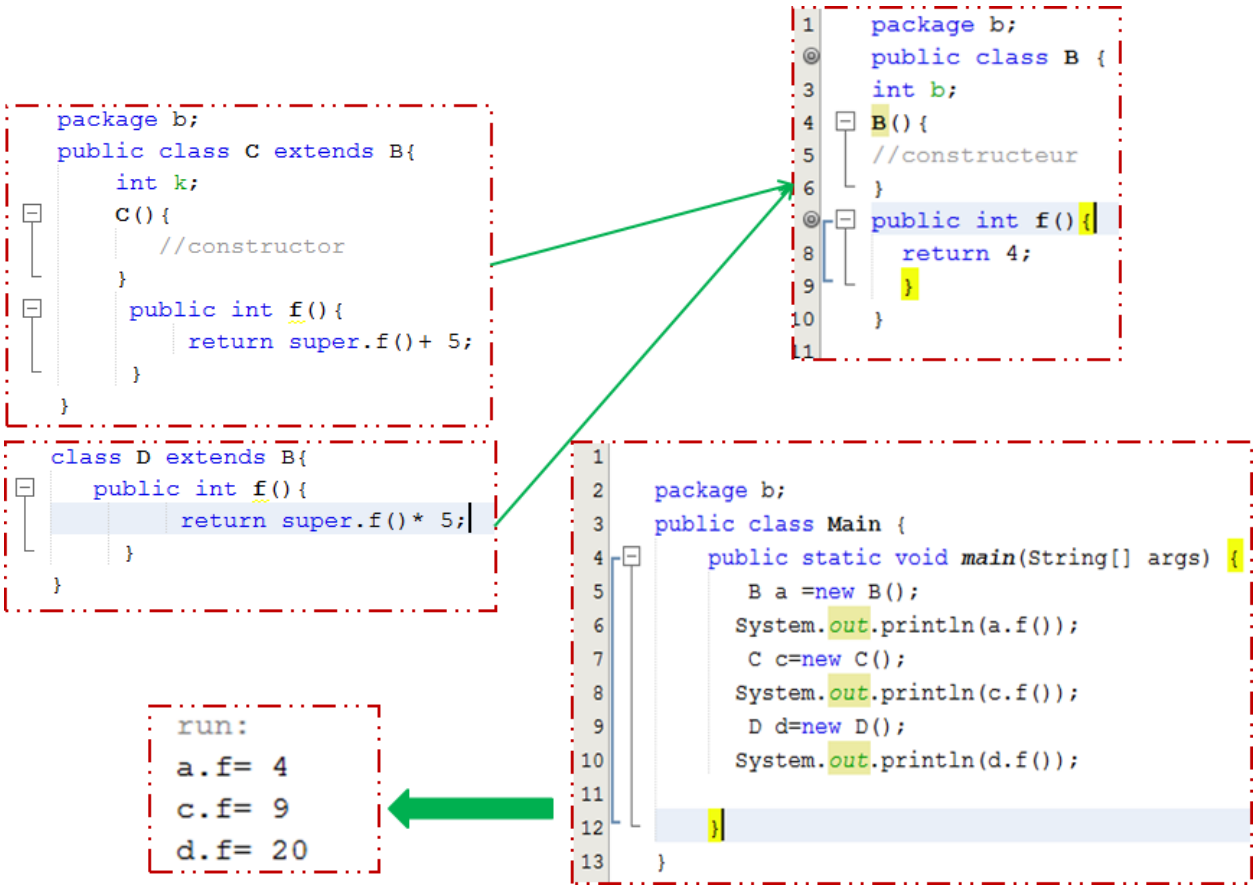



FIGURE 5.5 – Exemple de la redéfinition des méthodes

**Exemple 20** Un exemple de masquage de données et le mot clés *super* est illustré dans la Figure 5.6. Dans cette Figure on trouve ce qui suit :

- ☆ Une classe $B$ qui comporte attribut $b$, initialisé à 6, et une méthode $f$ qui renvoie la valeur de $b$ ;
- ☆ Une classe $C$ héritée de la classe $B$ et comporte aussi d'un attribut nommé $b$, initialisé à 4. L'attribut $b$ de la classe $C$ masque (hides) l'attribut $b$ de la classe $B$. Dans la classe $C$, la méthode $f$ est redéfinie pour qu'elle renvoie la valeur de l'attribut $b$. ;
- ☆ Une classe $D$ héritée de la classe $B$ et comporte aussi d'un attribut nommé $b$, initialisé à 2. L'attribut $b$ de la classe $D$ masque (hides) l'attribut $b$ de la classe $B$. Dans la classe $D$, la méthode $f$ est redéfinie pour qu'elle renvoie la valeur de l'attribut $b$ de la super-classe $D$ ;
- ☆ Une classe $E$ héritée de la classe $B$ et comporte aussi d'un attribut nommé $b$, initialisé à 2. L'attribut $b$ de la classe $E$ masque (hides) l'attribut $b$ de la classe $B$. Dans la classe $E$, la méthode $f$ est redéfinie pour qu'elle renvoie la

valeur de l'attribut $b$ de la super-classe $B$ ($super.b$) la valeur de l'attribut $b$ de la classe $E$ ;

- ☆ Une méthode *main*, où quatre instances $a$, $c$, $d$, et $e$ sont déclarées de type, respectivement, $B$, $C$, $D$, et $E$. Chaque déclaration d'instance est suivie par un appelle et affichage du résultat de la méthode $f$ correspondante ;
- ☆ Le résultat d'affichage de la méthode $f$ de chaque instance.

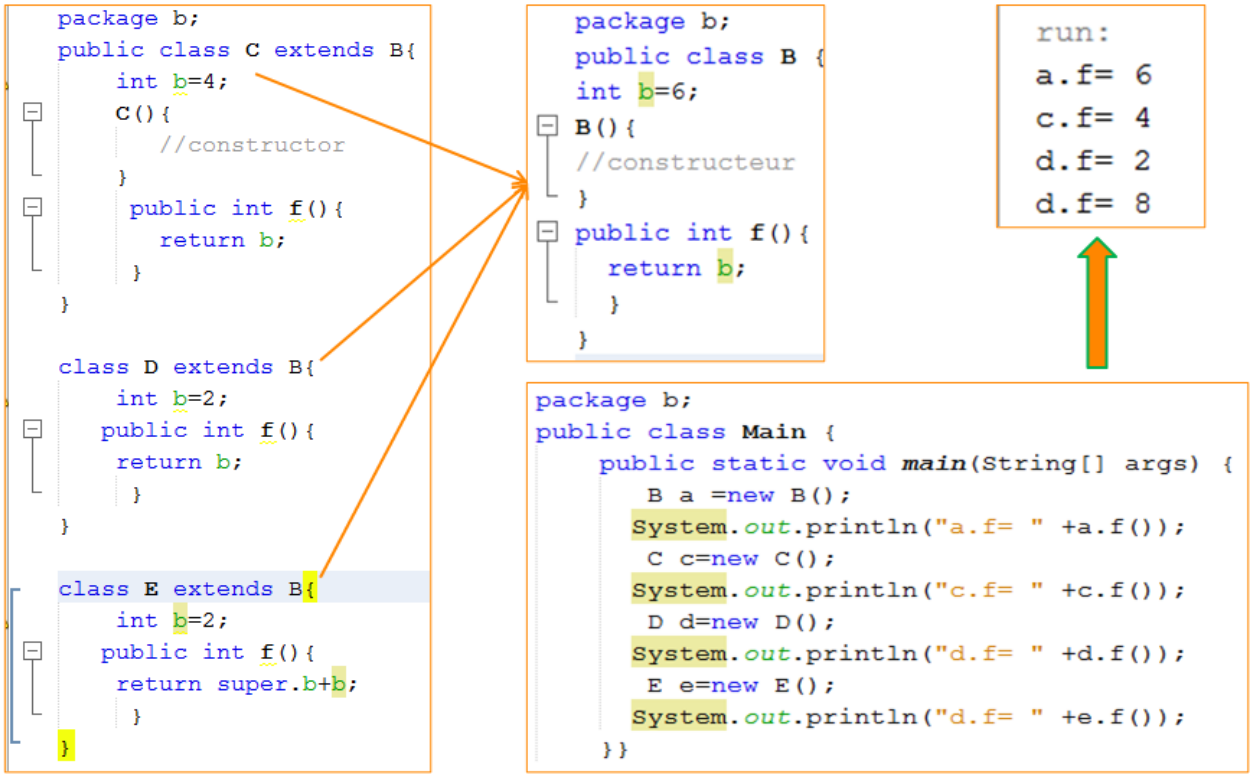



FIGURE 5.6 – Exemple de masquage de données

## 5.7 Constructeurs

Contrairement aux autres membres d'une super-classe, les constructeurs d'une super-classe ne sont pas hérités par ses sous-classes. Cela signifie qu'on doit définir un constructeur pour une classe fille ou utiliser le constructeur par défaut ajouté par le compilateur. Donc, chaque attribut doit être initialisé dans la classe où il est explicitement définit.

Un constructeur d'une sous-classe peut faire appel au constructeur de la super classe, utilisant l'instruction `super()` ou `super(args)`. Si le constructeur définit d'une sous classe ne contient pas un appel explicite à un constructeur de la superclasse, le compilateur ajoute l'instruction `super()` comme première instruction du constructeur.

**Exemple 21** Si on définit un constructeur comme :

```
class MyClass {
  private int myInt;
  public MyClass() {
    myInt = 10;
  }
}
```

alors, le compilateur réécrira (implicitement) le constructeur comme suit :

```
public MyClass() {
  super();
  myInt = 10;
}
```

R Nous avons vu précédemment que lorsqu'un constructeur est déclaré, aucun constructeur par défaut n'est ajouté à la classe. Par exemple, si on définit une classe comme suit :

```
class MyClass {
  public MyClass(int x) { ... }
}
```

Une déclaration telle que : `MyClass test = new MyClass();`, est non valide. Puisque la classe *MyClass* n'a pas de constructeur correspondant.

**Exemple 22** Prenons un autre exemple. Considérant la définition de la classe suivante :

```
public class Vehicule {
  private int numMat;
  private String Mark;
  private int couleur;

  public Vehicule(int numMat, String Mark, int couleur) {
    this.numMat = numMat;
    this.Mark = Mark;
    this.couleur = couleur;
  }
//.....
}
```

Puisque la classe *Vehicule* possède un constructeur paramétré, aucun constructeur par défaut (sans paramètres) n'est ajouté à la classe. Cela signifie une déclaration telle que : `Vehicule v = new Vehicule();` provoque une erreur de compilation.

Considérons maintenant la définition d'une sous-classe *Camion* comme suit :

```
1     public class Camion extends Vehicule {
2       private int volume;
3       private int capacite;
4
5       // ne comporte aucun constructeur
6
7       public int getVolume() {return volume;}
8       public int getCapacite() {return capacite;}
9
10      public void setVolume(int volume) {this.volume = volume;}
11      public void setCapacite(int capacite){this.capacite=
    capacite;}
12
13      public void transporer(){
14        System.out.print("Camoin transporte des produits ...");}
15    }
```

La compilation de cette définition, provoque une erreur du compilateur. Puisqu'aucun constructeur n'est défini pour la classe, le compilateur ajoute un constructeur par défaut (implicitement) comme suit :

```
1     public class Camion extends Vehicule {
2       private int volume;
3       private int capacite;
4
5       Camion(){ super(); }
6
7       // .....
8     }
```

Ce constructeur appelle le constructeur de la super-classe sans arguments, mais il n'y a pas de constructeur correspondant dans la super-classe. Ainsi, une erreur de compilation se produit. Voici une définition correcte de la classe *Camion* :

```
1     public class Camion extends Vehicule {
2       private int volume;
3       private int capacite;
4
5       public Camion(){ super(100,"Volvo", 3); }
6
7         // et/ou:
8
9       public Camion(int m, String k; int c){
10        super(m,k, c); }
11    // .....
12    }
```

R

- ☆ Il est recommandé de définir toujours un constructeur pour chaque classe créée et ne pas compter sur les constructeurs par défaut.
- ☆ Si une classe a une super-classe, introduire un appel explicite à un constructeur de la super-classe dans le constructeur de la sous classe.

## 5.8 Ordre des constructeurs

Dans une relation d'héritage à plusieurs niveaux, la construction d'une sous-classe commence par appeler le constructeur de la super-classe la plus générale puis, dans l'ordre, les constructeurs des super-classes qui en héritent, avant de finir par l'initialisation de la partie spécifique de la classe instanciée.

**Exemple 23** Supposant une classe $C$ héritant d'une classe $B$ qui elle-même hérité d'une classe $A$ (ce lien d'héritage est illustré par la Figure 5.7). La construction d'une instance de $C$, `C monC= new C(...)`, appelle, explicitement ou non, le constructeur de $B$ qui lui même appelle celui de $A$. Comme ces appels sont placés en premier lieu dans la liste d'initialisation :

- ☆ le constructeur de $C$ va commencer par exécuter la construction de sa partie définit dans la classe $A$ (classe la plus générale dont on dérive), en donnant des valeurs à ses attributs $a1$ et $a2$ ;
- ☆ Dés que Cette première étape est terminée, le constructeur de $B$ initialise $b1$ et construit ainsi la partie $B$ de l'instance $monC$.
- ☆ Enfin, le constructeur de $C$ peut initialiser la partie spécifique à cette sous-classe, les attributs $c1$ et $c2$, ce qui termine la construction de l'instance $monC$.

**Exemple 24** Prenons un autre exemple. Le Pseudo-code 5.7 contient un extrait de code de trois classes, $A$, $B$, et $C$, tel que : $A$ est la classe générale (super-classe), $B$ est hérité de $A$, et $C$ est hérité de $B$. Chacune de ces classes contient un seul constructeur qui affiche une salutation :

- ☆ Le constructeur de la classe $A$ affiche "Hello ..." ;
- ☆ Le constructeur de la classe $B$ affiche "Bonjour ..." ;
- ☆ Le constructeur de la classe $A$ affiche "Hola ...".

La méthode *main* contient une seule instruction : la déclaration d'une instance $c$ de la classe $C$. Cet extrait de code se termine par le résultat d'exécution de la méthode *main*. Les trois salutations sont affichées, car le constructeur de la classe

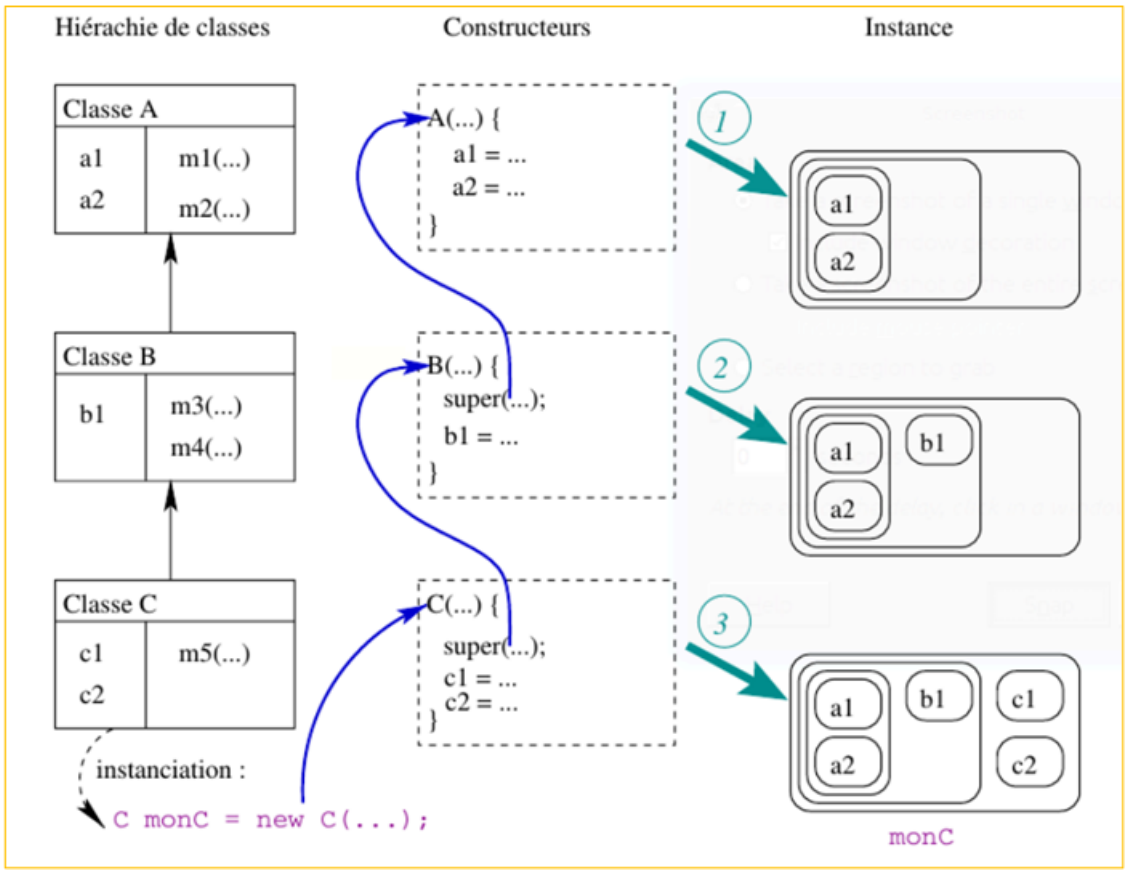



FIGURE 5.7 – Illustration de l'ordre d'appel des constructeurs dans une relation d'héritage à plusieurs niveaux [7].

$C$ appelle implicitement celui de la classe $B$, qui lui même appelle implicitement le constructeur de son super-classe $A$.

```
1  public class A {
2    public A() { System.out.println("Hello ...");}
3    }
4
5  class B extends A {
6    public B() { System.out.println("Bonjour ...");}
7    }
8
9  class C extends B {
10   public C() { System.out.println("Hola ...");}
11   }
12 // Méthode main
13  public static void main(String[] args)
14 {
15   // Création d'u,e instance c de type C
16   C c=new C();
```

```
}

// Résultat d'exécution :
  Hello ...
  Bonjour ...
  Hola ...
```

Pseudo-code 5.7 – Exemple de l'ordre d'appel des constructeurs dans une relation d'héritage à plusieurs niveaux

R Dans une relation d'héritage à plusieurs niveaux en Java, on ne peut accéder aux membres des grands-parents que via la classe parent. Donc, l'écriture `super().super()` est non autorisée en Java.

## 5.9 Limitation d'héritage (final)

Dans la POO, on peut désactiver une classe d'être héritée par d'autres classes. Comme on peut aussi limiter une méthode d'être redéfinie ou masquée par des sous classes. Pour ce faire, on utilise mot clé `final` dans la déclaration de la classe et la définition de la méthode.

Les principales raisons pour limiter l'héritage des classes et des méthodes sont la sécurité, l'exactitude et les performances. Si une classe est importante pour des raisons de sécurité, on ne peut éviter que quelqu'un hérite cette classe et perturbe la sécurité que la classe est censée implémenter. Parfois, on déclare une classe/méthode finale pour préserver l'exactitude du programme. Une méthode finale peut entraîner de meilleures performances au moment de l'exécution.

**Exemple 25** L'extrait de code suivant déclare une classe finale nommée $A$ :

```
public final class A {
    // Code goes here
}

```

La déclaration suivante de la classe $B$ ne sera pas compilée :

```
public  class B extends A{
  // Code goes here
}
```

**Exemple 26** L'extrait de code suivant déclare une méthode finale nommée $f$ dans la classe $A$ :

```
1  public  class A {
2
3   public final void f() {
4    // Code goes here
5   }
6   public void f2() {
7    // Code goes here
8   }
9  }
```

Dans la classe $B$ suivante, on ne peut pas redéfinir ou masquer la méthode $f$, car elle est déclarée $final$ dans la classe $A$. Par contre, on peut redéfinir ou masquer la méthode $f2$.

```
1 public class B extends A {
2
3  // Cannot override A.f() here because it is final in class A
4  // OK to override f2() because it is not final in class A
5
6  public void f2() {
7    // Code goes here
8  }
9 }
```

Il ya de nombreuses classes et méthodes dans les bibliothèques de classes Java déclarées finales. Plus particulièrement, la classe *String* est une classe finale.

## 5.10 Conclusion

Le concept d'héritage constitue l'un des fondements de la POO. Il offre notamment des possibilités de réutilisation des composants logiciels que sont les classes. Dans la POO il y a un autre concept qui est aussi fondamental, et qui est considéré également comme étant un complément d'héritage : le *polymorphisme*. Ce concept sera présenté dans le chapitre suivant.

# 6. Polymorphisme

Sommaire



## 6.1 Introduction

Le polymorphisme s'agit d'un concept extrêmement puissant en programmation orientée objet, indispensable pour une utilisation efficace de l'héritage. La puissance de polymorphisme réside dans la possibilité de manipulation des objets sans en connaître (tout à fait) le type. Il existe de nombreux types de polymorphisme. Parmi ces types, le polymorphisme d'héritage est le plus important dans la POO. Dans ce chapitre, nous allons introduire les différents types de polymorphisme ainsi que des concepts liés au polymorphisme.

## 6.2 Concept polymorphisme

Le mot "*polymorphisme*" trouve sa racine dans deux mots grecs : "*poly*" (signifie plusieurs) et "*morphos*" (signifie forme). En programmation, le polymorphisme est la

capacité d'une entité (par exemple, une variable, une classe, une méthode, objet, code, paramètre, etc.) de prendre différentes significations selon les contextes. L'entité qui prend différentes significations est appelée entité polymorphe (polymorphic entity).

Le but du polymorphisme est d'écrire des codes réutilisables et maintenables en écrivant des codes en termes d'un type générique qui fonctionne pour de nombreux types (ou idéalement pour tous les types). En d'autre terme, Le polymorphisme permet à un même code de s'adapter aux types de données auxquels il s'applique. Ainsi, il rend le code générique, écrit de façon unifiée, pour différents types de données.

## 6.3 Types de polymorphisme

Différents types de polymorphisme existent. Chaque type de polymorphisme porte un nom qui indique généralement comment ce type de polymorphisme est obtenu en pratique. Le polymorphisme peut être classé en deux catégories [9] (voir la Figure 6.1) : (1) Polymorphism ad hoc, et (2) Polymorphisme générique.

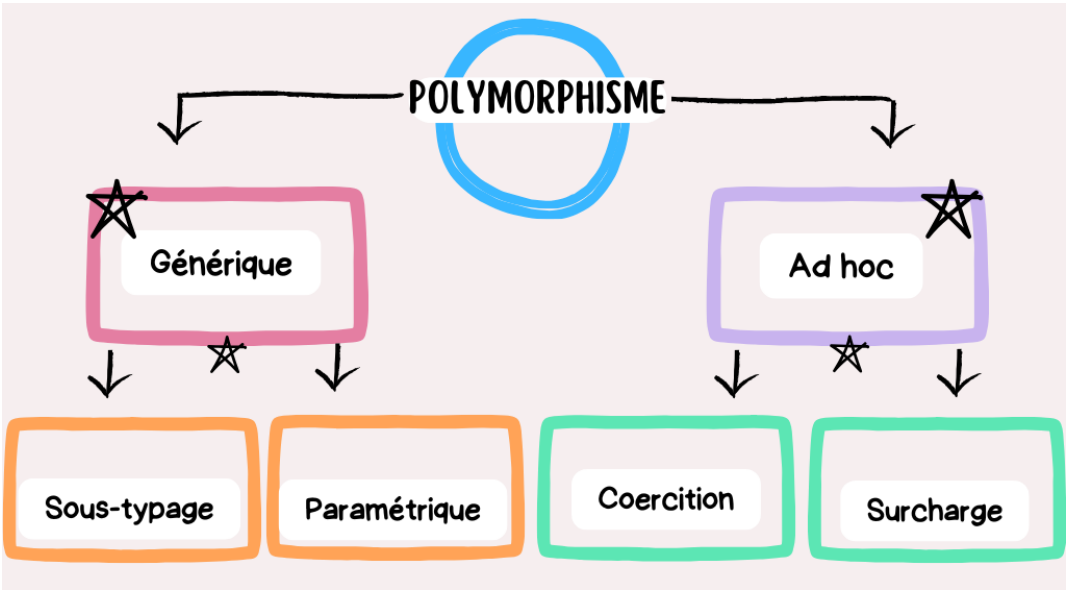



FIGURE 6.1 – Types de polymorphisme.

### 6.3.1 Polymorphisme ad hoc :

Si un code fonctionne pour un nombre fini de types et que tous ces types doivent être connus lors de l'écriture du code, on parle de polymorphisme ad hoc. Le polymorphisme ad hoc est également appelé apparent polymorphisme, parce que ce n'est pas du polymorphisme au vrai sens du terme. Le polymorphisme ad hoc est divisé en deux catégories : (1) Polymorphisme de surcharge, (2) Polymorphisme de

coercition.

**Polymorphisme de surcharge :** ce type de polymorphisme signifie la surcharge des résultats lorsqu'une méthode ou un opérateur a au moins deux définitions qui fonctionnent sur des types différents. Dans un tel cas, le même nom (pour la méthode ou l'opérateur) est utilisé pour différentes définitions. Autrement dit, le même nom présente de nombreux comportements et donc le polymorphisme. Ces méthodes et opérateurs sont appelés "*méthodes surchargées*" et "*opérateurs surchargés*". Java permet la surcharge des méthodes comme il a des opérateurs surchargés. Par contre, Java ne permet pas de fournir une nouvelle définition d'un opérateur.

**Exemple 27** L'extrait de code suivant montre un exemple de surcharge d'opérateur en Java, l'opérateur est + :

```
1  int a = 100 + 9; // Adds two integers
2  double b = 30.5 + 8.34; // Adds two floating-point numbers
3  String str = "Hello " + "everybody"; // Concatenates two strings
4
```

**Exemple 28** L'extrait de code suivant montre un exemple de surcharge de méthodes en Java :

```
1      public class Compute {
2      public static int somme(int a, int b) {
3      //Code to determine the sum of two integers goes here
4      }
5      public static double sum(double a, double b) {
6      //Code to determine the sum of two floating numbers goes here
7      }
8      public static int somme(int[] tab) {
9      //Code to determine the sum in an array of int goes here
10     }
11   }
```

La méthode *somme* de la classe *Compute* est surchargée. Elle a trois définitions et chacune de ces définitions effectue la même tâche de calcul de la somme, mais sur des types différents. La première définition calcule la somme de deux entiers, la deuxième calcule la somme de deux nombres réels, et la troisième calcule la somme des élément d'un tableau des entiers.

**Polymorphisme de coercition :** la coercition se produit lorsqu'un type est implicitement converti automatiquement en un autre type, même si cela n'était pas explicitement prévu. Les langages de programmation (y compris Java) effectue différents types de coercition dans différents contextes : affectation, paramètres de

méthode, etc.

**Exemple 29** l'extrait de code suivant, présente des conversions explicite et implicite de *int* en *double*.

```
1
2  int num = 890;
3  double a = (double)num; // Explicit conversion of int to double
4  double b = num; // Implicit conversion of int to double (coercion
    )
```

**Exemple 30** Considérant l'extrait de code suivant qui montre une définition d'une méthode *square*(), qui prend un paramètre de type double :

```
1    double square(double num) {
2      return num * num;
3    }
```

La méthode *square*() peut être appelé avec un paramètre réel de type *double*, comme suit :

```
1    double a = 12.35;
2  double result = square(a);
```

La même méthode *square*() peut également être appelée avec un paramètre type *int*, comme suit :

```
1  int b = 9;
2  double result = square(b);
```

Dans ce cas, la méthode *square*() est appelée méthode polymorphe. La méthode *square*() est polymorphe en raison de la conversion de type implicite (coercition de *int* en *double*) fournie par le langage Java.

R Supposons que $f$ soit une méthode qui déclare un paramètre formel de type $T$. Si $S$ est un type qui peut être implicitement converti en $T$, la méthode $f$ est dite polymorphe par rapport à $S$ et $T$.

### 6.3.2 Polymorphisme générique (universel)

Un code est dit universellement polymorphe s'il fonctionne sur un nombre infini de types. Le polymorphisme générique comporte deux types : (1) Polymorphisme par sous-typage (héritage), et (2) Polymorphisme paramétrique.

**Polymorphisme par sous-typage (héritage) :** Le polymorphisme par sous-typage (ou héritage) s'agit du type de polymorphisme le plus courant pris en charge par les langages de programmation orientés objets, y compris Java. Ce polymorphisme se produit lorsqu'un code écrit à l'aide d'un type fonctionne pour tous ses sous-types.

Ce type de polymorphisme est possible sur la base de la règle de sous-typage selon laquelle une valeur qui appartient à un sous-type appartient également au supertype.

Supposons que $T$ soit un type et que $S1$, $S2$, et $S3$ soient des sous-types de $T$. Une valeur qui appartient à $S1$, $S2$, et $S3$ appartient également à $T$. Cette règle de sous-typage permet d'écrire le code suivant :

```
1  T t;
2  S1 a;
3  S2 b;
4  ...
5  t= a;// A value of type a can be assigned to a variable of type T
6  t= b;// A value of type b can be assigned to a variable of type T
```

Java prend en charge ce type de polymorphisme en utilisant l'héritage, qui est un mécanisme de sous-typage. On peut définir une méthode en Java en utilisant un paramètre formel d'un type, par exemple $Vehicule$ (définit dans le Pseudo-code 5.1), et cette méthode peut être appelée sur tous ses sous-types, par exemple $Voiture$ (définit dans le Pseudo-code 5.2) et $Camion$ (définit dans le Pseudo-code 5.3).

**Exemple 31** Supposant la méthode $Acheter$ prend un paramètre formel de type $Vehicule$ comme suit :

```
1 void Acheter(Vehicule v){
2 System.out.print(v.getNumMat()+"est achetée") }
```

La méthode $Acheter$ fonctionnera aussi pour les sous-classes de la classe $Vehicule$. Donc, on peut écrire un code comme ceci :

```
1   Vehicule bmw1 = new Vehicule();
2   Voiture bmw2  = new Voiture();
3   Camion chman1   = new Camion();
4 Acheter(x);// Use the Vehicule type
5 Acheter(a);// Use the Voiture type, which is a subclass of Vehicule
6 Acheter(b);// Use the Camion type, which is a subclass of Vehicule
```

R Lorsqu'une variable (telle que $bmw1$) est déclarée de type classe $S$ (telle que $Vehicule$), la variable peut être une référence à une instance de $S$ ou à l'une de ses sous-classes (telles que $Voiture$ et $Camion$). L'inverse n'est pas valide.

Dans le polymorphisme par sous-typage, le nombre de types pour lesquels le code fonctionne est contraint mais infini. La contrainte est que tous les types doivent être un sous-type du type de création de code. S'il n'y a aucune restriction sur le nombre de sous-types qu'un type peut avoir, le nombre de sous-types est infini (du moins en théorie). Le polymorphisme par sous-typage permet non seulement la création des code réutilisables, mais également la création des codes extensibles et flexibles. La

méthode *Acheter* fonctionne sur toutes les sous-classes de la classe $Vehicule$. Cette méthode continuera à fonctionner pour toutes les sous-classes de la classe $Vehicule$, qui seront définies dans le futur, sans aucune modification.

R Java utilise d'autres mécanismes, comme la redéfinition et la spécialisation des méthodes, ainsi que des règles d'héritage pour rendre le polymorphisme par sous-typage plus efficace et utile.

**Polymorphisme paramétrique :** Le polymorphisme paramétrique s'appelle aussi "*vrai polymorphisme*" car il permet d'écrire un vrai code générique qui fonctionne pour tous les types de données (liés ou non). Dans le polymorphisme paramétrique, un code est écrit de sorte qu'il fonctionne sur n'importe quel type de données.

Le polymorphisme paramétrique est obtenu en utilisant une variable de type (liste de données par exemple) lors de l'écriture du code, plutôt qu'en utilisant un type spécifique. La variable type suppose un type spécifique pour lequel le code doit être exécuté. Java prend en charge le polymorphisme paramétrique depuis Java 5 via les génériques. Java prend en charge les entités polymorphes (par exemple, les classes paramétrées) ainsi que les méthodes polymorphes (méthodes paramétrées) qui utilisent le polymorphisme paramétrique.

R La classe `ArrayList<T>` et l'interface `List` sont une classe générique et une interface générique supportant un type paramétré. Une variable de type est une variable définie dans une section entre <>, par exemple $T$ dans `ArrayList<T>`.

**Exemple 32** Il est possible de définir une même fonction *concat* permettant de concaténer deux listes quel que soit le type de données qu'elles contiennent. Une variable de type `List` est une variable de type qui correspond à n'importe quel type, donc *concat* est une fonction qui supporte le polymorphisme paramétré.

En Java, le polymorphisme paramétrique est obtenu en utilisant des génériques. Tous les types de collections en Java utilisent des génériques.

**Exemple 33** Le Pseudo-code 6.1 utilise des génériques. Ce code utilise une variable $List1$, une liste de données de type $String$ et une variable $List2$, une liste de données de type $Integer$. À l'aide de génériques, on peut traiter un objet List comme une liste de n'importe quel type en Java.

```
// Create a List of Strings
List<String> List1 = new ArrayList<String>();
// Add two Strings to the List
List1.add("one 1");
List1.add("two 2");
// Get the first String from the List
String st = List1.get(0);

// Create a List of Integers
List<Integer> List2 = new ArrayList<Integer>();
// Add two Integers to the list
List2.add(12);
List2.add(18);
// Get the first Integer from the List
int a = List2.get(0);
```

Pseudo-code 6.1 – Exemple de types génériques en Java

## 6.4 Mot clé *instanceof*

L'opérateur `instanceof` permet de vérifier si une référence d'objet est une instance d'une certaine classe ou interface. Il s'applique aux références d'objets, mais ne sert pas à comparer deux objets entre eux mais simplement à en déterminer la classe d'instanciation ou l'interface d'implémentation. L'opérateur retourne `true` si l'objet désigne effectivement une instance de la classe ou d'une classe dérivée de cette dernière, sinon il retourne toujours `false`. La syntaxe d'utilisation de cet opérateur est comme suit :

```
    objectName instanceof className
```

`a instanceof B` renvoi `true` si :

1. la variable $a$ stocke une référence à un objet de type $B$ ;
2. la variable $a$ stocke une référence à un objet dont la classe hérite de $B$ ;
3. la variable $a$ stocke une référence à un objet qui implémente l'interface $B$.

Sinon, l'opérateur `instanceof` renvoie `false`. Si l'objet $a$ est égal à `null`, dans tous les cas, le résultat est `false`.

**Exemple 34** Le Pseudo-code 6.2 montre un exemple d'utilisation de l'opérateur *instanceof*. Ce code commence par la création des instances de différentes classes ($Vehicule$, $Camion$, et $Voiture$) et les mettre dans le tableau $T$. Par la suite, le code affiche si chaque élément de $T$ est une instance de la classe $Vehicule$, $Camion$, ou $Voiture$. Le code termine par compter le nombre des camions dans $T$.

```
Vehicule v    = new Vehicule();
Vehicule v1   = new Camion();
Vehicule v2   = new Voiture();
Voiture car1  = new Voiture();
Camion truck1 = new Camion();

Vehicule [] T = {v,v1,v2,car1,truck1};

for(int i=0;i<T.length;i++)
System.out.print(T[i] instanceof Vehicule +" ");
  //true true true true true

for(int i=0;i<T.length;i++)
System.out.print(T[i] instanceof Voiture+" ");
  //false false true true false

for(int i=0;i<T.length;i++)
System.out.print(T[i] instanceof Camion+" ");
  //false true false false true

int k=0;
for(int i=0;i<T.length;i++)
  if(T[i] instanceof Camion) k++;
System.out.println("Number of truck= "+k);// 2
```

Pseudo-code 6.2 – Exemple d'utilisation de l'opérateur instanceof

## 6.5 Conclusion

Le polymorphisme permet de traiter les instances à un niveau plus élevé d'abstraction, où ils sont considérés comme des instances d'une super classe ou interface commune. Cette séparation entre l'implémentation spécifique et le comportement général des instances permet d'écrire le code d'une manière plus modulaire et flexible.

Bien que le polymorphisme soit fortement associé à l'héritage, il n'est pas uniquement dépendant de lui. Le polymorphisme peut également être obtenu par le biais d'interfaces et de classes abstraites, où plusieurs classes non liées implémentent une interface commune ou étendent la même classe abstraite. Ces deux concepts sont aussi des concepts primordiaux de la programmation orientée objet et qu'on va entamer dans le chapitre suivant.

# 7. Classes abstraites & Interfaces

## Sommaire



## 7.1 Introduction

Parfois, on peut créer une classe uniquement pour représenter un concept plutôt que pour représenter des objets. Java permet de créer une classe dont les objets ne peuvent pas être créés. Son but est simplement de représenter une idée commune aux objets d'autres classes. Dans la programmation orientée objet, ce principe peut être mis en oeuvre par la création des *classes abstraites* ou par la définition des *interfaces*. Une classe abstraite est une classe de base qui servira uniquement à hériter des classes complémentaires. Une interface est une classe abstraite utilisée uniquement pour désigner un ensemble de fonctionnalités sans implémentation et qui seront définies seulement dans des sous classes. Dans, ce chapitre on va aborder ces deux concepts

avec des exemples illustratifs (pour plus de détails sur les extraits de code de ce chapitre, le lecteur peut se référer à la référence [9]).

## 7.2 Classes abstraites

Les classes abstraites permettent de définir des concepts génériques qui sont communs à toutes les sous-classes mais qui sont trop abstraits pour être codés en toute généralité.

### 7.2.1 Déclaration

Pour déclarer une classe abstraite, on doit utiliser le mot-clé `abstract` dans la déclaration de la classe. Par exemple, le code suivant déclare une classe abstraite *Shape* :

```
1  public abstract class Shape {
2    // No code for now
3  }
```

Du fait que la classe *Shape* a été déclarée abstraite, on ne peut pas créer son objet même s'il possède un constructeur public (constructeur par défaut ajouté par le compilateur). On peut déclarer une variable d'une classe abstraite comme on fait pour une classe concrète. L'extrait de code suivant montre certaines utilisations valides et non valides de la classe *Shape* :

```
1  Shape s; // OK
2  s=new Shape(); // A compile-time error. Cannot create a Shape
    object
```

Les classes abstraites se rassemblent aux classes concrètes, à l'exception de l'utilisation du mot-clé `abstract` dans la déclaration. Une classe abstraite possède, comme n'importe quelle autres classe concrète, des variables d'instance et des méthodes d'instance pour définir l'état et le comportement de ses objets. Mai, la déclarant d'une classe abstraite, indique que la classe a des définitions de méthodes (comportements) incomplètes pour ses objets et qu'elle doit être considérée comme incomplète pour la création des objets.

### 7.2.2 Méthodes abstraites

Dans une classe abstraite, une méthode incomplète est une méthode ayant une déclaration, mais pas de corps. Les accolades qui suivent la déclaration de la méthode indiquent le corps de la méthode. Dans le cas d'une méthode incomplète, les accolades sont remplacées par un point-virgule. Si une méthode est incomplète, on doit l'indiquer en utilisant le mot-clé `abstract` dans la déclaration de la méthode.

**Exemple 35** L'extrait de code suivant montre un exemple de déclaration d'une méthode abstraite *draw* dans la classe *Shape* :

```
1   public abstract class Shape {
2     public Shape() {
3     }
4     public abstract void draw();
5   }
```

R La déclaration d'une classe abstraite, ne signifie pas nécessairement qu'elle possède au moins une méthode abstraite. Une classe abstraite peut avoir toutes ses méthodes soient concrètes. Comme elle peut avoir toutes ses méthodes soient abstraites. Comme elle peut avoir des méthodes concrètes et d'autres abstraites. Comme exemple voie la classe *Shape* du Pseudo-code 7.1.

R

- ☆ On ne peut pas créer d'objets d'une classe abstraite ;
- ☆ Si une classe possède une méthode déclarée abstraite, la classe doit être déclarée abstraite.
- ☆ Si une classe ne possède aucune méthode abstraite, on peut toujours déclarer la classe abstraite.
- ☆ Une méthode abstraite est déclarée de la même manière que toute autre méthode, sauf que son corps est indiqué uniquement par un point-virgule.

```
1  public abstract class Shape {
2    private String name;
3    public Shape() {
4      this.name = "Unknown shape";
5    }
6    public Shape(String name) {
7      this.name = name;
8    }
9    public String getName() {
10     return this.name;
11   }
12   public void setName(String name) {
13     this.name = name;
14   }
15   // Abstract methods
16   public abstract void draw();
17   public abstract double getArea();
18   public abstract double getPerimeter();
19 }
```

Pseudo-code 7.1 – Exemple d'une classe abstraite qui contient des méthodes concrètes et d'autres abstraites.

### 7.2.3 Héritage des classes abstraites

Une classe abstraite garantit l'utilisation de l'héritage, du moins en théorie. Sinon, une classe abstraite en elle-même est inutile. Par exemple, jusqu'à ce que quelqu'un fournisse les implémentations des méthodes abstraites de la classe *Shape*, ses autres parties (variables d'instance, méthodes concrètes et constructeurs) n'ont aucune utilité. Donc, pour fournir l'implémentation des méthodes abstraites on doit créer des sous-classes d'une classe abstraite.

**Exemple 36** Le Pseudo-code 7.2 contient un code d'une classe *Rectangle*, hérité de la classe abstraite *Shape*. Du fait que la classe *Rectangle* n'est pas déclarée abstraite, elle est une classe concrète et ses objets peuvent être créés.

```
1  public class Rectangle extends Shape {
2    private final double width;
3    private final double height;
4    public Rectangle(double width, double height) {
5      //Set the shape name as "Rectangle"
6      super("Rectangle");
7      this.width = width;
8      this.height = height;
9    }
10   //Provide an implementation for inherited abstract draw()
     method
11   @Override
12   public void draw() {
13     System.out.println("Drawing a rectangle...");
14   }
15   //Provide an implementation for inherited abstract getArea()
     method
16   @Override
17   public double getArea() {
18     return width * height;
19   }
20   //Provide an implementation for inherited abstract getPerimeter
     () method
21   @Override
22   public double getPerimeter() {
23     return 2.0 * (width + height);
24   }
25 }
```

Pseudo-code 7.2 – Exemple d'une classe *Rectangle* héritée de la classe abstraite *Shape*.

Il existe de nombreuses règles qui régissent l'utilisation de classes et de méthodes abstraites dans un programme Java. La plupart de ces règles sont les suivantes :

- ☆ Une classe abstraite doit être sous-classée pour être utile, et la sous-classe doit remplacer et fournir une implémentation pour les méthodes abstraites
- ☆ Si une classe concrète héritée d'une classe abstraite ne remplace pas toutes les méthodes abstraites de sa super-classe et ne les implémente pas, elle est considérée comme incomplète et doit être déclarée abstraite.
- ☆ Si une classe héritée d'une classe abstraite remplace et fournit des implémentations pour toutes les méthodes abstraites de la super-classe et ne déclare aucune méthode abstraite, il n'est pas nécessaire de la déclarer abstraite, bien qu'elle puisse l'être.
- ☆ Une classe abstraite ne doit pas déclarer tous les constructeurs privés. Sinon, la classe abstraite ne peut pas être sous-classée. Rappelons que que les constructeurs de toutes les classes ancêtres (y compris une classe abstraite) sont toujours invoqués lorsqu'un objet d'une classe est créé.
- ☆ Une classe abstraite ne peut pas être déclarée finale. Rappelons qu'une classe finale ne peut pas être dérivée par d'autres classes, ce qui limite une classe abstraite d'être utile dans un vrai sens.
- ☆ Une méthode abstraite est peut être héritée par une sous-classe comme toute autre méthode.
- ☆ Une méthode abstraite ne peut pas être déclarée *static*. Une méthode abstraite doit être remplacée et implémentée par une sous-classe. Une méthode statique ne peut pas être remplacée. Toutefois, elle peut être masquée.
- ☆ Une méthode abstraite ne peut pas être déclarée privée. Une méthode privée ne peut pas être héritée et ne peut donc pas être remplacée. Tandis que, l'exigence d'une méthode abstraite est qu'une sous-classe doit pouvoir la remplacer et en fournir une implémentation.
- ☆ On peut déclarer une variable d'un type de classe abstraite ; et on peut appeler des méthodes de la classe abstraite en utilisant cette variable. L'extrait du code suivant donne un aperçu logique (pour plus de détaille réfère à la section de polymprphisme par sous-typage) :

```
1    // Upcasting at work
2    Shape s = new Rectangle(2.0, 5.0);
3    double area = s.getArea(); // s.getArea() will call the
   getArea() method of the Rectangle class.
```

- ☆ Une méthode abstraite dans une classe peut remplacer une méthode abstraite de sa super-classe sans fournir d'implémentation. La méthode abstraite de

sous-classe peut affiner le type de retour ou la liste d'exceptions de la méthode abstraite remplacée. L'extrait de code suivant montre un exemple d'une tell situation :

```
1  public abstract class A {
2    public abstract void m1() throws CE1, CE2;
3  }
4
5  public abstract class B extends A {
6    public abstract void m1() throws CE1;
7  }
8
9  public class C extends B {
10   public void m1() { // Code goes here
11   }
12 }
```

## 7.3 Interfaces

Java n'autorise que l'héritage simple où chaque classe ne peut avoir qu'une seule super-classe. Dans certaines situations, l'héritage simple seul est insuffisant pour produire une bonne conception du programme. L'héritage multiple (une sous-classe peut avoir plusieurs super-classes) est utile quand une nouvelle classe veut ajouter un nouveau comportement et garder la plupart ou tout l'ancien comportement. L'héritage multiple est fournit par Java via le concept *interface*.

Une interface est une classe complètement abstraite utilisée pour regrouper les méthodes associées avec des corps vides, en fournissant uniquement leur signature. Ensuite, une classe peut implémenter une ou plusieurs interfaces. Le but principal de la déclaration d'une interface est de créer une spécification (ou un concept) abstrait en déclarant zéro ou plusieurs méthodes abstraites.

R Une interface ne fournit pas d'implémentation réelle des méthodes, mais elle fournit simplement une liste de méthodes que la classe qui implémente l'interface doit fournir.

### 7.3.1 Déclaration d'interface

En Java, une interface se déclare avec le mot-clé `interface`. Une interface est déclarée dans son propre fichier qui porte le même nom que l'interface. La syntaxe de déclaration d'une interface en Java est comme suit :

```
[modifiers] interface interfaceName {

  <constant-declaration>
  <method-declaration>
  <nested-type-declaration>
}
```

Une déclaration d'interface commence par une liste facultative de modificateurs. Similaire à une classe, une interface peut avoir une visibilité publique ou au package. Le mot clé public est utilisé pour indiquer que l'interface a une visibilité publique. L'absence de modificateur visibilité indique que l'interface a une visibilité package. Une déclaration d'interface est implicitement abstraite. En d'autres termes, une déclaration d'interface est toujours abstraite que on la déclare abstraite explicitement ou non. Une interface peut avoir trois types de membres :

- ☆ Des champs constants
- ☆ Des méthodes abstraites, statiques, privées et par défaut
- ☆ Des types statiques (interfaces et classes imbriquées)

R La déclaration d'interface ressemble beaucoup à une déclaration de classe, sauf qu'une interface ne peut pas avoir de variables d'instance et de classe mutables. Contrairement à une classe, une interface ne peut pas être instanciée. Tous les membres d'une interface sont implicitement publics.

#### 7.3.1.1 Déclaration des champs constants :

Une interface ne peut avoir que des attributs constants. Les attributs de l'interface sont par défaut *public*, *static* et $final$. Bien que la syntaxe de déclaration d'interface permette l'utilisation de ces mots-clés dans une déclaration des champs, leur utilisation est redondante. Il est recommandé de ne pas les utiliser lors de la déclaration des champs dans une interface. La déclaration des champs constants d'interface est comme montré par l'extrait de code suivant, qui déclare une interface nommé *Choices* :

```
public interface Choices {
  int YES = 1;
  int NO = 2;
}
```

L'accès à un champ d'une interface se fait par : `InterfaceNaame.fieldName`. On peut utiliser `Choices.Yes` et `Choices.No` pour accéder aux valeurs des champs $Yes$ et $No$. L'extrait de code suivant montre un exemple d'utilisation de cette notation pour accéder aux champs d'une interface :

```
public class ChoicesTest {
  public static void main(String [] args) {
  System.out.println("Choices.YES = " + Choices.YES);
    //  Choices.YES = 1
  System.out.println("Choices.NO = " + Choices.NO);
    //Choices.NO = 2
  }
}
```

R

- ☆ Les champs d'une interface sont toujours finales, que le mot clé *final* soit utilisé ou non dans leur déclaration. Cela implique qu'on doit initialiser un champ au moment de la déclaration ;
- ☆ Une interface ne peut pas contenir des constructeurs ;
- ☆ C'est une convention d'utiliser toutes les lettres majuscules pour identifier un champ dans une interface pour indiquer qu'il s'agit de constantes.

#### 7.3.1.2 Déclaration des méthodes :

On peut déclarer quatre types de méthodes dans une interface :

- ☆ méthodes abstraites
- ☆ méthodes statiques *static methods*
- ☆ méthodes par défaut (*Default methods*)
- ☆ méthodes privées

**Exemple 37** L'extrait de code suivant présente un exemple d'une interface contient quatre types de méthodes :

```
    interface AnInterface {
      // An abstract method
      int m1();
      // A static method
      static int m2() {
        // The method implementation goes here
      }
      // A default method
      default int m3() {
        // The method implementation goes here
      }
      // A private method
      private int m4() {
        // The method implementation goes here
      }
    }
```

R Le modificateur *final* n'est pas autorisé dans la déclaration des méthodes dans les interfaces.

**Méthodes abstraites :** toutes les déclarations de méthode dans une interface sont implicitement abstraites et publiques sauf si elles sont déclarées statiques ou par défaut. Une méthode abstraite d'une interface n'a pas d'implémentation. Le corps de la méthode abstraite est toujours représenté par un point-virgule et non par une paire d'accolades. L'extrait de code suivant déclare une interface nommée *Player* contenant quatre méthodes abstraites :

```
1  public interface Player {
2    public abstract void play();
3    public abstract void stop();
4    public abstract void forward();
5    public abstract void rewind();
6  }
```

L'utilisation des mots-clés *abstract* et *public* dans une déclaration de méthode d'interface est redondante, même si elle est autorisée par le compilateur, car une méthode dans une interface est implicitement abstraite et publique. La déclaration précédente de l'interface *Player* peut être réécrite comme suit sans changer sa signification :

```
1  public interface Player {
2    void play();
3    void stop();
4    void forward();
5    void rewind();
6  }
```

R Les déclarations de méthodes abstraites dans une interface peuvent contenir des paramètres, un type de retour et une clause throws (exception). Les méthodes abstraites d'une interface sont héritées par des classes qui implémentent l'interface, et les classes les remplacent pour leur fournir une implémentation. Cela implique qu'une méthode abstraite dans une interface ne peut pas être déclarée finale.

**Méthodes statiques (static methods) :** Contrairement aux méthodes statiques d'une classe, les méthodes statiques d'une interface ne sont pas héritées par des classes d'implémentation ou des sous-interfaces (une interface qui est hérité d'une autre interface est appelée une sous-interface). Il y a une seule façon d'appeler les méthodes statiques d'une interface : en utilisant le nom de l'interface, $InterfaceName.methodeName()$. L'extrait de code Pseudo-code 7.3 présente un exemple de déclaration d'une interface contenant une méthode static $letThemWalk$

et une autre abstraite, *walk*.

```
1  public interface Walkable {
2    // An abstract method
3    void walk();
4    // A static convenience method
5    public static void letThemWalk(Walkable[] list) {
6      for (Walkable w : list) {
7        w.walk();
8      }
9    }
10 }
```

Pseudo-code 7.3 – Exemple de déclaration d'une interface nommée *Walkable*

**Méthodes par défaut :** Les méthodes par défaut ont été introduites à partir de Java 8. Avant Java 8, les interfaces ne pouvaient avoir que des méthodes abstraites. Les méthodes par défaut ont été ajoutées pour que les interfaces existantes puissent évoluer.

Depuis Java 8, une méthode d'interface peut être dotée d'un corps. Il s'agit alors d'une définition par défaut (default method) qu'il convient de déclarer au moyen du mot-clé `default`. Une méthode par défaut fournit une implémentation par défaut de la méthode pour la classe qui implémente l'interface, mais ne remplace pas la méthode par défaut. Quelles sont les similarités et les différences entre une méthode concrète dans une classe et une méthode par défaut dans une interface ?

- ☆ Les deux fournissent une implémentation de la méthode ;
- ☆ Les deux ont accès au mot-clé *this* de la même manière. Autrement dit, le mot-clé *this* est la référence de l'objet sur lequel la méthode est appelée ;
- ☆ La différence majeure réside dans l'accès à l'état de l'objet. Une méthode concrète dans une classe peut accéder aux variables d'instance de la classe. Cependant, une méthode par défaut n'a pas d'accès aux variables d'instance de la classe implémentant l'interface. La méthode par défaut peut accéder aux autres membres de l'interface, par exemple aux autres méthodes, constantes ;
- ☆ Les deux méthodes peuvent avoir une *clausethrows*.

R Une méthode par défaut est une méthode d'instance et fournit une implémentation par défaut ; elle peut être héritée (can be overridden). Ainsi, Une classe implémentant une interface n'est plus obligée de redéfinir les méthodes par défaut de celle-ci.

**Méthodes privées :** Avant JDK 9, toutes les méthodes d'une interface étaient implicitement publiques. Donc, à partir de JDK 9, on peut avoir des méthodes privées dans une interface qui sont ni des méthodes d'instance abstraites, ni des méthodes par défaut, et qui peuvent êtres des méthodes statiques.

### 7.3.2 Implémentation d'interface

Une interface permet de spécifier un protocole qu'un objet garantit d'offrir lorsqu'il interagit avec d'autres objets. Il spécifie le protocole en termes de méthodes abstraites et par défaut. Cette spécification sera implémentée à un moment donné par une classe. Lorsqu'une classe implémente une interface, elle fournit des implémentations pour toutes les méthodes abstraites de l'interface. Une classe peut fournir une implémentation partielle des méthodes abstraites de l'interface, et dans ce cas, la classe doit se déclarer abstraite.

Une classe qui implémente une ou plusieurs interfaces utilise le mot clé `implements` pour spécifier le nom de l'interface. Le mot-clé `implements` est suivi d'une liste de noms d'interfaces séparés par des virgules. Une classe peut implémenter plusieurs interfaces. L'extrait de code suivant montre un exemple de d'implémentation de l'interface du Pseudo-code 7.4, la classe *Fish* :

```
1  public class Fish implements Swimmable {
2    private String name;
3    public Fish(String name) {
4      this.name = name;
5    }
6    @Override
7    public void swim() {
8      System.out.println(name + " (a fish) is swimming.");
9    }
10   // More code for the Fish class goes here
11 }
```

La classe qui implémente une interface doit remplacer et implémenter toutes les méthodes abstraites déclarées dans l'interface. Sinon, la classe doit être déclarée abstraite. Les méthodes par défaut d'une interface sont également héritées par les classes d'implémentation. Les classes d'implémentation peuvent choisir (mais ne sont pas obligées) de remplacer les méthodes par défaut. Les méthodes statiques d'une interface ne sont pas héritées par les classes d'implémentation.

R Une classe implémentant des interfaces peut avoir d'autres méthodes qui ne sont pas héritées des interfaces implémentées. D'autres méthodes peuvent avoir le même nom et un nombre et/ou des types de paramètres différents de ceux déclarés dans les interfaces implémentées.

Un objet peut être créé d'une classe qui implémente une interface de la même manière (en utilisant l'opérateur new avec son constructeur), de la création d'un objet d'une classe concrète. On peut créer un objet de la classe *Fish* comme suit :

```
// Create an object of the Fish class
Fish fifi = new Fish("Fifi");
```

Lorsqu'une classe implémente une interface, son objet a un type supplémentaire, qui est le type défini par l'interface implémentée. Dans le cas de la classe *Fish*, l'objet *fifi* a deux types : *Fish* et *Swimmable*. Puisqu'un objet de la classe *Fish* a deux types, *Fish* et *Swimmable*, on peut affecter la référence d'un objet *Fish* à une variable de type *Fish* ainsi qu'à une variable de type *Swimmable*. Le code suivant résume cela :

```
Fish guppi = new Fish("Guppi");
Swimmable hilda = new Fish("Hilda");
```

### 7.3.3 Implémentation de multiples interfaces

Une classe peut implémenter plusieurs interfaces. Toutes les interfaces qu'une classe implémente sont listées après le mot clé *implements* dans la déclaration de classe. Les noms d'interface sont séparés par une virgule. En implémentant plusieurs interfaces, la classe s'engage à fournir l'implémentation de toutes les méthodes abstraites dans toutes les interfaces.

Il n'y a pas de limite au nombre maximum d'interfaces implémentées par une classe. Lorsqu'une classe implémente une interface, ses objets obtiennent un type supplémentaire. Si une classe implémente plusieurs interfaces, ses objets reçoivent autant de nouveaux types que le nombre d'interfaces implémentées.

**Exemple 38** Le Pseudo-code 7.5 montre une exemple d'une classe *Turtle* qui implémente deux interfaces : *Walkable* (Pseudo-code 7.3) et *Swimmable* (Pseudo-code 7.4). Un objet de type *Turtle* a trois types : *Turtle*, *Walkable*, et *Swimmable*.

```

public interface Swimmable {
  void swim();
}
```

Pseudo-code 7.4 – Exemple de déclaration d'une interface nommée *Swimmable*

```
1  public class Turtle implements Walkable, Swimmable {
2    private String name;
3    public Turtle(String name) { this.name = name;}
4 // Adding a bite() method to the Turtle class
5    public void bite() {
6      System.out.println(name + " (a turtle) is biting.");
7    }
8 // Implementation for the walk() method of the Walkable interface
9    @Override
10   public void walk() {
11     System.out.println(name + " (a turtle) is walking.");
12   }
13 // Implementation for the swim() method of the Swimmable interface
14   @Override
15   public void swim() {
16     System.out.println(name + " (a turtle) is swimming.");
17   }
18 }
```

Pseudo-code 7.5 – Exemple d'une classe *Turtle* qui implémente deux intefaces: *Walkable* et *Swimmable*

## 7.4 Conclusion

La création des classes abstraites est la technique utilisée par le programmeur d'une super classe pour exiger que les programmeurs de ses sous classes définissent les fonctionnalités nécessaires des comportements. Les classes abstraites ne peuvent pas être instanciées et si une classe contient une méthode abstraite, la classe doit être déclarée abstraite. En plus, une classe peut être déclarée abstraite même si elle ne contient aucune méthode abstraite. Les méthodes abstraites sont censées être remplacées et fournies une implémentation par des sous-classes.

Une interface est une spécification destinée à être implémentée par des classes. Une interface peut contenir des membres qui sont des constantes statiques, des méthodes abstraites, des méthodes par défaut, des méthodes statiques et des types imbriqués. En plus, une interface ne peut pas avoir de variables d'instance et ne peut pas être instanciée. La classe implémentant une interface hérite tous les membres de l'interface, à l'exception les méthodes statiques. Si la classe hérite des méthodes abstraites des interfaces implémentées, elle doit les remplacer et leur fournir une implémentation, ou la classe doit se déclarer abstraite. En Java, les interfaces permet aussi d'avoir l'héritage multiple en implémentant plusieurs interfaces en même temps.

# 8. Gestion des exceptions

Sommaire



## 8.1 Introduction

La gestions des exceptions est une technique qui permet d'améliorer la fiabilité d'un programme. Elle consiste à la gestion des erreurs qui se produisent pendant l'exécution d'un programme et qui conduisent soit à l'arrêt du programme soit à des résultats incorrects. En java, le mécanisme de gestion des exceptions permet de traiter les erreurs qui se produisent pendant l'exécution d'un programme et de séparer le code qui décrit le déroulement normal du programme et le code de gestion d'erreurs. Dans ce chapitre, nous allons présenter le concept de gestion des exceptions et son implémentation en Java.

## 8.2 C'est quoi une exception

Une exception est une condition qui peut survenir lors de l'exécution d'un programme Java lorsqu'un chemin d'exécution normal n'est pas défini. Par exemple, un programme Java peut rencontrer une expression numérique qui tente de diviser un entier par zéro. Une telle condition peut se produire lors de l'exécution de l'extrait de code suivant :

```
int x = 10, y = 0, z;
z = x / y; // Divide-by-zero
```

La division par zéro est un problème sérieux nécessitant l'arrêt de l'exécution du programme. Une telle situation est exceptionnelle mais réparable. Pour éviter l'arrêt d'exécution d'un programme dans telles situations, le programme doit être capable de déterminer la gravité de l'erreur et de prendre les mesures qui s'imposent pour la réparer. Par exemple, on peut réécrire le code précédent comme suit :

```
  int x = 10, y = 0, z;
  if (y == 0) {
    // Report the abnormal/error condition here
  } else {
    // Perform division here
    z = x / y;
  }
```

R En Java, nous utilisons le mot "*exception*" au lieu de "*erreur*" pour indiquer une condition anormale dans un programme ; l'expression "*gestion des exceptions*" est utilisée au lieu de "*gestion des erreurs*". En général, on dit qu'une erreur se produit et on la gère. En Java, on dit qu'une *exception* est *levée* et qu'on "l'*attrape*".

## 8.3 Avantages de gestion des exceptions

Dans certains cas, l'ajout de la gestion des erreurs au moyen des if-else introduit de nombreuses instructions if-else imbriquées aboutissant à un code spaghetti. Ainsi, la manière de gestion des erreurs qui utilise des instructions if-else n'est élégante ni maintenable. Java offre une meilleure façon de gérer les erreurs : en séparant le code qui effectue des actions du code qui gère les erreurs. L'utilisation des routines de gestion des exceptions dans un code, augmente sa fiabilité. Considérant l'extrait de code suivant :

```
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter integer: ");
  int number = scanner.nextInt();
```

Si un utilisateur saisie une valeur d'entrée qui n'est pas un entier, on obtiendrait un message d'erreur comme celui-ci :

```
Exception in thread "main" java.util.InputMismatchException
  at java.util.Scanner.throwFor(Scanner.java:819)
  at java.util.Scanner.next(Scanner.java:1431)
  at java.util.Scanner.nextInt(Scanner.java:2040)
  at java.util.Scanner.nextInt(Scanner.java:2000)
  at Ch8Sample1.main(Ch8Sample1.java:35)
```

Ce message d'erreur indique que le système a détecté une exception appelée :

$$InputMismatchException,$$

une erreur qui se produit lorsque nous essayons de convertir une chaîne qui ne peut pas être convertie en valeur numérique.

Lorsque nous laissons le système gère les exceptions, une seule exception levée entraînera très probablement l'arrêt du programme. Au lieu de dépendre au système pour la gestion des exceptions, nous pouvons augmenter la fiabilité et la robustesse du programme si nous détectons nous-mêmes les exceptions en incluant des routines de récupération d'erreurs dans notre programme. Modifiant le code précédent pour qu'il boucle jusqu'à ce qu à la saisie d'une valeur valide pouvant être convertie en un entier. Pour ce faire, nous devons envelopper les instructions susceptibles de lever une exception avec l'instruction de contrôle `try-catch`. Dans cet exemple, il n'y a qu'une seule instruction qui peut potentiellement lever une exception, à savoir : `int number = scanner.nextInt();` Si on veut simplement afficher un message d'erreur lorsque l'exception est levée, on peut ajouter l'instruction `try-catch` au code précédent comme suit :

```
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter integer: ");
  try {
// A statement that could throw an exception
    int number = scanner.nextInt();
  } catch (InputMismatchException e) {
//The type of exception to be caught
    System.out.println("Invalid Entry. Please enter digits only.");
  }
```

Si l'instruction d'entrée ne lève pas d'exception, le programme renvoie la valeur entier saisie et continue l'exécution de la suite des instructions. S'il y a une exception, un message d'erreur est affiché à l'intérieur du bloc catch et la routine de saisie se répète.

## 8.4 Implémentation de la gestion des exceptions

L'implémentation de la gestion des exceptions se fait au moyen de quatre mécanismes qui sont chacun liés à un mot-clé de Java :

- ☆ Le lancement d'une exception dans une méthode découvrant une situation anormale sans pouvoir la traiter directement se fait au moyen du mot clé `throw`.
- ☆ Une méthode qui permet de traiter une exception lancée par un certain nombre de ses instructions, englobera ces dernières dans un bloc précédé du mot clé `try` (essayant d'exécuter ces instructions sachant qu'elle peuvent potentiellement lancer une exception). Le bloc *try* est donc un endroit réceptif aux exceptions lancées.
- ☆ Un bloc *try* est suivi de un ou plusieurs blocs désigné par le mot clé `catch`. si une instruction du bloc *try* a lancé une exception, son exécution va se diriger vers ces blocs. Les blocs *catch* attrapent donc l'exception pour la traiter.
- ☆ Les blocs catch sont parfois suivis d'un bloc associé au mot clé `finally` qui se charge de faire le ménage (comme libérer certaines ressources) après le traitement d'une exception.

### 8.4.1 Mots clés try & catch

Un bloc *try* commence par le mot clé `try`. Le code du bloc try est placé entre les accolades ouvrantes et fermantes. Il doit être suivi d'un ou plusieurs blocs *catch*, ou d'un bloc *finally*, ou une combinaison des deux. On peut associer un ou plusieurs blocs *catch* à un bloc *try*. La syntaxe générale d'un bloc *try-catch* est la suivante :

```
try {
  // Your code that may throw an exception goes here
} catch (ExceptionClass1 e1){
  // Handle exception of ExceptionClass1 type
} catch (ExceptionClass2 e2){
  // Handle exception of ExceptionClass2 type
} catch (ExceptionClass3 e3){
  // Handle exception of ExceptionClass3 type
}
```

- ☆ Le bloc *try* contient les instructions susceptibles de lever (déclencher) une exception ;
- ☆ Le bloc *catch* sert de gestionnaire d'exception.
- ☆ Le paramètre de l'instruction *catch* indique le type et le nom d'instance de l'exception générée.

Si un événement indésirable survient dans le bloc *try*, la partie éventuellement non exécutée de ce bloc est abandonnée et le premier bloc catch s'exécute. Si un bloc

catch est défini pour capturer l'exception générée du bloc *try* alors elle est traitée en exécutant le code associé au bloc *catch*. Si le bloc *catch* est vide, alors l'exception capturée est ignorée.

**Exemple 39** Utilisons un bloc *try-catch* pour gérer l'éventuelle exception de division par zéro :

```
  int x = 10, y = 0, z;
  try {
  z = x / y;
  System.out.println("z = " + z);
  } catch (ArithmeticException e) {
  // Get the description of the exception
  String msg = e.getMessage();
  // Print a custom error message
  System.out.println("An error has occurred. The error is: " + msg)
    ;
  }
  System.out.println("At the end of the program.");

// this code displays: ------------------------------------------
An exception has occurred. The error is: / by zero
At the end of the program.
```

**Exemple 40** Prenant un autre exemple dans lequel deux exceptions peuvent être enlevées :

```
   // declaration of a table of objects
try{
  for(int i=0;i<t.length;i++)
    // instructions to access to t
}catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Check your table index: "+e.getMessage());
}catch(NullPointerException e2){
System.out.println("Check your table objects: "+e2.getMessage());
}
```

S'il y a une tentative d'accès hors les bornes du tableau $t$, le programme affiche : "Check your table index :"+ $e.getMessage()$. Si les objets de $t[i]$ ne sont pas créés, l'exécution du programme affiche le message : "Check your table objects : "+ $e2.getMessage()$.

### 8.4.2 Mot clé finally

nous avons vu qu'un bloc *try* peut également avoir zéro ou un bloc *finally*. Un bloc *finally* n'est jamais utilisé seul. Il est toujours utilisé avec un bloc *try*. La syntaxe pour utiliser un bloc *finally* est la suivante :

```
finally {
  // Code for finally block goes here
}
```

Un bloc *finally* commence par le mot-clé clé `finally`, suivi d'une accolade ouvrante et d'une accolade fermante. Le code d'un bloc *finally* est placé entre les accolades. Il existe deux combinaisons possibles de blocs *try*, *catch* et *finally* : *try-catch-finally* ou *try-finally*. La syntaxe d'un bloc *try-catch-finally* est la suivante :

```
try {
  // Code for try block goes here
} catch(Exception1 e1) {
// Code for catch block goes here
} finally {
// Code for finally block goes here
}
```

La syntaxe d'un bloc *try – finally* est comme suit :

```
  try {
    // Code for try block goes here
  } finally {
    // Code for finally block goes here
  }
```

Un bloc *finally* est garanti d'être exécuté quoi qu'il arrive dans le bloc *try* et/ou *catch* associé. Il existe deux exceptions à cette règle : le bloc *finally* peut ne pas être exécuté si le thread qui exécute le bloc *try* ou *catch* meurt, ou si l'application Java se ferme, par exemple en appelant la méthode *exit()* à l'intérieur du bloc *try* ou *catch*.

Généralement, un bloc *finally* est utilisé pour écrire un code de nettoyage. Par exemple, on peut utiliser certaines ressources dans un programme qui doivent être libérées à la fin de leur utilisation. Un bloc *try-finally* permet d'implémenter cette logique.

On écrit souvent des blocs *try-finally* lors la création des programmes qui effectuent des transactions de base de données et des fichiers d'entrées/sorties. On utilise une connexion à la base de données dans le bloc *try* et libérez la connexion dans le bloc *finally*. Lorsque on travaille avec un programme lié à une base de données, on doit libérer la connexion à la base de données utilisée, peu importe ce qui arrive à la transaction.

### 8.4.3 Classe Throwable

En Java, une exception est représentée comme une instance de la classe *Throwable* ou l'un de ses sous-classes. La classe *Throwable* a deux sous classes : *Error* et

*Exception*. La Figure 8.1 montre quelques classes d'exception.

- ☆ La Classe *Error* est la classe des erreurs graves que le programmeur ne peut pas les traités. Ces erreurs provoque obligatoirement l'arrêt du programme, par exemple : $Outofmemory$ ou $VirtualMachineerror$.
- ☆ La classe *Exception* est la classe des exceptions que le programmeur peut les traiter. Elles sont liés au langage Java par exemple : $NullPointerException$.
- ☆ la classe *RuntimeException* est une sous-classe de la classe *Exception*, concerne les erreurs liées à l'environnement d'exécution, par exemple : *Arithmetic Error* . Ces exceptions n'ont pas forcément besoin d'être rattrapées.

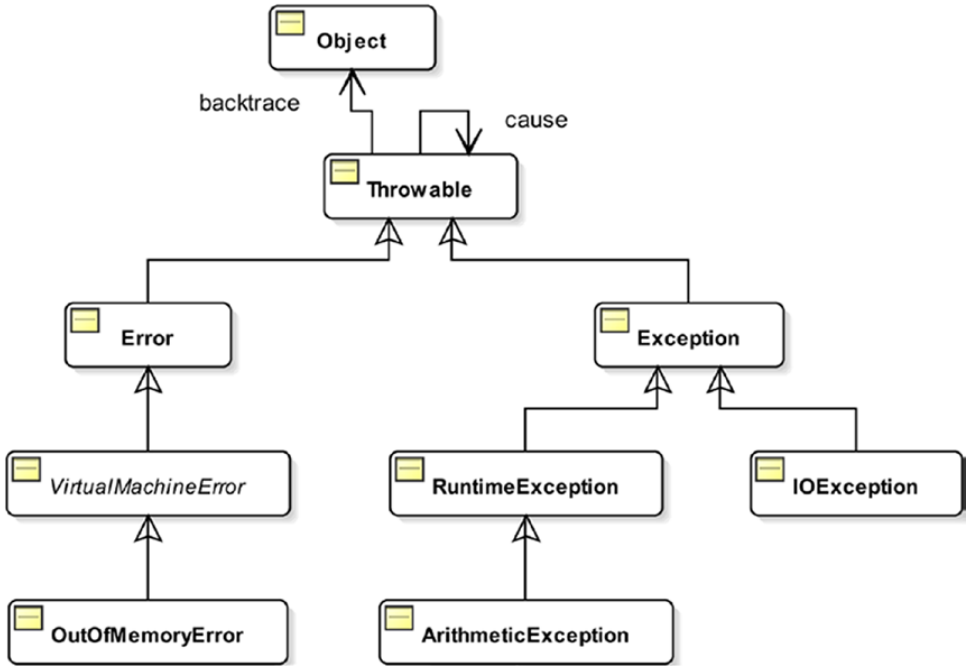



FIGURE 8.1 – Hiérarchie des classes des exceptions [9].

R La classe Object n'appartient pas à la famille des classes d'exception. Il est représenté sur la Figure 8.1 comme un ancêtre de la classe *Throwable* dans la hiérarchie d'héritage.

Lorsqu'une exception est levée, elle doit être une instance de la classe *Throwable* ou de l'une de ses sous-classes. Le paramètre du bloc *catch* doit être de type *Throwable* ou l'une de ses sous-classes, telle que : *Exception*, *ArithmeticException*, *IOException*, etc. Les blocs *catch* suivants sont valides car ils spécifient des types *Throwable* en tant que paramètre, qui sont la classe *Throwable* ou ses sous-classes :

```
1 // Throwable is a valid exception class
2 catch(Throwable t) {
3 }
```

```
// Exception is a valid exception class because it is a subclass of
    Throwable
catch(Exception e) {
}
// IOException class is a valid exception class because it is a
   subclass of Throwable
catch(IOException t) {
}
// ArithmeticException is a valid exception class because it is a
   subclass of Throwable
catch(ArithmeticException t) {
}
```

R On peut également créer des classes d'exception personnalisées héritées de l'une des classes d'exception. La Figure 8.1 ne montre que quelques-unes des centaines de classes d'exception disponibles dans la bibliothèque de classes Java.

**Quelques méthodes de la classe Throwable :** ci-dessous certaines des méthodes couramment utilisées de la classe Throwable.

- ☆ `getCause()` : renvoie la cause de l'exception ;
- ☆ `getMessage()` : renvoie le message d'erreur détaillé décrivant l'exception ;
- ☆ `printStackTrace()` : affiche la liste des appels de méthodes ayant conduit à l'exception ;
- ☆ `Exception()` : constructeur sans argument ;
- ☆ `Exception(String)` : Constructeur avec Argument.

**Quelques sous classes de la sous-classes de RuntimeException :** La sous classe *RuntimeException* est la super classe des exceptions qui peuvent être levées lors un fonctionnement normal de la machine virtuelle Java. Une liste partiale des sous classes de cette classe est la suivante :

- ☆ *NullPointerException* : utilisation d'une référence *null* ;
- ☆ *IndexOutOfBoundsException* : un indice (sur un tableau, une chaîne, une liste) a dépassé les limites inférieures ou supérieures.
- ☆ *ArithmeticException* : une exception est survenue sur une opération arithmétique, comme une division d'un entier par zéro.
- ☆ *NegativeArraySizeException* : une exception générée lors la création d'un tableau ou une chaîne avec une taille négative.
- ☆ *ClassCastException* : tentative de cast d'un objet dans un type incorrecte.

- ☆ *IllegalArgumentException* : une méthode a été appelée ou invoquée avec un mauvais argument.

## 8.5 Conclusion

Le système de gestion des exceptions permet de faire en sorte qu'une erreur détectée dans une partie d'un programme puisse être gérée et traitée dans une autre partie. Une exception représente une condition d'erreur, et lorsqu'elle se produit, on dit qu'une exception est levée. Une exception levée doit être gérée soit en l'interceptant, soit en la propageant à d'autres méthodes. Dans ce chapitre, on a introduit la notion de gestion des exceptions en Java et on a vu que le mécanisme de gestion des exceptions est puissant et très utile dans la programmation.

# Bibliographie


[1] Iuliana Cosmina. *Java for Absolute Beginners : Learn to Program the Fundamentals the Java 9+ Way*. Apress, 2018.

[2] Samanta Debasis and Monalisa Sarma. *Joy with Java : Fundamentals of Object Oriented Programming*. Cambridge University Press, 2023.

[3] Claude Delannoy. *S'initier à la Programmation et à l'Orienté Objet avec des Exemples en C, C++, C#, Python, Java et PHP*. Groupe Eyrolles, second edition, 2016.

[4] Claude Delannoy. *Programmer en Java : Couvre Java 10 à Java 14*. Groupe Eyrolles, 11e edition, 2020.

[5] Michel Divay. *La Programmation Objet en Java : Cours et exercices corrigés*. Dunod, 2006.

[6] Amany Fawzy Elgamal. *Object-Oriented Programming*. Cambridge Scholars Publishing, Newcastle, 2024.

[7] Jamila Sam, Jean-Cédric Chappelier, and Vincent Lepetit. *Introduction à la Programmation Orientée Objet (en JAVA)*. EPFL Press, 2016.

[8] Vaskaran Sarcar. *Interactive Object-Oriented Programming in Java : Learn and Test Your Programming Skills*. Apress, second edition, 2020.

[9] Kishori Sharan and Adam L. Davis. *Beginning Java 17 : Fundamentals Object-Oriented Programming in Java 17*. Apress, third edition, 2022.