

Programmation Orientée Objet **EN JAVA**

Pr. Mohammed **Gabli**

Préambule

Java est un langage de programmation orientée objet avec lequel on peut faire de nombreuses sortes de programmes :

- des applications, sous forme de fenêtre ou de console ;
- des applets, qui sont des programmes Java incorporés à des pages web ;
- des applications pour appareils mobiles, avec J2ME ;
- des applications destinées aux applications d'entreprise J2EE ;
- et bien d'autres !

Chaque point demande plusieurs chapitres et beaucoup de temps. Et parce qu'on doit commencer par le commencement et le commencement de tout cela est Java SE (Java Standard Edition), alors on s'intéresse dans ce cours de présenter les bases du langage java.

Ce polycopié est destiné aux étudiants des Licences en informatique de l'Université Mohamed Premier, Oujda. Il concerne le module de la programmation orientée objet en Java de la Filière Sciences Mathématiques et Informatique (SMI, Semestre 5).

Table des matières

1	Introduction à Java	7
1.1	Historique et introduction	7
1.2	La plateforme Java	8
1.3	Premier programme en JAVA	8
1.3.1	Installation de JDK	9
1.3.2	Compilation	9
1.3.3	Exécution	10
1.3.4	Description du premier programme	10
1.4	Commentaires en JAVA	11
1.5	Données primitifs	11
1.6	Expressions et opérateurs	13
1.7	Exercices	13
2	Classes et objets	16
2.1	Introduction	16
2.2	Déclaration d'une classe	18
2.2.1	Définition des attributs	18
2.2.2	Définition des méthodes	19
2.3	Utilisation des classes	21
2.4	Portée des attributs	23
2.5	Surcharge des méthodes	23
2.6	Méthodes statiques et finales	26
2.6.1	Attributs statiques	26
2.6.2	Constantes final et static	27
2.6.3	Méthodes statiques	27
2.7	Lecture à partir du clavier	28
2.8	Chaines de caractères	30

2.9	Exercices	32
3	Constructeurs	35
3.1	Introduction	35
3.2	Définition	36
3.3	Utilisation de this	37
3.4	Surcharge des constructeurs	37
3.5	Constructeur par défaut	39
3.6	Constructeur de copie	40
3.7	Exercices	41
4	Héritage	43
4.1	Introduction	43
4.2	Exemple introductif	43
4.3	Utilisation de l'héritage	45
4.4	Accès aux attributs	46
4.5	Héritage hiérarchique	47
4.6	Redéfinition	48
4.7	Héritage et constructeurs	50
4.8	Héritage et méthodes finales	54
4.9	Polymorphisme et héritage	55
4.9.1	Introduction	55
4.9.2	Liaison dynamique	55
4.9.3	Polymorphisme et méthodes de classes	57
4.9.4	Utilité du polymorphisme	58
4.10	Exercices	60
5	Tableaux et collections	61
5.1	Tableaux	61
5.1.1	Déclaration et initialisation	61
5.1.2	Parcourir un tableau	63
5.1.3	Copie de tableaux	64
5.1.4	Passage et retour d'un tableau dans une méthode	65
5.1.5	Tableaux d'objets	67
5.2	Tableaux à plusieurs dimensions	67
5.2.1	Parcourir un tableau multi-dimensionnel	69

5.2.2	Initialisation au début	69
5.3	La collection ArrayList	70
5.4	Exercices	72
6	Classes abstraites, interfaces et packages	74
6.1	Classes abstraites	74
6.1.1	Méthodes abstraites	74
6.1.2	Classes abstraites	74
6.1.3	Utilité et exemple	75
6.2	Interfaces	76
6.2.1	Définition et déclaration	76
6.2.2	Implémentation d'une interface	76
6.2.3	Polymorphisme et interfaces	78
6.2.4	Héritage et interfaces	80
6.3	Packages	81
6.3.1	Création d'un package	81
6.3.2	Exemple	81
6.3.3	Classes du même package	83
6.3.4	Fichiers jar	84
6.4	Exercices	84
7	Gestion des exceptions	87
7.1	Introduction	87
7.2	Exceptions	87
7.3	Types d'exceptions	88
7.4	Méthodes des exceptions	89
7.5	Capture des exceptions	89
7.6	Utilisation du bloc try-catch	90
7.6.1	Un seul catch	90
7.6.2	plusieurs catch	91
7.6.3	Bloc finally	92
7.7	Exceptions personnalisées	92
7.7.1	Déclenchement d'une exception	92
7.7.2	Exemple complet	93
7.8	Exercices	94

8	Flux d'entrées/sorties en JAVA	95
8.1	Introduction	95
8.2	Flux binaires	95
8.2.1	Flux d'entrée	95
8.2.2	Flux de sortie	96
8.3	Flux de caractères	96
8.3.1	Flux d'entrée	96
8.3.2	Flux de sortie	96
8.4	Démarche de lecture et d'écriture de données	97
8.5	Lecture dans un fichier	97
8.5.1	Lecture des données à partir d'un fichier	97
8.5.2	Exemple : découper une chaîne de caractère	98
8.6	La classe Random	99
8.6.1	La méthode nextInt()	99
8.6.2	La méthode nextFloat()	99
8.6.3	La méthode nextBoolean()	99
8.6.4	La méthode nextInt(int n)	99
8.6.5	Exemple	99
8.7	Ecriture dans un fichier	100
8.8	Exercice	101
9	De UML à JAVA	102
9.1	Introduction	102
9.2	Classe et membres	102
9.3	Classe abstraite	103
9.4	Interface	103
9.5	Héritage	103
9.6	Réalisation (Implémentation)	104
9.7	Association	104
10	Introduction à Java graphique avec NetBeans	107
10.1	Introduction	107
10.2	Création d'un projet	108
10.3	Ajout d'une interface graphique dans le projet	108
10.3.1	Ajouter un composant visuel à votre projet	108
10.3.2	Suppression d'un fichier du projet	108

10.3.3	Compilation du projet	109
10.3.4	Éléments de base d'une fenêtre	109
10.4	Attachement d'un évènement à un objet	109
10.5	Exercices	112

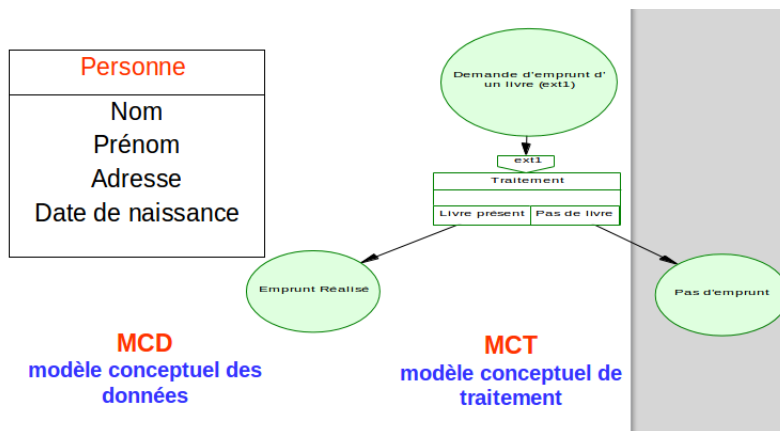
Chapitre 1

Introduction à Java

1.1 Historique et introduction

La *programmation classique* sépare les données des traitements. Ainsi, une application se décompose en deux étapes bien distinctes :

- La définition des *structures de données* capables d'accueillir l'information que l'on veut gérer.
- La conception de *fonctions et de procédures* travaillant sur les structures de données conçues à l'étape précédente.



Cette forme de programmation, rend difficile la mise en place de contrôles, destinés à garantir la cohérence et l'intégrité des données. L'utilisateur peut en effet accéder directement aux données, sans utiliser les fonctions mises à sa disposition. En **C** par exemple, on peut écrire :

```
typedef struct { char    nom[20] ;
                 char    sexe ;
                 float   taille ; } Personne ;

void Lire () {...}
void Afficher ( Personne *P ) {...}
void emprunter () {...}
...
```


La *programmation orientée objet* a été introduite afin de remédier à ces inconvénients. Son principe de base consiste à :

- regrouper (encapsuler) au sein d'une même unité (appelée classe) les données et les traitements.
- contrôler l'accès aux données en les déclarant privées ce qui obligera l'utilisateur de la classe à employer les traitements encapsulés pour agir sur celles-ci.

En général, Les langages orientés objets prennent en charge les quatre caractéristiques importantes suivantes : *Encapsulation, Abstraction, Héritage et Polymorphisme*.

Au niveau d'analyse et conception orientée objets, le langage de modélisation le plus populaire est *UML*, et au niveau de programmation orientée objet, on trouve de nombreux langages dont le plus populaire est *JAVA*.

Le projet Java est né en 1991 au sein de *Sun Microsystems* (racheté par *Oracle* en 2010) afin de produire des logiciels indépendants de toute architecture matérielle. L'idée consistait à traduire d'abord un programme source, non pas directement en langage machine, mais dans un pseudo langage universel disposant des fonctionnalités communes à toutes les machines. Ce code intermédiaire (*bytecode*) se trouve ainsi compact et portable sur n'importe quelle machine ; il suffit simplement que cette dernière dispose d'un programme approprié (*machine virtuelle*) permettant de l'interpréter dans le langage de la machine concernée. Les versions JAVA se succédèrent de *JDK1.0* en 1996 à *JDK1.8* (appelé *JAVA 8*) en 2014. La version 9 est annoncée pour 2016.

1.2 La plateforme Java

Les composantes de la plateforme Java sont :

- Le langage de programmation.
- La machine virtuelle (The Java Virtual Machine - JVM).
- La librairie standard.

Java est compilé et interprété :

- Le code source Java (se terminant par `.java`) est compilé en un fichier Java bytecode (se terminant par `.class`)
- Le fichier Java bytecode est interprété (exécuté) par la JVM
- La compilation et l'exécution peuvent se faire sur différentes machines
- Le fichier bytecode est portable. Le même fichier peut être exécuté sur différentes machines (hétérogènes).

1.3 Premier programme en JAVA

Un code source qui permet d'afficher le message *Bonjour* :

```
public class PremierExemple {
    public static void main(String[] args) {
        System.out.println("Bonjour");
    }
}
```

Le programme doit être enregistré (**obligatoirement**) dans un fichier portant le même nom que celui de la classe : `PremierExemple.java`.

Pour compiler le programme précédent, il faut tout d'abord installer l'environnement de développement **JDK** (**J**ava **D**evelopment **K**it).

1.3.1 Installation de JDK

Installation sous Linux

Installer openjdk-7, en tapant la commande :

```
sudo apt-get install openjdk-7-jdk
```

Ou bien, télécharger la version de jdk (jdk-7uxy-linux-i586.tar.gz ou jdk-7uxy-linux-x64.tar.gz) correspondant à votre architecture (32 ou 64 bits) de :

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

Décompresser le fichier téléchargé en tapant la commande :

- sur les systèmes 32 bits : `tar xzf jdk-7uxy-linux-i586.tar.gz`
- sur les systèmes 64 bits : `tar xzf jdk-7uxy-linux-x64.tar.gz`

Remplacer xy par le numéro de mise-à-jour, par exemple 55.

Ajouter dans le fichier .bashrc (gedit ~/.bashrc) la ligne suivante :

```
PATH=~/.jdk1.7.0_xy/bin:$PATH
```

Installation sous Windows

Télécharger la version de jdk (jdk-7uxy-windows-i586.exe ou jdk-7uxy-windows-x64.exe) correspondant à votre architecture (32 ou 64 bits) de :

<http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>

Exécuter le fichier téléchargé et ajouter C:\Program Files\Java\jdk1.7.0_xy\bin au chemin, en modifiant la variable path. La valeur de path doit ressembler à ce qui suit :

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.7.0_xy\bin
```

Pour modifier la valeur de path :

1. cliquer sur **démarrer** puis **Panneau de configuration** puis **Système et sécurité** puis **Paramètres système avancés**
2. cliquer sur **Variables d'environnement** puis chercher dans **variables système** Path et modifier son contenu.

1.3.2 Compilation

Dans une console, déplacez vous dans le répertoire où se trouve votre fichier et tapez la commande suivante :

```
javac PremierExemple.java
```

Après la compilation et si votre programme ne comporte aucune erreur, le fichier `PremierExemple.class` sera généré.

1.3.3 Exécution

Il faut exécuter le fichier `.class` en tapant la commande (sans extension) :

```
java PremierExemple
```

Après l'exécution, le message suivant sera affiché.

Bonjour

1.3.4 Description du premier programme

Première classe

Dans Java, toutes les déclarations et les instructions doivent être faites à l'intérieure d'une classe.

`public class PremierExemple`, veut dire que vous avez déclaré une classe qui s'appelle `PremierExemple`.

Méthode main

Pour être exécuté, un programme Java doit contenir la méthode spéciale `main()`, qui est l'équivalent de `main` du langage C.

- `String[] args`, de la méthode **main** permet de récupérer les arguments transmis au programme au moment de son exécution.
- `String` est une classe. Les crochets (`[]`) indiquent que **args** est un tableau (voir plus loin pour plus d'informations sur l'utilisation des tableaux).
- Le mot clé **void**, désigne le type de retour de la méthode **main()**. Il indique que **main()** ne retourne aucune valeur lors de son appel.
- Le mot clé **static** indique que la méthode est accessible et utilisable même si aucun objet de la classe n'existe.
- Le mot clé **public** sert à définir les droits d'accès. Il est obligatoire dans l'instruction **public static void main(String[] args)** et peut être omis dans la ligne **public class PremierExemple**.

Méthode println

Sert à afficher un message, en effet dans l'instruction `System.out.println("Bonjour")` :

- **System** : est une classe
- **out** : est un objet (un flux de sortie) dans la classe `System`
- **println()** : est une méthode (fonction) dans l'objet `out`. Les méthodes sont toujours suivi de `()`.
- les points séparent les classes, les objets et les méthodes.

- chaque instruction doit se terminer par ";"
- Bonjour : est une chaîne de caractères.

1.4 Commentaires en JAVA

Les commentaires peuvent s'écrire sur une seule ligne ou sur plusieurs lignes, comme dans l'exemple suivant :

```
/*
    ceci est un commentaire
    sur plusieurs lignes
*/
public class PremierExemple {
    // ceci est un commentaire sur une seule ligne
    public static void main(String[] args) {
        System.out.println("Bonjour");
    }
}
```

1.5 Données primitifs

Java permet la déclaration de types de données primitifs.

Lors de la déclaration d'une variable sans qu'aucune valeur ne lui soit affecté, la variable sera non initialisée.

Par exemple, lors de déclaration de la variable **age** de type **int** : **int age;**

si vous essayez de l'utiliser dans une expression ou de l'afficher, vous allez recevoir une erreur lors de la compilation (contrairement au langage **C**) indiquant que la variable n'est pas initialisée (**The local variable age may not have been initialized**).

Type numérique

En java, il existe 4 types entiers et 2 types réels (table 1.1).

Type	Valeur Minimale	Valeur Maximale	Taille en octets
byte	-128	127	1
short	-32 768	32 767	2
int	-2 147 483 648	2 147 483 647	4
long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	8
float	$-3.4 * 10^{38}$	$3.4 * 10^{38}$	4
double	$-1.7 * 10^{308}$	$1.7 * 10^{308}$	8

TABLE 1.1 – Types primitifs en Java

Remarques :

1. Par défaut, une valeur comme 3.14 est de type **double**, donc pour l'affecter à une variable de type **float**, il faut la faire suivre par la lettre **f** (majuscule ou minuscule).

```
float pi=3.14F, min=10.1f;
```

2. Par défaut les entiers sont de type **int**. Pour affecter une valeur inférieure à -2147483648 ou supérieure à $2\ 147\ 483\ 647$ à un **long**, il faut la faire suivre par la lettre **l** (majuscule ou minuscule).

```
long test=4147483647L;
```

Type caractère

On utilise le type **char** pour déclarer un caractère. Par exemple :

```
char c='a';  
char etoile='*';
```

Les caractères sont codés en utilisant l'**unicode**. Ils occupent 2 octets en mémoire. Ils permettent de représenter 65536 caractères, ce qui permet de représenter (presque) la plupart des symboles utilisés dans le monde. Dans la table 1.2, on retrouve quelques séquences d'échappement.

Séquence d'échappement	Description
\n	nouvelle ligne (new line)
\r	retour à la ligne
\b	retour d'un espace à gauche
\\	\ (back slash)
\t	tabulation horizontale
\'	apostrophe
\"	guillemet

TABLE 1.2 – Séquences d'échappement

Type boolean

Il permet de représenter des variables qui contiennent les valeurs vrai (**true**) et faux (**false**).

```
double a=10, b=20;  
boolean comp;  
comp = a > b; //retourne false  
comp = a <= b; //retourne true
```

Constantes

Pour déclarer une constante, il faut utiliser le mot clé **final**. Par convention, les constantes sont écrites en majuscule.

Exemple :

```
final int MAX = 100;
final double PI = 3.14;
...
final int MAX_2 = MAX * MAX;
```

1.6 Expressions et opérateurs

Comme pour le langage C, Java possède les opérateurs arithmétiques usuels (+, -, *, /, %) et de comparaison (<, <=, >, >=, ==, !=). On retrouve aussi les expressions arithmétiques, les comparaisons et les boucles usuelles du langage C. D'autres façons propres au langage Java seront vues dans les chapitres suivants.

Exemple :

```
public class Expressions {
    public static void main(String[] args) {
        double a=10, b=20, max, min;

        max = b;
        min = a;
        if (a > b) {
            max = a;
            min = b;
        }
        System.out.println("max = " + max + " et min = " + min);

        //Carres des nombres impairs de 1 a 30
        for(int i=1; i<=30; i+=2)
            System.out.println(i+"^2 = "+i*i);
    }
}
```

1.7 Exercices

Exercice 1

Donnez le résultat des expressions suivantes :

a - $5 + 7 * 4$

b - $20 / 3 + 4$

c - $14 \% 2$

d - $6 \% 4 - 1$

Exercice 2

Donnez le résultat des expressions booléennes suivantes :

a - $4 \leq 9$ **c** - $3 + 4 == 8$ **e** - $5 != 5$ **g** - $9 != -9$
b - $12 \geq 12$ **d** - $7 < 9 - 2$ **f** - $15 != 3 * 5$ **h** - $3 + 5 * 2 == 16$

Exercice 3

Si $j = 5$ et $k = 6$, alors la valeur de $j++ == k$ est :

a - 5 **b** - 6 **c** - true **d** - false

et la valeur de $++j == k$ est :

a - 5 **b** - 6 **c** - true **d** - false

Exercice 4

Quel est le résultat du code suivant ?

```
int i, j;
for(i = 0; i < 3; i++)
    for(j = 0; j < 2; j++)
        System.out.print(i + " " + j + " ");
```

a - 0 0 0 1 1 0 1 1 2 0 2 1 **c** - 0 1 0 2 1 1 1 2
b - 0 1 0 2 0 3 1 1 1 2 1 3 **d** - 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2

Exercice 5

Qu'affiche le code suivant ?

```
int i, j;
for(i = 1, j = 4; i < j; ++i, --j)
    System.out.print(i + " " + j + " ");
```

a - 1 4 2 5 3 6 4 7... **c** - 1 4 2 3
b - 1 4 2 3 3 2 **d** - n'affiche rien

Exercice 6

Quel est le résultat du code suivant ?

```
int d = 0;
do
{
    System.out.print(d + " ");
    d++;
} while (d < 2);
```

a - 0

b - 0 1

c - 0 1 2

d - n'affiche rien

Chapitre 2

Classes et objets

2.1 Introduction

Java est un langage orientée objet. Tout doit être à l'intérieure d'une classe.

Le nom d'une classe (**identifiant**) doit respecter les contraintes suivantes :

- l'identifiant doit commencer par une lettre, `_` ou `$`. Le nom d'une classe ne peut pas commencer par un chiffre.
- un identifiant ne doit contenir que des lettres, des chiffres, `_`, et `$`;
- un identifiant ne peut être un mot réservé (voir table 2.1);
- un identifiant ne peut être un des mots suivants : `true`, `false` ou `null`. Ce ne sont pas des mots réservés mais des types primitifs et ne peuvent pas être utilisés par conséquent.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

TABLE 2.1 – mots clés

En java, il est, par convention, souhaitable de commencer les noms des classes par une lettre majuscule et utiliser les majuscules au début des autres noms pour agrandir la lisibilité des programmes.

La table 2.2 contient quelques mots qui peuvent être utilisés comme noms de classes. La table 2.3 contient quelques mots qui peuvent être utilisés comme noms de classes mais ne sont pas recommandés. Si vous utilisez un de ces mots, la compilation se passe sans problème. La table 2.4 contient quelques mots qui ne peuvent pas être utilisés comme noms de classes.

Nom de la classe	description
Etudiant	Commence par une majuscule
PrixProduit	Commence par une majuscule et le deuxième mot commence par une majuscule
AnnéeScolaire2014	Commence par une majuscule et ne contient pas d'espace

TABLE 2.2 – Quelques noms de classes valides

Nom de la classe	description
etudiant	ne commence par une majuscule
ETUDIANT	le nom en entier est en majuscule
Prix_Produit	_ n'est pas utilisé pour indiquer un nouveau mot
annéescolaire2014	ne commence par une majuscule ainsi que le deuxième, ce qui le rend difficile à lire

TABLE 2.3 – Quelques noms de classes non recommandées

Nom de la classe	description
Etudiant#	contient #
double	mot réservé
Prix Produit	contient un espace
2014annéescolaire	commence par un chiffre

TABLE 2.4 – Quelques noms de classes non valides

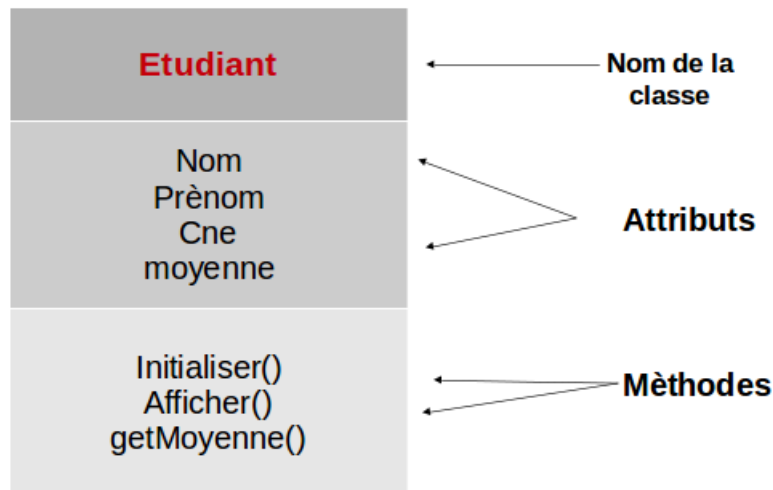


FIGURE 2.1 – Représentation en UML de la classe Etudiant

2.2 Déclaration d'une classe

Une classe peut contenir des méthodes (fonctions) et des attributs (variables). Une classe est créée en utilisant le mot clé **class**. Pour illustrer ceci, nous allons créer le prototype de la classe **Etudiant** :

```
class Etudiant {
    //Declarations
    //attributs et methodes
}
```

Remarques :

1. on peut définir plusieurs classes dans un même fichier à condition qu'une seule classe soit précédée du mot clé **public** et que le fichier porte le même nom que la classe publique ;
2. une classe peut exister dans un fichier séparé (qui porte le même nom suivi de **.java**) ;
3. pour que la machine virtuelle puisse accéder à une classe contenant la méthode main, il faut que cette classe soit publique.

2.2.1 Définition des attributs

Nous supposons qu'un étudiant est caractérisé par son nom, son prénom, son numéro cne et sa moyenne. Soit la représentation en UML illustrée par la figure 2.1. En JAVA, on peut coder les attributs de notre classe comme suit (voir aussi figure 2.2) :

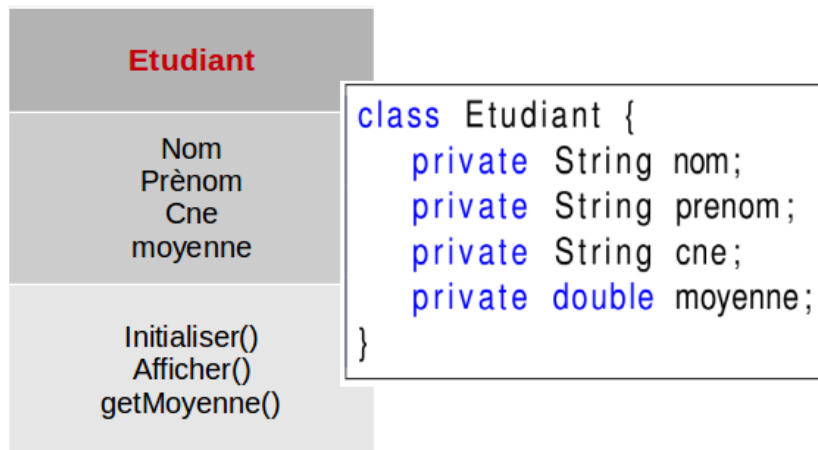


FIGURE 2.2 – Codage des attributs de la classe Etudiant

```
class Etudiant {
    private String nom;
    private String prenom;
    private String cne;
    private double moyenne;
}
```

Par convention, les noms des attributs et des méthodes doivent être en minuscule. Le premier mot doit commencer en minuscule et les autres mots doivent commencer en majuscule (**ceciEstUneVariable**, **ceciEstUneMethode**).

Remarque : la présence du mot clé **private** (privé) indique que les variables ne seront pas accessibles de l'extérieure de la classe où elles sont définies. C'est possible de déclarer les variables non privées, mais c'est déconseillé.

2.2.2 Définition des méthodes

Dans la classe étudiant, nous avons défini trois méthodes :

- **initialiser()** : qui permet d'initialiser les informations concernant un étudiant ;
- **afficher()** : qui permet d'afficher les informations concernant un étudiant ;
- **getMoyenne** : qui permet de retourner la moyenne d'un étudiant.

Voici une suite de notre code qui permet de représenter ces méthodes (voir aussi figure 2.3) :

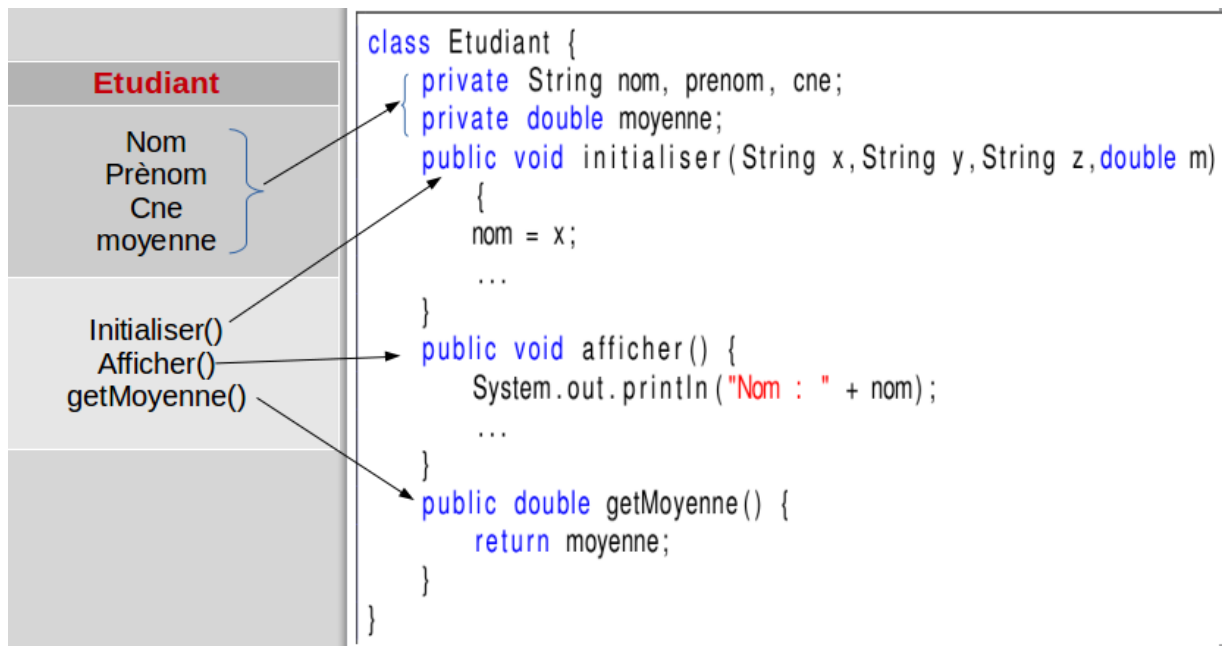


FIGURE 2.3 – Exemple de codage de la classe Etudiant

```

class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Initialiser les informations concernant l'etudiant
    public void initialiser(String x, String y, String z, double m) {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }

    //Affichage des informations concernant l'etudiant
    public void afficher() {
        System.out.println("Nom : "+nom + ", prenom : " +prenom +
            ", CNE : "+cne + " et moyenne : "+moyenne);
    }

    //retourner la moyenne
    public double getMoyenne() {
        return moyenne;
    }
}

```

Remarques :

- On peut définir plusieurs méthodes à l'intérieur d'une classe.
- La présence du mot clé **public** (publique) indique que les méthodes sont accessibles de

- l'extérieure de la classe. C'est possible de déclarer des méthodes privées.
- Il existe d'autres modes d'accès aux variables et aux méthodes qu'on verra plus loin.

2.3 Utilisation des classes

Après déclaration d'une classe, elle peut être utilisée pour déclarer un objet (variable de type classe) à l'intérieur de n'importe quelle méthode. Pour utiliser la classe étudiant

```
Etudiant et1;
```

Contrairement aux types primitifs, la déclaration précédente ne réserve pas de place mémoire pour l'objet de type **Etudiant** mais seulement une référence à un objet de type **Etudiant**. Pour réserver de la mémoire, il faut utiliser le mot clé **new** de la façon suivante :

```
et1 = new Etudiant();
```

Au lieu de ces deux instructions, vous pouvez utiliser une seule instruction :

```
Etudiant et1 = new Etudiant();
```



FIGURE 2.4 – Référence en JAVA

Les objets sont manipulés avec des **références**. Exemple classique de la télécommande et la Télévision (voir figure 2.4).

Télé = objet et Télécommande = référence.

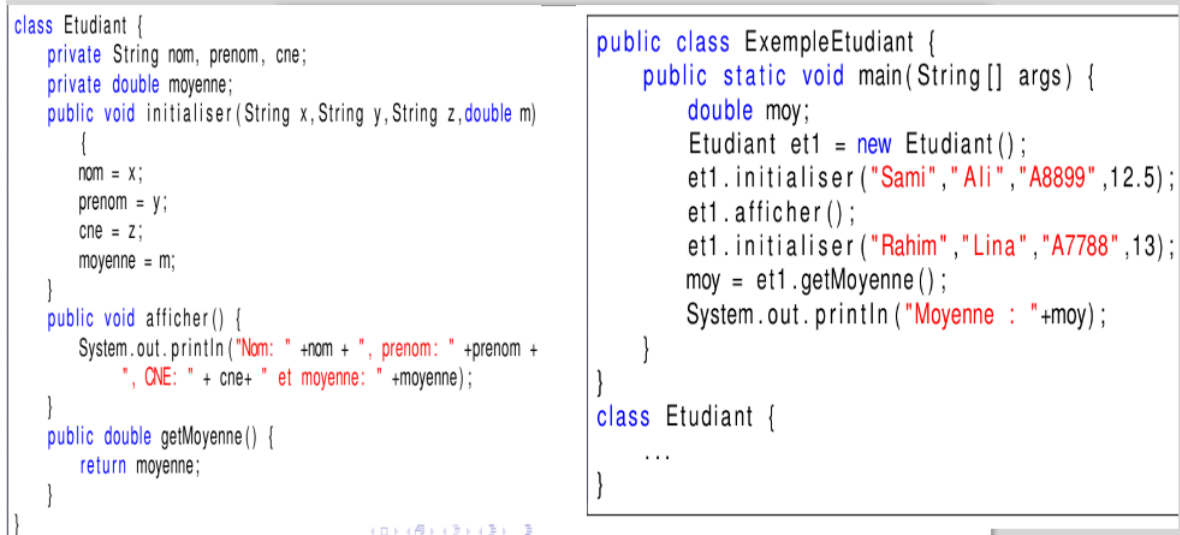
- Pour modifier l'objet (par exemple augmenter le son, ...) on utilise la référence (la télécommande);
- Avoir une référence ne veut pas dire avoir l'objet (on peut avoir la télécommande sans avoir la télé).

A présent, on peut appliquer n'importe quelle méthode à l'objet **et1**. par exemple, pour initialiser les attributs de **et1**, on procède de la façon suivante :

```
et1.initialiser("Salim", "Mohammed", "A8899", 12.5);
```

Exemple :

Dans l'exemple suivant, on utilisera la classe **ExempleEtudiant**, pour tester la classe **Etudiant**, c'est notre classe principale (qui contient le `main()`) :



```

class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;
    public void initialiser(String x,String y,String z,double m)
    {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }
    public void afficher() {
        System.out.println("Nom: " +nom + ", prenom: " +prenom +
            ", CNE: " + cne+ " et moyenne: " +moyenne);
    }
    public double getMoyenne() {
        return moyenne;
    }
}

public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et1 = new Etudiant();
        et1.initialiser("Sami", "Ali", "A8899",12.5);
        et1.afficher();
        et1.initialiser("Rahim", "Lina", "A7788",13);
        moy = et1.getMoyenne();
        System.out.println("Moyenne : "+moy);
    }
}

class Etudiant {
    ...
}

```

FIGURE 2.5 – La classe Etudiant et la classe principale ExempleEtudiant

```

public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et1 = new Etudiant();
        et1.initialiser("Salim", "Mohammed", "A8899",12.5);
        et1.afficher();
        et1.initialiser("Oujdi", "Ahmed", "A7788",13);
        moy = et1.getMoyenne();
        System.out.println("Moyenne : "+moy);
        et1.afficher();
    }
}

class Etudiant {
    ...
}

```

Exécution :

Le résultat de l'exécution du programme précédent est le suivant :

Nom : Salim, prenom : Mohammed, CNE : A8899 et moyenne : 12.5

Moyenne : 13.0

Nom : Oujdi, prenom : Ahmed, CNE : A7788 et moyenne : 13.0

2.4 Portée des attributs

Les attributs sont accessibles à l'intérieure de toutes les méthodes de la classe. Il n'est pas nécessaire de les passer comme arguments.

À l'extérieure des classes, les attributs privés ne sont pas accessibles. Pour l'exemple de la classe **ExempleEtudiant**, une instruction de type :

```
Etudiant et = new Etudiant();  
moy = et.moyenne;
```

aboutit à une erreur de compilation (**The field Etudiant.moyenne is not visible**). En effet l'attribut **moyenne** est accessible à l'intérieur de la classe **Etudiant** mais non plus à l'intérieur de la classe **ExempleEtudiant**; c'est pourquoi qu'on a utilisé, dans l'exemple précédent, la méthode **getMoyenne()** pour accéder à la moyenne.

2.5 Surcharge des méthodes

On parle de surcharge, lorsque plusieurs méthodes possèdent le même nom. Ces différentes méthodes ne doivent pas avoir le même nombre d'arguments ou des arguments de même types.

Exemple

On ajoutera à la classe **Etudiant** trois méthodes qui portent le même nom. Une méthode qui contient :

1. trois arguments de types **double**;
2. deux arguments de types **double**;
3. deux arguments de types **float**.


```

class Etudiant {
    ...
    //calcul de la moyenne de trois nombres
    public double calculMoyenne(double m1, double m2, double m3) {
        double moy = (m1 + m2 + m3)/3;
        return moy;
    }

    //calcul de la moyenne de deux nombres (doubles)
    public double calculMoyenne(double m1, double m2) {
        double moy = (m1 + m2)/2;
        return moy;
    }

    //calcul de la moyenne de deux nombres (float)
    public double calculMoyenne(float m1, float m2) {
        double moy = (m1 + m2)/2;
        return moy;
    }
}

```

Utilisation de la classe Etudiant

Maintenant, on utilisera la classe principale **ExempleEtudiant**, pour tester la classe modifiée **Etudiant** :

```

public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et1 = new Etudiant();
        et1.initialiser("Salim", "Mohammed", "A8899", 12.5);

        //Appel de calculMoyenne(double m1, double m2, double m3)
        moy = et1.calculMoyenne(10.5, 12, 13.5);

        //Appel de calculMoyenne(double m1, double m2)
        moy = et1.calculMoyenne(11.5, 13);

        //Appel de calculMoyenne(float m1, float m2)
        moy = et1.calculMoyenne(10.5f, 12f);

        //Appel de calculMoyenne(double m1, double m2)
        //13.5 est de type double
        moy = et1.calculMoyenne(11.5f, 13.5);
    }
}

```

Conflit

Considérons l'exemple suivant :

```
class Etudiant {
    ...
    public double calculMoyenne(double m1, float m2) {
        return (m1 + m2)/2;
    }

    public double calculMoyenne(float m1, double m2) {
        return (m1 + m2)/2;
    }
}
```

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et1 = new Etudiant();
        et1.initialiser("Salim", "Mohammed", "A8899", 12.5);

        //Appel de calculMoyenne(double m1, float m2)
        moy = et1.calculMoyenne(11.5, 13f);

        //Appel de calculMoyenne(float m1, double m2)
        moy = et1.calculMoyenne(10.5f, 12.0);

        //11.5 et 13.5 sont de type double
        //Erreur de compilation
        moy = et1.calculMoyenne(11.5, 13.5);

        //11.5f et 13.5f sont de type float
        //Erreur de compilation
        moy = et1.calculMoyenne(11.5f, 13.5f);
    }
}
```

L'exemple précédent conduit à des erreurs de compilation :

- `moy = et.calculMoyenne(11.5, 13.5)` aboutit à l'erreur de compilation : The method `calculMoyenne(double, float)` in the type `Etudiant` is not applicable for the arguments `(double, double)`;
- `moy = et.calculMoyenne(11.5f, 13.5f)` aboutit à l'erreur de compilation : The method `calculMoyenne(double, float)` is ambiguous for the type `Etudiant`.

2.6 Méthodes statiques et finales

2.6.1 Attributs statiques

Les variables statiques sont appelés **variables de classe**. Elles sont partagées par toutes les instances de la classe. Pour chaque instance de la classe, il n'y a qu'une seule copie d'une variable statique par classe. Il n'y a pas de création d'une nouvelle place mémoire lors de l'utilisation de « **new** ». Pour déclarer une variable statique, il faut utiliser le mot clé **static**.

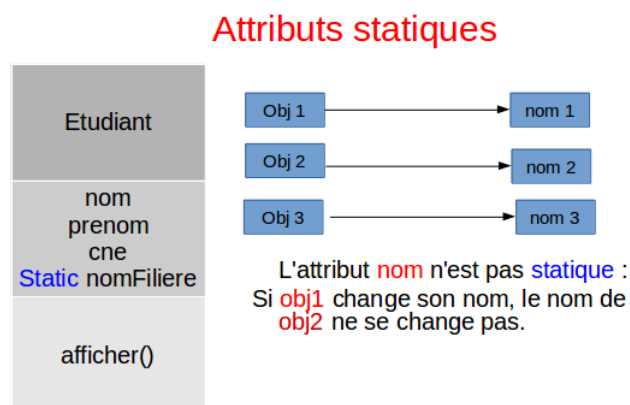


FIGURE 2.6 – Avec un attribut non static

Exemple :

```
public class VarStatiques {
    public static void main(String[] args) {
        System.out.println(Jeu.meilleurScore); // Affiche 0.
        Jeu.meilleurScore++;
        System.out.println(Jeu.meilleurScore); // Affiche 1
        Jeu j = new Jeu();
        j.calculScore();
        System.out.println(Jeu.meilleurScore); // Affiche 10
        //ou bien
        System.out.println(j.meilleurScore);   // Affiche 10
        j.calculScore();
        System.out.println(Jeu.meilleurScore); // Affiche 20
    }
}
class Jeu {
    static int meilleurScore = 0;
    int score = 0;
    void calculScore() {
        score += 10;
        if (meilleurScore < score)    meilleurScore = score;
    }
}
```

}

Les figures 2.6 et 2.7 montrent clairement le comportement d'une classe (des objets) vis-à-vis un attribut déclaré `static` ou non.

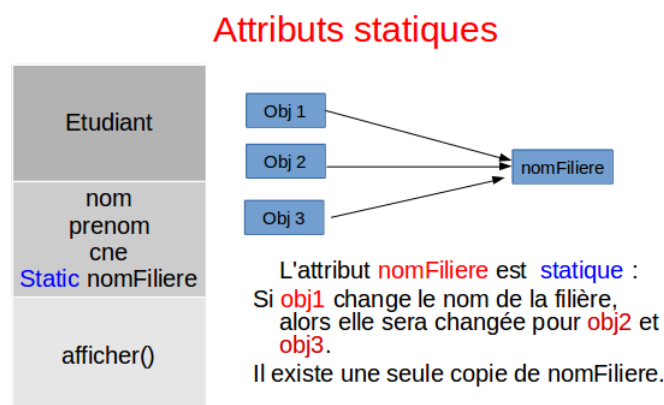


FIGURE 2.7 – Avec un attribut static

2.6.2 Constantes `final` et `static`

Une constante commune à toutes les instances d'une classe peut être déclarée en « `final static` ».

```
class A{  
    final static double PI=3.1415927;  
    static final double Pi=3.1415927;  
}
```

La classe « **Math** » fournit les constantes statiques **Math.PI** (égale à 3.14159265358979323846) et **Math.E** (égale à 2.7182818284590452354).

Dans la classe « **Math** », la déclaration de **PI** est comme suit :

```
public final static double PI = 3.14159265358979323846;
```

PI est :

- `public`, elle est accessible par tous ;
- `final`, elle ne peut pas être changée ;
- `static`, une seule copie existe et elle est accessible sans déclarer d'objet `Math`.

2.6.3 Méthodes statiques

Une méthode peut être déclarée statique en la faisant précéder du mot clé `static`. Elle peut être appelée directement sans créer d'objet pour cette méthode. Elle est appelée **méthode de classe**.

Il y a des restrictions sur l'utilisation des méthodes statiques :

- elles ne peuvent appeler que les méthodes statiques ;
- elles ne peuvent utiliser que les attributs statiques ;

— elles ne peuvent pas faire référence à `this` et à `super`.

Exemple (une méthode statique ne peut utiliser que les attributs statiques) :

```
public class MethodesClasses {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int n;
        System.out.print("Saisir un entier : ");
        n = clavier.nextInt();
        System.out.print("Factorielle : " + n + " est : "
            + Calcul.factorielle(n));
    }
}

class Calcul {
    private int somme;
    static int factorielle(int n) {
        //ne peut pas utiliser somme
        if (n <= 0)
            return 1;
        else
            return n * factorielle(n - 1);
    }
}
```

La classe « **Math** » fournit les méthodes statiques **sin**, **cos**, **pow**, ...

2.7 Lecture à partir du clavier

Pour lire à partir du clavier, il existe la classe **Scanner**. Pour utiliser cette classe, il faut la rendre visible au compilateur en ajoutant la ligne :

```
import java.util.Scanner;
```

Parmi les méthodes de la classe **Scanner**, on trouve :

<code>nextShort()</code>	permet de lire un short
<code>nextByte()</code>	permet de lire un byte
<code>nextInt()</code>	permet de lire un int
<code>nextLong()</code>	permet de lire un long . Il n'est pas nécessaire d'ajouter L après l'entier saisi.
<code>nextFloat()</code>	permet de lire un float . Il n'est pas nécessaire d'ajouter F après le réel saisi.
<code>nextDouble()</code>	permet de lire un double
<code>nextLine()</code>	permet de lire une ligne et la retourne comme un String
<code>next()</code>	permet de lire la donnée suivante comme String

Exemple :

```
import java.util.Scanner;
public class TestScanner
{
    public static void main(String[] args)
    {
        String nom;
        int age;
        double note1 , note2 , moyenne;
        //clavier est un objet qui permette la saisie au clavier
        //vous pouvez utiliser un autre nom (keyb, input, ...)
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir votre nom : ");
        nom = clavier.nextLine();
        System.out.print("Saisir votre age : ");
        age = clavier.nextInt();
        System.out.print("Saisir vos notes : ");
        note1 = clavier.nextDouble();
        note2 = clavier.nextDouble();
        moyenne = (note1+note2)/2;
        System.out.println("Votre nom est " + nom + ", vous avez "
            + age + " ans et vous avez obtenu "+ moyenne);
        clavier.close(); //fermer le Scanner
    }
}
```

Problèmes liés à l'utilisation de nextLine()

Reprenons l'exemple précédent et au lieu de commencer par la saisie du nom, on commence par la saisie de l'age.

```

import java.util.Scanner;
public class TestScanner
{
    public static void main(String[] args)
    {
        String nom;
        int age;
        double note1, note2, moyenne;
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir votre age : ");
        age = clavier.nextInt();
        System.out.print("Saisir votre nom : ");
        nom = clavier.nextLine();
        System.out.print("Saisir vos notes : ");
        note1 = clavier.nextDouble();
        note2 = clavier.nextDouble();
        moyenne = (note1+note2)/2;
        System.out.println("Votre nom est " + nom + ", vous avez "
            + age + " ans et vous avez obtenu " + moyenne);
    }
}

```

L'exécution de l'exemple précédent est la suivante :

```

Saisir votre age : 23
Saisir votre nom : Saisir vos notes : 12
13
Votre nom est , vous avez 23 ans et vous avez obtenu 12.5

```

Lors de la saisie de l'age, on a validé par **Entrée**. La touche **Entrée** a été stocké dans **nom** !
 Pour éviter ce problème, il faut mettre `clavier.nextLine()` avant `nom = clavier.nextLine()`;

```

{
    public static void main(String[] args)
    {
        ...
        age = clavier.nextInt();
        System.out.print("Saisir votre nom : ");
        clavier.nextLine();
        nom = clavier.nextLine();
        ...
    }
}

```

2.8 Chaines de caractères

Considérons l'exemple suivant :

```
import java.util.Scanner;

public class TestComparaisonChaines {
    public static void main(String[] args) {
        String nom = "smi", str;
        Scanner input = new Scanner(System.in);
        System.out.print("Saisir le nom : ");
        str = input.nextLine();
        if (str == nom)
            System.out.println(nom + " = " + str);
        else
            System.out.println(nom + " est different de " + str);
    }
}
```

Dans cet exemple, on veut comparer les deux chaînes de caractères *nom* initialisée avec "smi" et *str* qui est saisie par l'utilisateur.

Lorsque l'utilisateur exécute le programme précédent, il aura le résultat suivant :

Saisir le nom : smi

smi est different de smi

Il semble que le programme précédent produit des résultats incorrects. Le problème provient du fait, que dans java, **String** est une classe et par conséquent chaque chaîne de caractères est un objet. L'utilisation de l'opérateur == implique la comparaison entre les références et non des contenus.

L'exemple corrigé

```
import java.util.Scanner;

public class TestComparaisonChaines {
    public static void main(String[] args) {
        String nom = "smi", str;
        Scanner input = new Scanner(System.in);
        System.out.print("Saisir le nom : ");
        str = input.nextLine();
        if (str.equals(nom))
            System.out.println(nom + " = " + str);
        else
            System.out.println(nom + " est different de " + str);
    }
}
```

Affichage des objets

Le code suivant :

```
Etudiant etud = new Etudiant("Sami", "Ali", "A20");
System.out.println(etud);
```


affichera quelque chose comme : *Etudiant@1b7c680*. Cette valeur correspond à la référence de l'objet. Si on souhaite afficher le contenu de l'objet en utilisant le même code, il faut utiliser la méthode **toString**.

```
public class TestEtudiantToString {
    public static void main(String[] args) {
        Etudiant etud = new Etudiant("Sami", "Ali", "A20");
        System.out.println(etud);
    }
}
class Etudiant {
    private String nom, prenom, cne;
    ...
    public String toString(){
        return "Nom: "+nom+", Prenom: "+prenom+" et CNE: "+cne;
    }
    ...
}
```

Ce code affichera :

Nom: Sami, Prenom: Ali et CNE: A20

2.9 Exercices

Exercice 1

Parmi les identifiants suivants, quels sont ceux qui sont valides ?

- | | | | |
|---------------------|-------------|------------|---------------|
| a - nomEtudiant | c - static | e - 23code | g - serie# |
| b - prenom Etudiant | d - BONJOUR | f - code13 | h - Test_Comp |

Exercice 2

Soient **ClasseA** et **ClasseB**, deux classes déclarées dans le même fichier :

```

public class ClasseA {
    private double x;
}

public class ClasseB {
    private int n;

    void meth() {
        System.out.println("n = " + n);
    }
}

```

1. La déclaration précédente est-elle correcte ? Sinon, corrigez la.
2. Donnez un nom au fichier.
3. Quels sont les fichiers générés après compilation ?

Exercice 3

Considérons le programme java suivant :

```

public class Point {
    public int x;
    private int y;
    public static void main(String[] args) {
        Point pA;
        pA.x= 2;
        pA.y= 3;
        System.out.println("abscisse= " + pA.x + " ordonnee = "+ pA.y);
    }
}

```

1. Sous quel nom de fichier devez-vous enregistrer ce programme ?
2. Corrigez les erreurs syntaxiques.

Exercice 4

Supposons qu'une classe nommée **ClasseA** contient un attribut entier privé nommé **b**, un attribut entier public nommé **c** et un attribut entier public statique nommé **d**. Dans une méthode **main()**, d'une autre classe, on instancie un objet de la façon suivante :

ClasseA objA = new ClasseA();

Parmi les déclarations suivantes, quelles sont celles qui sont valides ?

- | | | |
|--------------------|--------------------|--------------------|
| 1. objA.b = 10; | 3. objA.c = 30; | 5. objA.d = 50; |
| 2. ClasseA.b = 20; | 4. ClasseA.c = 40; | 6. ClasseA.d = 60; |

Exercice 5

Supposons que vous avez créé un programme qui contient seulement les variables :

```
int v = 4;  
int w = 6;  
double x = 2.2;
```

Supposons aussi que vous avez une méthode avec l'entête :

```
public static void calculer(int x, double y)
```

Parmi les déclarations suivantes, quelles sont celles qui sont valides ?

a – calculer(v, w);	c – calculer(x, y);	e – calculer(1.1, 2.2);
b – calculer(v, x);	d – calculer(18, x);	f – calculer(5, 7);

Chapitre 3

Constructeurs

3.1 Introduction

On a vu dans le chapitre 2, que pour initialiser les attributs de la classe **Etudiant**, on a défini une méthode **initialiser()**.

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Initialiser les informations concernant l'etudiant
    public void initialiser(String x, String y, String z, double m) {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }
    ...
}
```

De plus dans la classe principale **ExempleEtudiant** et pour l'instancier, on était obligé d'appeler une méthode spéciale **Etudiant()** : *Etudiant et1 = new Etudiant();*

```

public class ExempleEtudiant {
    public static void main(String[] args) {
        double moy;
        Etudiant et1 = new Etudiant();
        et1.initialiser("Salim", "Mohammed", "A8899", 12.5);
        et1.afficher();
    }
}

class Etudiant {
    ...
}

```

Cette façon de faire n'est pas conseillé pour les raisons suivantes :

- on n'a pas profité de la méthode spéciale **Etudiant()** ;
- pour chaque objet créé, on doit l'initialiser en appelant la méthode d'initialisation ;
- si on oublie d'initialiser l'objet, il sera initialisé par défaut, ce qui peut poser des problèmes lors de l'exécution du programme, du fait que le programme sera compilé sans erreurs.

Pour remédier à ces inconvénients, on utilise les constructeurs.

3.2 Définition

Un **constructeur** est une méthode, sans type de retour, qui porte le même nom que la classe. Il est invoqué lors de la déclaration d'un objet.

Une classe peut avoir plusieurs constructeurs (**surcharge**), du moment que le nombre d'arguments et leurs types ne sont pas les mêmes.

Exemple

```

class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    // Constructeur
    public Etudiant(String x, String y, String z, double m) {
        nom = x;
        prenom = y;
        cne = z;
        moyenne = m;
    }
    ...
}

```

Pour créer un objet et l'initialiser, on remplace les deux instructions :

```
Etudiant et1 = new Etudiant();  
et1.initialiser("Salim", "Mohammed", "A8899", 12.5);  
par l'instruction :  
Etudiant et1 = new Etudiant("Salim", "Mohammed", "A8899", 12.5);
```

3.3 Utilisation de **this**

Dans les arguments du constructeur **Etudiant**, on a utilisé : x, y, z et m. On peut utiliser les mêmes noms que les attributs privés de la classe en faisant appel au mot clés **this**.

Exemple

```
class Etudiant {  
    private String nom, prenom, cne;  
    private double moyenne;  
  
    //Constructeur  
    public Etudiant(String nom, String prenom, String cne,  
                    double moyenne) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.cne = cne;  
        this.moyenne = moyenne;  
    }  
    ...  
}
```

Dans `this.nom = nom`, this.nom correspond aux attribut `nom` de la classe et nom correspond à la variable locale. En général, Le mot-clé `this` désigne, dans une classe, l'instance courante de la classe elle-même.

3.4 Surcharge des constructeurs

On va modifier la classe **Etudiant** pour qu'il possède deux constructeurs :

1. un à trois arguments ;
2. l'autre à quatre arguments.

Exemple 1

```
class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Constructeur 1
    public Etudiant(String nom, String prenom, String cne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
    }

    //Constructeur 2
    public Etudiant(String nom, String prenom, String cne,
        double moyenne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
        this.moyenne = moyenne;
    }
    ...
}
```

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        Etudiant et = new Etudiant("Salim", "Mohammed", "A8899", 12.5);

        //l'attribut moyenne est initialisé par défaut (0.0)
        Etudiant et1 = new Etudiant("Oujdi", "Ahmed", "A7799");
    }
}
```

Exemple 2

Dans le constructeur 2 de l'exemple 1, trois instructions ont été répétées. Pour éviter cette répétition, on utilise l'instruction **this(arguments)**. L'exemple 1 devient :

```

class Etudiant {
    private String nom, prenom, cne;
    private double moyenne;

    //Constructeur 1
    public Etudiant(String nom, String prenom, String cne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
    }

    //Constructeur 2
    public Etudiant(String nom, String prenom, String cne,
        double moyenne) {
        //Appel du constructeur 1
        this(nom, prenom, cne);
        this.moyenne = moyenne;
    }
    ...
}

```

L'instruction **this(arguments)** doit être la première instruction du constructeur. Si elle est mise ailleurs, le compilateur génère une erreur.

3.5 Constructeur par défaut

Le constructeur par défaut est un constructeur qui n'a pas d'arguments.

Exemple

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        //Utilisation du constructeur par défaut
        Etudiant et1 = new Etudiant();
    }
}

class Etudiant {
    ...
    //Constructeur par défaut
    public Etudiant() {
        nom = "";
        prenom = "";
        cne = "";
        moyenne = 0.0;
    }

    //Autres constructeurs
    ...
}
```

Remarques :

1. Si aucun constructeur n'est utilisé, le compilateur initialise les attributs aux valeurs par défaut.
2. Dans les exemples 1 et 2 de la section 3.4, l'instruction `Etudiant et1 = new Etudiant();` n'est pas permise parce que les deux constructeurs ont des arguments.
3. Un constructeur ne peut pas être appelé comme les autres méthodes. L'instruction `et1.Etudiant("Oujdi", "Mohammed", "A8899");` n'est pas permise.

3.6 Constructeur de copie

Java offre un moyen de créer la copie d'une instance en utilisant le constructeur de copie. Ce constructeur permet d'initialiser une instance en copiant les attributs d'une autre instance du même type.

Exemple

```
public class ExempleEtudiant {
    public static void main(String[] args) {
        Etudiant et1 = new Etudiant("Oujdi", "Mohammed", "A8899");
        Etudiant et2 = new Etudiant(et1);
    }
}

class Etudiant {
    ...
    //Constructeur de copie
    public Etudiant(Etudiant autreEt) {
        nom = autreEt.nom;
        prenom = autreEt.prenom;
        cne = autreEt.cne;
        moyenne = autreEt.moyenne;
    }
    ...
}
```

Remarque

et1 et **et2** sont deux références différentes pour deux objets ayant les mêmes valeurs d'attributs.

3.7 Exercices

Exercice 1

Corrigez les erreurs dans le programme suivant :

```
public class Boite
{
    private int largeur;
    private int longueur;
    private int hauteur;
    private BoiteFixe()
    {
        longueur = 1;
        largeur = 1;
        hauteur = 1;
    }
    public Boite(int largeur, int longueur, int hauteur)
    {
        largeur = largeur;
        longueur = longueur;
    }
}
```

```

        hauteur = hauteur;
    }
    public void afficher()
    {
        System.out.println("largeur: " + largeur + " longueur: " +
            longueur + " hauteur: " + hauteur);
    }
    public double getVolume()
    {
        double volume = longueur * largeur * hauteur;
        return volume;
    }
}

```

Exercice 2

1. Créez la classe **Cercle** qui contient les attributs **rayon**, **diametre** et **surface**. La classe contient un constructeur qui initialise le rayon et calcule les autres attributs ($S = \pi r^2$ et $D = 2\pi r$, utilisez la constante `Math.PI` pour le calcul). La classe contient aussi les méthodes **setRayon** et **getRayon**.
2. Créez une classe nommée **TestCercle** qui contient la méthode **main()**. Dans cette méthode :
 - a – déclarez deux objets **Cercle**;
 - b – en utilisant la méthode **setRayon**, modifiez les valeurs des deux cercles;
 - c – affichez les informations concernant chaque cercle.

Chapitre 4

Héritage

4.1 Introduction

Comme pour les autres langages orientés objets, Java permet la notion d'héritage qui permet de créer de nouvelles classes à partir d'autres classes existantes. L'héritage permet de réutiliser des classes déjà définies en adaptant les attributs et les méthodes (par ajout et/ou par modification).

Une classe qui hérite d'une classe existante est appelée classe **dérivée**, **sous-classe** ou **classe-fille**. La classe, dont hérite d'autres classes, est appelée classe **super-classe**, **classe-mère** ou **classe-parente**.

Syntaxe

```
class SousClasse extends SuperClass
```

Remarques :

- Java ne permet pas l'héritage multiple. C'est-à-dire, une classe ne peut pas hériter de plusieurs classes. Elle ne peut hériter que d'une seule classe (Fig. 4.1 à droite).
- Une classe peut hériter d'une classe dérivée. Considérons la classe **A** qui est la **super-classe** de **B** et **B** qui est la **super-classe** de **C**. **A** est la **super-super-classe** de **C** (Fig. 4.1 à gauche).

4.2 Exemple introductif

Considérons les deux classes : **Etudiant** et **Professeur** (Fig. 4.2). Pour les deux classes :

1. les attributs **nom** et **prenom** sont en commun ;
2. les méthodes **afficher()** et **setNom()** sont en commun ;
3. la classe **Etudiant** contient l'attribut **cne** et la classe **Professeur** contient l'attribut **som**.

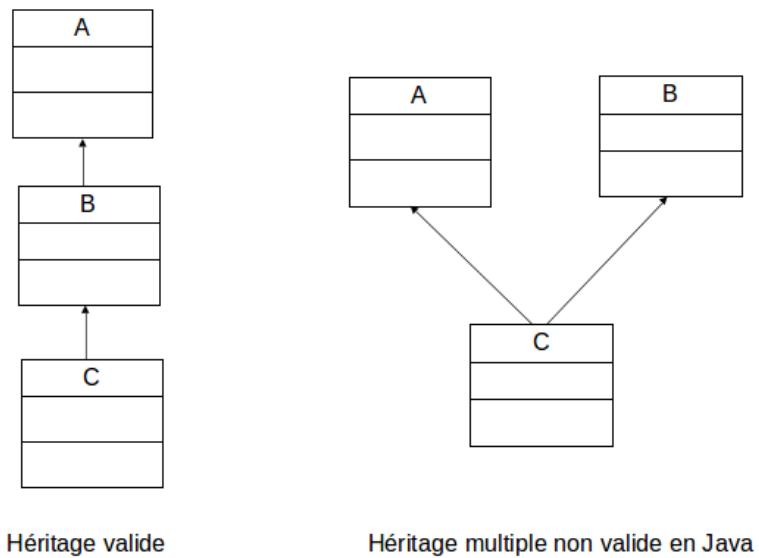


FIGURE 4.1 – L'héritage multiple est non valide en Java

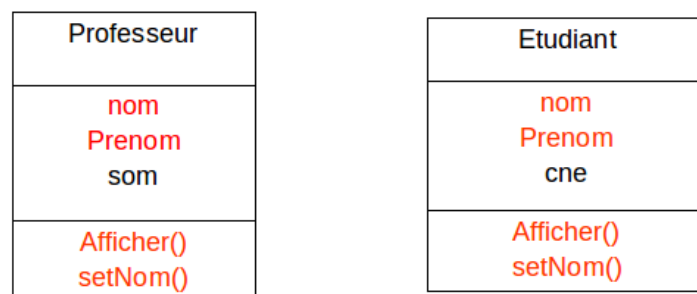


FIGURE 4.2 – Presentation en UML des deux classes Etudiant et Professeur

```

class Etudiant {
    private String nom, prenom, cne;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}

class Professeur {
    private String nom, prenom, som;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
    void setSom(String som){
        this.som = som;
    }
}

```

Listing 4.1 – Classes Etudiant et professeur

4.3 Utilisation de l’héritage

Un étudiant et un professeur sont des personnes. Définissons une nouvelle classe **Personne** (listing 4.2) :

```

class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom :"+nom);
        System.out.println("Prenom :"+prenom);
    }
    void setNom(String nom){
        this.nom = nom;
    }
}

```

Listing 4.2 – Classe Personne

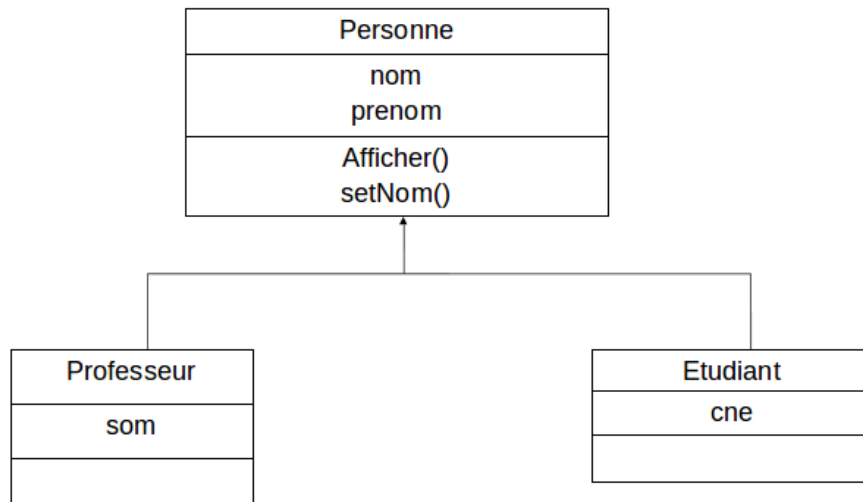


FIGURE 4.3 – Presentation en UML de l’héritage

Les deux classes peuvent être modifiées en utilisant la classe **Personne** . Elles deviennent comme le montre la figure 4.3 et le décrit le listing 4.3.

```

class Etudiant extends Personne {
    private String cne;
    void setCne(String cne){
        this.cne = cne;
    }
}

class Professeur extends Personne{
    private String som;

    void setSom(String som){
        this.som = som;
    }
}
  
```

Listing 4.3 – Classes Professeur et Etudiant héritent de la classe Personne

4.4 Accès aux attributs

L’accès aux attributs privés (**private**) d’une super-classe n’est pas permis d’une façon directe. Supposons qu’on veut définir, dans la classe **Etudiant**, une méthode **getNom()** qui retourne le nom alors, l’instruction suivante n’est pas permise car le champ **Personne.nom** est non visible (The field **Personne.nom** is not visible).

```

class Etudiant extends Personne {
    private String cne;
    String getNom() {
        return nom; //non permise
    }
}

```

Pour accéder à un attribut d'une **super-classe**, il faut soit :

- rendre l'attribut publique, ce qui implique que ce dernier est accessible par toutes les autres classes (déconseillé);
- définir, dans la classe **Personne**, des méthodes qui permettent d'accéder aux attributs privés (**getters** et **setters**);
- déclarer l'attribut comme protégé en utilisant le mot clé **protected**.

Exemple

Dans le listing 4.4, la classe **Etudiant** peut accéder à l'attribut **nom** puisqu'il est protégé.

```

class Personne {
    protected String nom;
    ...
}
class Etudiant extends Personne {
    ...
    String getNom() {
        return nom;
    }
}

```

Listing 4.4 – Accès protégé aux attributs

Remarques :

1. Un attribut protégé est accessible par toutes les sous-classes et par toutes les classes du même paquetage (on verra plus loin la notion de **package**) ce qui casse l'encapsulation.
2. Le mode protégé n'est pas très utilisé en Java.

4.5 Héritage hiérarchique

Comme mentionné dans l'introduction, une classe peut être la **super-super-classe** d'une autre classe. Reprenons l'exemple du listing 4.3 et ajoutons la classe **EtudiantEtranger**. Un étudiant étranger est lui aussi un étudiant dont on veut lui ajouter la nationalité (listing 4.5 et Fig. 4.4).

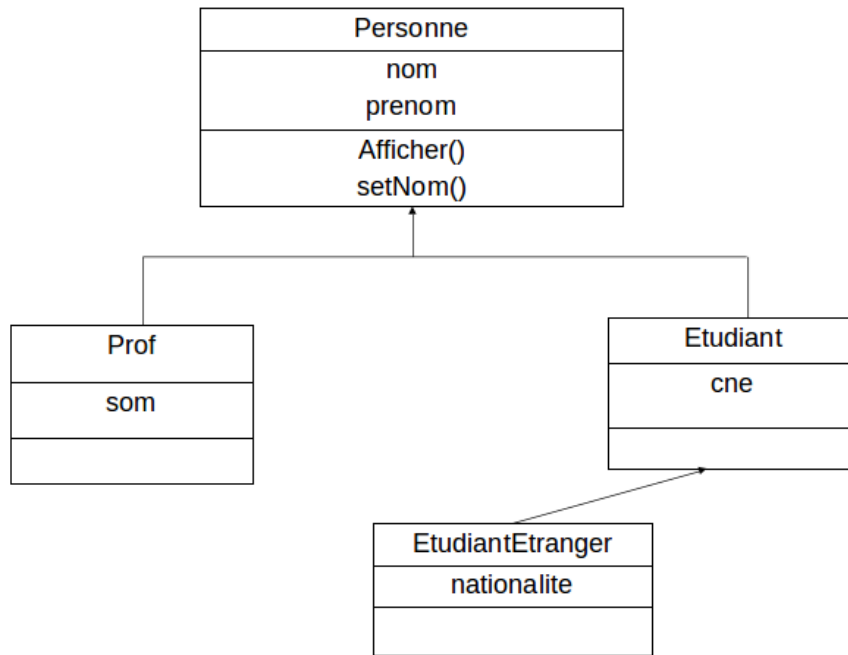


FIGURE 4.4 – Héritage hiérarchique

```

class Personne {
    ...
}
class Etudiant extends Personne {
    ...
}
class EtudiantEtranger extends Etudiant {
    private String nationalite;
    ...
}
  
```

Listing 4.5 – **EtudiantEtranger** hérite de **Etudiant** et **Etudiant** hérite de **Personne**

4.6 Redéfinition

Définition

- **Redéfinition (overriding)** : comme pour le cas de surcharge à l'intérieure d'une classe, une méthode déjà définie dans une super-classe peut avoir une nouvelle définition dans une sous-classe (listings 4.6 et 4.7).

Remarques :

1. Il ne faut pas confondre surcharge et redéfinition !
2. On verra plus de détails concernant la redéfinition dans le chapitre concernant le polymorphisme.

Exemple 1

```
class A
{
    public void f(int a, int b)
    {
        //instructions
    }
    //Autres methodes et attributs
}

class B extends A
{
    public void f(int a, int b)
    {
        //la methode redefinie f() de la super-classe
    }
    //Autres methodes et attributs
}
```

Listing 4.6 – Redéfinition de la méthode f()

Exemple 2

Reprenons l'exemple du listing 4.5 et ajoutons à la classe **Personne** la méthode **afficher()** qui permet d'afficher le nom et le prénom. Dans les classes **Etudiant**, **EtudiantEtranger**, et **Professeur**, la méthode **afficher()** peut être redéfinie avec le même nom et sera utilisé pour afficher les informations propres à chaque classe. Pour ne pas répéter les instructions se trouvant dans la méthode de base, il faut utiliser le mot clé **super** (listing 4.7).

```
class Personne {
    private String nom, prenom;

    void afficher() {
        System.out.println("Nom : "+nom);
        System.out.println("Prenom : "+prenom);
    }
    ...
}

class Etudiant extends Personne {
    private String cne;
```

```

        void afficher() {
            super.afficher();
            System.out.println("CNE :"+cne);
        }
        ...
    }
    class EtudiantEtranger extends Etudiant {
        private String nationalite;
        void afficher() {
            super.afficher();
            System.out.println("Nationalite :"+nationalite);
        }
        ...
    }

    class Professeur extends Personne{
        private String som;
        void afficher() {
            super.afficher();
            System.out.println("SOM :"+som);
        }
        ...
    }
}

```

Listing 4.7 – Redéfinition de la méthode **afficher()**

Remarques :

1. La méthode **super.afficher()** doit être la première instruction dans la méthode **afficher()**.
2. La méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Etudiant**.
3. Si la classe **Etudiant** n'avait pas la méthode **afficher()**, alors par transitivité, la méthode **super.afficher()** de la classe **EtudiantEtranger**, fait appel à **afficher()** de la classe **Personne**.
4. Il n'y a pas de : **super.super**.

4.7 Héritage et constructeurs

Une sous-classe n'hérite pas des constructeurs de la super-classe.

Exemple 1

Reprenons l'exemple du listing 4.5 et ajoutons à la classe **Personne** un seul constructeur.

Si aucun constructeur n'est défini dans les classes **Etudiant** et **Professeur**, il y aura erreur de compilation (listing 4.8).

```

class Personne {
    private String nom, prenom;

    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}

class Etudiant extends Personne {
    ...
    private String cne;
    //Pas de constructeur
    ...
}

class Professeur extends Personne{
    ...
    private String cne;
    //Pas de constructeur
    ...
}

```

Listing 4.8 – Héritage et constructeurs

Exemple 2

```

class Personne {
    private String nom, prenom;

    //Constructeur
    public Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    ...
}

class Etudiant extends Personne {
    ...
    private String cne;
    //Constructeur
    public Etudiant(String nom, String prenom, String cne) {
        super(nom, prenom);
        this.cne = cne;
    }
}

```

```

    }
    ...
}

class EtudiantEtranger extends Etudiant {
    private String nationalite;
    //Constructeur
    public EtudiantEtranger(String nom, String prenom,
        String cne, String nationalite) {
        super(nom, prenom, cne);
        this.nationalite = nationalite;
    }
    ...
}

```

Listing 4.9 – Héritage multiple et constructeurs

Dans cet exemple, le constructeur de la classe **EtudiantEtranger** fait appel au constructeur de la classe **Etudiant** qui à son tour fait appel au constructeur de la classe **Personne**.

Exemple 3

```

class Rectangle{
    private double largeur;
    private double hauteur;
    public Rectangle(double l, double h) {
        largeur = l;
        hauteur = h;
    }
    ...
}

class Carre extends Rectangle {
    public Carre(double taille) {
        super(taille, taille);
    }
    ...
}

```

Listing 4.10 – Héritage et constructeurs

Remarques :

1. **super** doit être la première instruction dans le constructeur et ne doit pas être appelé deux fois.
2. Il n'est pas nécessaire d'appeler **super** lorsque la super-classe admet un constructeur par défaut. Cette tâche sera réalisée par le compilateur.

3. Les arguments de super doivent être ceux d'un des constructeur de la super-classe.
4. Aucune autre méthode ne peut appeler super(...).
5. Il n'y a pas de : **super.super**.

Opérateur « instanceof »

Pour les types primitifs, les instructions suivantes sont vraies :

```
int i;  
float x;  
double y;  
...  
x = i;  
y = x;
```

Un **int** est un **float** et un **float** est un **double**.

Par contre, les instructions suivantes sont fausses :

```
i = x;  
x = y;
```

Un **float** n'est pas un **int** et un **double** n'est pas un **float**.

Pour les objets, si **B** est sous-classe de **A**, alors on peut écrire :

```
A a = new B(...); //a est de type « A », mais l'objet référencé par a est de type « B ».  
A a1;  
B b = new B();  
a1 = b; // a1 de type « A », référence un objet de type « B »
```

Par contre, on ne peut pas avoir :

```
A a=new A();  
B b;  
b=a; // erreur: on ne peut pas convertir du type A vers le type B
```

instanceof

Si « B » est une sous classe de « A » alors l'instruction :

b instanceof A ; retourne true.

```

Etudiant etud = new Etudiant();
boolean b = etud instanceof Personne;
System.out.println("Resultat : "+b); //Resultat : true

EtudiantEtranger etudEtranger =new EtudiantEtranger();
b = etudEtrange instanceof Personne;
System.out.println("Resultat : "+b); //Resultat : true

Personne personne = new Etudiant();
b = personne instanceof Etudiant;
System.out.println("Resultat : "+b); //Resultat : true

personne = new Personne();
b = personne instanceof Etudiant;
System.out.println("Resultat : "+b); //Resultat : false

```

L'instruction « `personne instanceof Object;` » retourne « `true` » car toutes les classes héritent, par défaut, de la classe **Object**.

4.8 Héritage et méthodes finales

Pour éviter la redéfinition d'une méthode lors de l'héritage, on peut utiliser le mot clé `final`. Les méthodes déclarées comme `final` ne peuvent être redéfinies.

Exemple :

```

class A {
    final void meth() {
        System.out.println("Methode finale.");
    }
}

class B extends A {
    void meth() { // Erreur : meth() ne peut pas etre redefinie
        System.out.println("Illegal!");
    }
}

```

Classes finales et héritage

Pour éviter qu'une classe soit héritée, il faut la déclarer comme **finale** en la faisant précédée par le mot clé `final`. Implicitement, toutes les méthodes d'une classe finale sont elles aussi finales.

Exemple :

```
final class A {  
    // ...  
}  
  
class B extends A { // Erreur : B ne peut pas heriter de A  
    // ...  
}
```

4.9 Polymorphisme et héritage

4.9.1 Introduction

Le mot **polymorphisme** vient du grecque : poly (pour plusieurs) et morph (pour forme). Il veut dire qu'une même chose peut avoir différentes formes. Nous avons déjà vu cette notion avec la redéfinition des méthodes dans le chapitre 4 section 4.6. Une même méthode peut avoir différentes définitions suivant la classe où elle se trouve.

4.9.2 Liaison dynamique

Considérons la classe **B** qui hérite de la classe **A** :

```
class A{  
    public void message(){  
        System.out.println("Je suis dans la classe A");  
    }  
    public void g(){  
        System.out.println("Methode g() de la classe A");  
    }  
}  
  
class B extends A{  
    public void message(){  
        System.out.println("Je suis dans la classe B");  
    }  
    public void f(){  
        System.out.println("Methode f() de la classe B");  
    }  
}
```

La méthode « message() » a été redéfinie dans la classe **B** et la méthode « f() » a été ajoutée dans B.

Considérons les instructions :

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



```

A a = new A();
B b = new B();
a.message();
b.message();
b.f();
b.g();
a = new B();
a.message();
}

```

Dans l'exécution on aura le résultat suivant :

```

Je suis dans la classe A
Je suis dans la classe B
Methode f() de la classe B
Methode g() de la classe A
Je suis dans la classe B

```

Lorsqu'une méthode est redéfinie (s'est spécialisée), c'est la version la plus spécialisée qui est appelée. La recherche de la méthode se fait dans la classe réelle de l'objet. La recherche s'est fait lors de l'exécution et non lors de la compilation. Ce processus s'appelle la **liaison dynamique**.

Il s'appelle aussi : **liaison tardive**, **dynamic binding**, **late-binding** ou **run-time binding**.

Remarques :

- Dans les instructions précédentes, la dernière instruction « `a.message()`; » fait appel à la méthode « `message()` » de la classe **B**.
Si on ajoute l'instruction :
`a.f()`;
après les instructions :
`a = new B();`
`a.message();`
on aura une erreur de compilation, du fait que la méthode « `f()` » n'est pas implémentée dans la classe de déclaration de l'objet « `a` » même si la classe réelle (la classe B) possède « `f()` ».
- La visibilité d'une méthode spécialisée peut être augmentée (par exemple de `protected` vers `public`) mais elle ne peut pas être réduite (par exemple de `public` vers `private`)

Mécanisme de la liaison dynamique : cas des méthodes redéfinies

Soit D la classe réelle d'un objet d et soit A la classe de déclaration de l'objet d ($A \ d = \text{new } D()$). Si on a l'instruction : $d.f()$; quelle méthode $f()$ sera appelée ?

1. Si la méthode $f()$ n'est pas définie dans une classe ancêtre de la classe de déclaration (classe A) alors erreur de compilation.
2. Si non
 - Si la méthode $f()$ est redéfinie dans la classe D , alors c'est cette méthode qui sera exécutée

- Sinon, la recherche de la méthode `f()` se poursuit dans la classe mère de `D`, puis dans la classe mère de cette classe mère, et ainsi de suite, jusqu'à trouver la définition d'une méthode `f()` qui sera alors exécutée.

4.9.3 Polymorphisme et méthodes de classes

Il n'y a pas de polymorphisme avec les méthodes de classes.

Exemple

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();

        e = new EtudiantEtranger();
        e.message();
    }
}

class Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public void message(){
        System.out.println("Je suis un etudiant etranger");
    }
}
```

L'exécution du programme précédent donnera :

```
Je suis un etudiant
Je suis un etudiant etranger
```

Si on modifie la méthode « `message()` » de la classe `Etudiant`, en la rendant statique :

```
public class MethodesInstances {
    public static void main(String[] args){
        Etudiant e = new Etudiant();
        e.message();

        e = new EtudiantEtranger();
        e.message();
    }
}
```

```

}

class Etudiant {
    public static void message(){
        System.out.println("Je suis un etudiant");
    }
}

class EtudiantEtranger extends Etudiant {
    public static void message(){
        System.out.println("Je suis un etudiant etranger");
    }
}

```

alors l'exécution du programme précédent donnera :

```

Je suis un etudiant
Je suis un etudiant

```

C'est la classe du type qui est utilisée (ici Etudiant) et non du type réel de l'objet (ici EtudiantEtranger).

4.9.4 Utilité du polymorphisme

Le polymorphisme permet d'éviter les codes qui contiennent plusieurs embranchements et tests.

Exemple

Considérons deux classes **Rectangle** et **Cercle** qui héritent d'une super-classe **FormeGeometrique**. Dans un tableau hétérogène (on verra les tableaux dans le chapitre suivant), on range des objets de type **Rectangle** et **Cercle**.

```

class FormeGeometrique {
    public void dessineRectangle() {} // methode vide
    public void dessineCercle() {} // methode vide
}

class Rectangle extends FormeGeometrique {
    public void dessineRectangle() {
        System.out.println("Je suis un rectangle ");
    }
}

class Cercle extends FormeGeometrique {
    public void dessineCercle() {
        System.out.println("Je suis un cercle ");
    }
}

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

```

```

FormeGeometrique [] figures = new FormeGeometrique[3];
figures[0]=new Rectangle();
figures[1]=new Cercle();
figures[2]=new Cercle();
for (int i=0; i < figures.length; i++) {
    if (figures[i] instanceof Rectangle)
        figures[i].dessineRectangle();
    else if (figures[i] instanceof Cercle)
        figures[i].dessineCercle();
}
}

```

Inconvénients du code précédent

Le code précédent a au moins deux inconvénients :

- Si on veut ajouter une nouvelle forme géométrique, un triangle par exemple, alors on est obligé de changer le code source de la super-classe. En effet, on doit ajouter la méthode vide public void **dessineTriangle()** ;
- Aussi on doit ajouter un branchement dans la classe **Test** pour tester si *figures[i]* *instanceof Triangle*. Par suite si on traite plusieurs formes géométrique, alors le code comportera plusieurs tests.

En exploitant le **polymorphisme**, on peut améliorer le code précédent comme suit.

```

class FormeGeometrique {
    public void dessineFigure() {} // methode vide
}
class Rectangle extends FormeGeometrique {
    public void dessineFigure() {
        System.out.println("Je suis un rectangle ");
    }
}
class Cercle extends FormeGeometrique {
    public void dessineFigure() {
        System.out.println("Je suis un cercle .... ");
    }
}
public class Test {
    public static void main(String[] args) {
        FormeGeometrique [] figures = new FormeGeometrique[3];
        figures[0] = new Rectangle();
        figures[1] = new Cercle();
        figures[2] = new Cercle();
        for (int i=0; i < figures.length; i++)
            figures[i].dessineFigure(); //pas besoin de faire de tests
    }
}

```

Avec ce code on aura des avantages parmi lesquels :

- on évite trop de tests ;
- le code devient ainsi extensible. En effet, on peut ajouter de nouvelles sous-classes sans toucher au code existant.

4.10 Exercices

Exercice 1

Créez une classe **Cheval** qui contient trois attributs privés **nom**, **couleur** et **anneeNaissance**. Ajoutez des getters et des setters pour les différents attributs.

Créez une sous-classe nommée **ChevalCourse** qui contient un attribut supplémentaire **nombreCourses** qui contient le nombre de courses qu'un cheval a terminé. Ajoutez des méthodes **get** et **set** pour le nouveau attribut.

Testez les deux classes dans une nouvelle classe **TestChevaux**.

Exercice 2

1. Créez la classe **Cercle** qui contient les attributs privés **rayon**, **diametre** et **surface**. La classe contient :
 - un constructeur qui initialise le rayon et calcule les autres attributs ;
 - les méthodes **setRayon** et **getRayon**. La méthode **setRayon** calcule aussi les valeurs de **diametre** et **surface**.
2. Créez une classe **Disque** qui hérite de **Cercle** et qui contient :
 - les attributs privés **hauteur** et **volume** ;
 - les méthodes **setHauteur** et **getHauteur** ;
 - un constructeur pour initialiser les différents attributs ;
 - redéfinie la méthode **setRayon** ;
 - les méthodes **getVolume()** et **setVolume()**.
3. Testez les deux classes dans la classe **TestCercleDisque**.

Chapitre 5

Tableaux et collections

5.1 Tableaux

5.1.1 Déclaration et initialisation

Comme pour les autres langages de programmation, Java permet l'utilisation des tableaux.

La déclaration d'un tableau à une dimension se fait de deux façons équivalentes :

```
type tab[]; ou type[] tab;
```

`tab` est une référence à un tableau.

Exemple :

```
int tab[ ];
```

```
int[ ] tab;
```

Contrairement au langage C, la déclaration `int tab[10];` n'est pas permise. On ne peut pas fixer la taille lors de la déclaration.

Remarques :

En Java, un tableau :

- est un objet ;
- est alloué dynamiquement (avec l'opérateur **new**) ;
- a un nombre fixe d'éléments de même type ;

La création d'un tableau peut se faire soit, lors de la déclaration soit par utilisation de l'opérateur **new**.

Exemples :

1. Création par initialisation au début :

```
int [] tab={12,10,30*4};
```

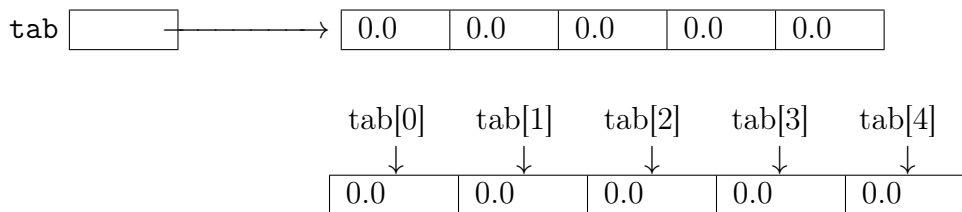
2. Création par utilisation de **new** :

```
int [] tab = new int[5];
```

La déclaration

```
double[] tab = new double[5];
```

Crée un emplacement pour un tableau de 5 réels (double) et fait la référence à **tab** comme illustré par le schéma suivant :



Les valeurs du tableau sont initialisées aux valeurs par défaut (0 pour int, 0.0 pour double ...).

Déclaration mixte

On peut combiner la déclaration de tableaux avec d'autres variables de même type. La déclaration :

```
double scores [], moyenne;
```

crée :

- un tableau non initialisé de type **double** nommé **scores** ;
- une variable de type **double** nommée **moyenne**.

Par contre, la déclaration :

```
double[] scores, moyenne;
```

crée deux tableaux non initialisés.

Taille d'un tableau

La taille d'un tableau est accessible par le champ « public final » **length**.

Exemple :

```
double[] scores = new double[10];
System.out.println(scores.length); //Affiche 10
```

Remarque :

La taille d'un tableau ne peut pas être changée. Par contre, la référence du tableau peut être changée. Elle peut référencer un tableau de taille différente.

Exemple :

```
double[] tab1 = new double[10];
double[] tab2 = new double[5];
tab1 = new double[7]; //tab1 reference maintenant un nouveau tableau de taille 7
```

```
tab2 = tab1; //tab1 et tab2 referencent le meme tableau
```

5.1.2 Parcourir un tableau

Comme pour le langage C, on peut accéder aux éléments d'un tableau en utilisant l'indice entre crochets ([]).

Exemple

```
double [] tab = new double [10];
int i;
for (i=0; i<tab.length; i++){
    tab[i]=i*i;
    System.out.println("tab["+i+"] = "+tab[i]);
}
```

Remarque :

La lecture d'un tableau peut se faire de la façon suivante :

```
for (double t: tab)
    System.out.println(t);
```

Cette méthode de parcours d'un tableau n'est valable que pour la lecture et ne peut pas être utilisée pour la modification. Elle a l'avantage d'éviter l'utilisation des indices.

Dans le listing suivant, pour calculer la somme des éléments du tableau, on n'a pas besoin de connaître les indices.

```
double somme = 0.0;
for (double t : tab)
    somme += t;

System.out.println("Somme =" +somme);
```

Parcourir les arguments de la ligne de commande

La méthode principale « main() » contient un tableau de « String ». Ce tableau peut être utilisé comme les autres tableaux. Pour afficher son contenu, on peut utiliser le code suivant :

```
System.out.println("taille de args "+args.length);
for (String arg : args)
    System.out.print(arg.toUpperCase()+" ");
```

Si le programme s'appelle « Test » et exécute la commande « java Test bonjour tout le monde », alors le code précédent affichera :

taille de args : 4
BONJOUR TOUT LE MONDE

5.1.3 Copie de tableaux

Comme on l'a vu précédemment, pour deux tableaux référencés par « tab1 » et « tab2 », l'instruction :

```
tab1 = tab2;
```

ne copie que la référence et non le contenu. Maintenant *tab1* et *tab2* désignent le même objet tableau qui était initialement référencé par *tab2*.

Exemple :

```
public class EgalTableaux {
    public static void main(String[] argv) {
        int [] tab1 = {1,2,3,4,5} ;
        int [] tab2 = {0,2,4,6,8} ;
        int [] tab3 = {0,2,4,6,8} ;
        tab1 = tab2;

        System.out.println(tab1.equals(tab2));
        System.out.println(tab3.equals(tab2));
        tab2[0] = 3;
        System.out.println("tab1[0]= "+tab1[0]);
        System.out.println("tab2[0]= "+tab2[0]);
        System.out.println("tab3[0]= "+tab3[0]);
    }
}
```

L'exécution de ce programme affiche :

```
true
false
tab1[0]= 3
tab2[0]= 3
tab3[0]= 0
```

Pour copier le contenu d'un tableau dans un autre, on peut soit :

- procéder de façon classique et copier élément par élément ;

```
for (int i=0;i<tab1.length;i++)
    tab2[i] = tab1[i];
```

- soit utiliser des méthodes prédéfinies dans la classe `Arrays` telles que : `System.arraycopy()`, `Arrays.copyOf()` ou `Arrays.copyOfRange()`.
On ne verra pas ces méthodes pour ce cours.

5.1.4 Passage et retour d'un tableau dans une méthode

Considérons le programme suivant :

```
public class TabMethodes {
    public static void main(String[] args) {
        int[] tab = {1, 2, 3, 4};
        int i;

        System.out.print("Debut de main: ");
        afficher(tab);
        //Appel de test (pour tous les elements du tableau)
        for (i = 0; i < tab.length; i++){
            test(tab[i]);
        }
        System.out.print("Fin de main: ");
        afficher(tab);
    }

    public static void test(int x) {
        System.out.print("Debut : \t" + x);
        x = 10;
        System.out.println("\tfin : \t" + x);
    }

    public static void afficher(int[] tableau) {
        int i;
        for (i = 0; i < tableau.length; i++){
            System.out.print(tableau[i] + "-");
        }
        System.out.println();
    }
}
```

L'exécution du programme précédent donne le résultat suivant :

```
Debut de main: 1-2-3-4-
Debut : 1 fin : 10
Debut : 2 fin : 10
Debut : 3 fin : 10
Debut : 4 fin : 10
Fin de main: 1-2-3-4-
```

D'après le résultat de l'exécution, le contenu du tableau n'a pas été changé du fait que « x » est locale à la méthode « test() » et tout changement de cette valeur ne sera pas visible à l'extérieur, et par conséquent le tableau reste inchangé.

Maintenant, on passe le tableau comme argument à la méthode test().

```

public class TabMethodesArg {
    public static void main(String[] args) {
        int[] tab = {1, 2, 3, 4};
        int i;
        System.out.print("Debut de main: ");
        afficher(tab);
        test(tab);
        System.out.print("Fin de main: ");
        afficher(tab);
    }

    public static void test(int [] tableau) {
        int i;
        for (i = 0; i < tableau.length; i++)
            tableau[i] = 10;
    }
    public static void afficher(int [] tableau) {
        int i;
        for (i = 0; i < tableau.length; i++){
            System.out.print(tableau[i]+"-");
        }
        System.out.println();
    }
}

```

on voit bien que toute modification affectera le tableau, en effet le résultat est :

```

Debut de main: 1-2-3-4-
Fin de main: 10-10-10-10-

```

Retour d'un tableau dans une méthode

Une méthode peut retourner un tableau.

Exemple

La méthode :

```

public static int [] scoreInitial() {
    int score[] = {2, 3, 6, 7, 8};
    return score;
}

```

peut être utilisé comme suit :

```

public static void main(String[] args) {
    int[] tab = scoreInitial();
}

```

```
} // on peut afficher le contenu du tableau tab;
```

5.1.5 Tableaux d'objets

L'utilisation des tableaux n'est pas limité aux types primitifs. On peut créer des tableaux d'objets. On a déjà utilisé les tableaux d'objets dans la méthode « `main(String[] args)` », puisque « `String` » est une classe.

Considérons la classe « `Etudiant` » vue dans les chapitres précédents :

```
class Etudiant {
    private String nom, prenom, cne;
    public Etudiant() {
        nom = " ";
        prenom = " ";
        cne = " ";
    }
    ...
}
```

La déclaration :

```
Etudiant[] etudiants = new Etudiant[30];
```

Crée l'emplacement pour contenir 30 objets de type « `Etudiant` ». Elle ne crée que les références vers les objets. L'instruction suivante

```
etudiants[0].afficher();
```

provoquera une erreur à l'exécution. Il faut d'abord créer l'objet `etudiants[0]` comme suit par exemple :

```
etudiants[0] = new Etudiant();
```

Pour créer tous les objets, il faut utiliser, par exemple, l'instruction suivante :

```
for(int i=0; i<etudiants.length; i++)
    etudiants[i]=new Etudiant();
```

5.2 Tableaux à plusieurs dimensions

On peut créer des tableaux à plusieurs dimensions par ajout de crochets (`[]`). Par exemple, l'instruction :

```
double[][] matrice;
```

déclare un tableau à 2 dimensions de type **double**.

Comme pour le tableau à une seule dimension, la création d'un tableau multi-dimensionnelle peut se faire par utilisation de l'opérateur **new**.

Exemple

L'instruction :

```
matrice = new double[4][3];
```

crée un tableau de 4 lignes et 3 colonnes.

On peut combiner les deux instructions précédentes :

```
double [][] matrice = new double[4][3];
```

Remarques

- En langage C, un tableau à plusieurs dimensions est en réalité un tableau à une dimension. Par exemple, la déclaration :
`double matrice [4][3];`
crée en mémoire un tableau (contiguë) de 12 double.
- En Java, un tableau à plusieurs dimensions n'est pas contiguë en mémoire. En Java, un tableau de plusieurs dimensions est un tableau de tableaux.
- On peut définir un tableau à 2 dimensions dont les colonnes n'ont pas la même dimension.

Exemple 1

```
double [][] tab = new double [2] [];  
tab[0] = new double [3]; //tab[0] est un tableau de 3 doubles  
tab[1] = new double [4]; //tab[1] est un tableau de 4 doubles  
  
System.out.println(tab.length); //Affiche 2  
System.out.println(tab[0].length); //Affiche 3  
System.out.println(tab[1].length); //Affiche 4  
  
for (int i=0; i<tab.length; i++){  
    for (int j=0; j<tab[i].length; j++){  
        tab[i][j]=i+j;  
        System.out.print ("tab["+i+"]["+j+"] = " + tab[i][j] + " ");  
    }  
}  
//Affiche tab[0][0] = 0.0 tab[0][1] = 1.0 tab[0][2] = 2.0  
//tab[1][0] = 1.0 tab[1][1] = 2.0 tab[1][2] = 3.0 tab[1][3] = 4.0
```

L'exemple précédent crée un tableau à 2 dimensions dont la première ligne est composée de 3 éléments et la deuxième ligne est composée de 4 éléments.

Exemple 2

Dans l'exemple suivant, on va créer un tableau triangulaire qui sera initialisé comme suit :

```
0  
0 0
```

```
0 0 0
0 0 0 0
```

```
final int N = 4;
int [][] tabTriangulaire = new int [N] [];
for (int n=0; n<N; n++)
    tabTriangulaire [n]= new int [n+1];

for (int i=0; i<tabTriangulaire.length; i++){
    for (int j=0; j<tabTriangulaire [ i ].length; j++)
        System.out.print ( tabTriangulaire [ i ][ j]+ "\t" );
    System.out.println ();
}
/* Affiche
0
0 0
0 0 0
0 0 0 0
*/
```

5.2.1 Parcourir un tableau multi-dimensionnel

Comme pour le cas à une seule dimension, on peut utiliser la boucle (pour chaque -for each) pour accéder au contenu d'un tableau multi-dimensionnel.

Exemple

```
double [][] matrice = new double [4][3];

for (int i=0; i<4; i++)
    for (int j=0; j<3; j++)
        matrice [ i ][ j]= i+j;

double somme=0;
for (double [] ligne : matrice)
    for (double val: ligne)
        somme += val;

System.out.println ( "somme = "+somme );
```

5.2.2 Initialisation au début

L'initialisation peut se faire comme suit :

```
double [][] matrice = {new double[4],new double[5]};
```

Elle peut se faire au début en affectant des valeurs au tableau :

```
double [][] tabMultiBis = {{1,2},{3,5},{3,7,8,9,10}};
```

Exemple

```
double [][] tabMulti = {new double [4],new double [5]};

System.out.println(tabMulti.length); //Affiche 2
System.out.println(tabMulti[0].length); //Affiche 4
System.out.println(tabMulti[1].length); //Affiche 5

double [][] tabMultiBis = {{1,2},{3,5},{3,7,8,9,10}};

System.out.println(tabMultiBis.length); //Affiche 3
System.out.println(tabMultiBis[0].length); //Affiche 2
System.out.println(tabMultiBis[1].length); //Affiche 2
System.out.print(tabMultiBis[2].length); //Affiche 5
```

5.3 La collection ArrayList

Introduction

Les tableaux ne peuvent pas répondre à tous les besoins de stockage d'objets (par exemple quand un nombre inconnu d'objets sont à stocker). La large diversité d'implémentations proposées par l'API Collections de Java permet de répondre à la plupart des besoins.

Une collection est un objet qui contient d'autres objets. Dans cette section, on verra la classe `ArrayList` qui se trouve dans le package `java.util`. Un `ArrayList` se comporte comme un tableau dynamique, il contient plusieurs méthodes dont les plus utilisées sont :

- **add(Object element)** permet d'ajouter l'objet `element` ;
- **get(int index)** retourne l'élément à l'indice demandé ;
- **remove(int index)** efface l'entrée à l'indice demandé ;
- **isEmpty()** renvoie `vrai` si l'objet est vide ;
- **removeAll()** efface tout le contenu de l'objet ;
- **contains(Object element)** retourne `vrai` si l'élément `element` est dans l'`ArrayList` ;
- **size()** retourne la taille de `ArrayList`.

Exemples

Exemple 1

```
import java.util.*;

public class Test {
```

```

public static void main(String[] args) {

    ArrayList Ar = new ArrayList();

    Ar.add(12);
    Ar.add("Bonjour tout le monde !");
    Ar.add(12.2);
    Ar.add('d');

    for(int i = 0; i < Ar.size(); i++){
        System.out.println("indice " + i + " : " + Ar.get(i));
    }

    Ar.remove(2);
    System.out.println("Après ");
    for(int i = 0; i < Ar.size(); i++){
        System.out.println("indice " + i + " : " + Ar.get(i));
    }

    System.out.println(Ar.contains('d'));
}
}

```

L'exécution de ce programme affiche :

```

indice 0 : 12
indice 1 : Bonjour tout le monde !
indice 2 : 12.2
indice 3 : d
Après
indice 0 : 12
indice 1 : Bonjour tout le monde !
indice 2 : d
true

```

Exemple 2

```

import java.util.*;
public class TabDynamique1 {
    public static void main(String[] args) {
        ArrayList<Etudiant> Ar = new ArrayList<Etudiant>();
        Etudiant et1 = new Etudiant("Ali", "Ahmed", "A7788");
        Etudiant et2 = new Etudiant("Salim", "Mohammed", "A8899");
        Ar.add(et1);
        Ar.add(et2);
        int i;
        for(i = 0; i < Ar.size(); i++){
            Ar.get(i).afficher();
        }
    }
}

```



```

        }
        Ar.remove(1);
        System.out.println("Après");
        for(i = 0; i < Ar.size(); i++){
            Ar.get(i).afficher();
        }
        System.out.println(Ar.contains('d'));
    }
}
class Etudiant {
    private String nom, prenom, cne;

    public Etudiant(String nom, String prenom, String cne) {
        this.nom = nom;
        this.prenom = prenom;
        this.cne = cne;
    }
    public void afficher() {
        System.out.println("Nom : "+nom + ", prenom : " +prenom +
            " et CNE : "+cne);
    }
}

```

L'exécution de ce programme affiche :

```

Nom : Ali, prenom : Ahmed et CNE : A7788
Nom : Salim, prenom : Mohammed et CNE : A8899
Après
Nom : Ali, prenom : Ahmed et CNE : A7788
false

```

Remarque

Dans l'exemple précédent, pour qu'on puisse utiliser la méthode afficher() de la classe Etudiant, on a caster l'arraylist : `ArrayList<Etudiant> Ar = new ArrayList<Etudiant>();`

5.4 Exercices

Exercice 1

Écrivez un programme qui contient un tableau de 10 entiers. Le programme affiche :

1. tous les entiers ;
2. tous les entiers dans le sens inverse ;
3. la somme des dix entiers ;
4. le minimum des entiers ;

5. le maximum des entiers ;
6. la moyenne des entiers ;
7. toutes les valeurs supérieures à la moyenne.

Exercice 2

1. Créez une classe **Personne** qui contient :
 - les attributs **privés** : **nom**, **prenom** de type **String** et **tel** de type **int** ;
 - un constructeur pour initialiser les différents attributs ;
 - une méthode **afficher()** qui affiche le nom, le prénom et le numéro de téléphone.
2. Écrivez une classe **CarnetTel** qui contient une méthode **main()**. Dans la méthode **main()** :
 - a - déclarez un tableau de 10 personnes ;
 - b - l'utilisateur doit saisir les informations concernant les dix personnes ;
 - c - affichez les informations concernant les différentes personnes ;
 - d - demandez à l'utilisateur de saisir un nom et affichez les informations concernant le nom saisi.
3. Reprendre la question 2, mais cette fois ci :
 - a - déclarez un `ArrayList` et y ajouter cinq personnes ;
 - b - supprimer la troisième personne.
 - c - affichez les informations concernant les différentes personnes avant et après suppression ;
 - d - demandez à l'utilisateur de saisir un nom et affichez les informations concernant le nom saisi s'il existe.

Exercice 3

Supposons vous avez une classe nommé **Etudiant** qui contient une méthode **setCNE** qui prend comme argument un **int** et vous avez dans une méthode **main** déclaré un tableau **tabEtud** de 20 étudiants. Parmi les déclarations suivantes, laquelle assigne un CNE au premier étudiant ?

- | | |
|--------------------------------|--------------------------------|
| a – Etudiant[0].setCNE(1234) ; | c – Etudiant.setCNE[0](1234) ; |
| b – tabEtud[0].setCNE(1234) ; | d – tabEtud.setCNE[0](1234) ; |

Chapitre 6

Classes abstraites, interfaces et packages

6.1 Classes abstraites

6.1.1 Méthodes abstraites

Une méthode est dite **abstraite**, lorsqu'on la déclare sans donner son implémentation : on ne fournit que le type de la valeur de retour et la signature (l'entête de la méthode).

Les méthodes abstraites sont déclarées avec le mot clé **abstract** et doivent obligatoirement être déclarées **public**.

Exemple :

```
public abstract void f(int i, float x);  
public abstract double g();
```

Une méthode abstraite d'une classe A peut être implémentée par les classes filles de A.

Une méthode static ne peut être abstraite (car on ne peut redéfinir une méthode static).

6.1.2 Classes abstraites

Une classe abstraite est une classe qu'on ne peut pas l'instancier directement car certaines de ses méthodes ne sont pas implémentées (abstraites).

Une classe abstraite peut contenir des variables, des méthodes implémentées et des méthodes abstraites.

Une classe abstraite est déclarée avec le mot clé **abstract class**.

Une classe abstraite est **non-instanciable**. En effet, si A est une classe abstraite, alors on peut créer une **référence** sur un objet de type A, mais il est interdit de créer une **instance** (un objet) de A.

```
A a;    // possible  
A a = new A(); // erreur.
```

Si B hérite de A et B est non abstraite alors on peut écrire : A a = `new` B();

Une sous-classe d'une classe abstraite doit implémenter toutes les méthodes abstraites, sinon elle doit être déclarée abstraite.

6.1.3 Utilité et exemple

L'intérêt des classes abstraites est le suivant :

la classe mère définit la structure globale d'un algorithme. Elle laisse aux classes filles le soin de définir des points bien précis de l'algorithme.

Exemple

Considérons les classes **Cercle** et **Rectangle** qui sont des sous classes de la classe **FigureGeometrique**. Les surfaces d'un cercle et d'un rectangle ne sont pas calculées de la même façon. Une solution consiste à définir dans la classe **FigureGeometrique** une méthode abstraite « `surface()` » et obliger les classes **Cercle** et **Rectangle** à implémenter cette méthode.

```
abstract class FigureGeometrique {
    public abstract double surface();
}

class Cercle extends FigureGeometrique {
    private double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    public double surface() {
        return Math.PI * rayon * rayon;
    }
}

class Rectangle extends FigureGeometrique {
    public double largeur, longueur;

    public Rectangle(double largeur, double longueur) {
        this.largeur = largeur;
        this.longueur = longueur;
    }

    public double surface() {
        return largeur * longueur;
    }
}
```

6.2 Interfaces

6.2.1 Définition et déclaration

Le langage c++ permet l'héritage multiple, par contre Java ne permet pas l'héritage multiple. Pour remédier à ceci, Java utilise une alternative qui est la notion **d'interfaces**.

Définition : une **interface** est un ensemble de méthodes abstraites.

Déclaration

La déclaration d'une interface se fait comme celle d'une classe sauf qu'il faut remplacer le mot clé `class` par `interface`.

Exemple

```
public interface Forme {  
    public abstract double perimetre (); //public abstract  
    public abstract double surface (); // est facultatif  
}
```

Les interfaces ont les propriétés suivantes :

- une interface est implicitement abstraite. On n'a pas besoin d'utiliser le mot clé `abstract` dans sa déclaration ;
- chaque méthode définie dans une interface est abstraite et public, par conséquent, les mots clés `public` et `abstract` peuvent être omis.

Règles

Une interface est similaire à une classe dans les points suivants :

- une interface peut contenir plusieurs méthodes (abstraites) ;
- une interface peut se trouver dans un fichier séparé (.java) ;
- peut se trouver dans un package (voir section 6.3).

Cependant, une interface :

- ne peut pas instancier un objet et par conséquent ne peut pas contenir de constructeurs ;
- ne peut pas contenir d'attributs d'instances. Tous les attributs doivent être `static` et `final` ;
- peut hériter de plusieurs interfaces (héritage multiple autorisé pour les interfaces).

Remarque : dans Java 8, une interface peut contenir des méthodes statiques.

6.2.2 Implémentation d'une interface

Une classe peut implémenter une interface en utilisant le mot clé `implements`. On parle d'implémentation et non d'héritage.

Exemple

```
class Rectangle implements Forme {
    private double largeur , longueur;

    public Rectangle(double largeur , double longueur) {
        this.largeur = largeur;
        this.longueur = longueur;
    }

    public double perimetre() {
        return 2 * (largeur + longueur);
    }

    public double surface() {
        return largeur * longueur;
    }
}
```

Implémentation partielle

Une classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être abstraite.

Exemple

```
abstract class Rectangle implements Forme {
    private double largeur , longueur;

    public double surface() {
        return largeur * longueur;
    }
}
```

Implémentation multiple

Une classe peut implémenter plusieurs interfaces.

Exemple

Considérons les 2 interfaces **I1** et **I2** :

```
interface I1 {
    final static int MAX = 20;
    void meth1();
    void meth2();
}
```

```
interface I2 {
    void meth3();
    void meth4();
}
```

La classe **A** peut implémenter les 2 interfaces **I1** et **I2** :

```
class A implements I1 , I2 {
    // attributs
    public void meth1() {
        int i=10;
        //on peut utiliser la constante MAX
        if (i < MAX)
            i++;
        //implementation de la methode
    }
    public void meth2() {
        // ...
    }
    public void meth3() {
        // ...
    }
    public void meth4() {
        // ...
    }
}
```

Implémentation et héritage

Une classe **B** peut hériter de la classe **A** et implémenter les 2 interfaces **I1** et **I2**, comme suit :

```
class B extends A implements I1,I2
```

6.2.3 Polymorphisme et interfaces

Déclaration

On peut déclarer des variables de type interface, par exemple **Forme forme**;

Pour instancier la variable **forme**, il faut utiliser une classe qui implémente l'interface **Forme**.
Par exemple :

```
forme = new Rectangle(2.5, 4.6);
```

Tableaux

Considérons la classe **Cercle** qui implémente elle aussi l'interface **Forme** et la classe **Carre** qui hérite de **Rectangle**.

```

class Cercle implements Forme {
    private double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    public double perimetre() {
        return 2 * Math.PI * rayon;
    }

    public double surface() {
        return Math.PI * rayon * rayon;
    }
}

class Carre extends Rectangle{
    private double largeur;

    public Carre(double largeur){
        super(largeur, largeur);
    }
}

```

On pourra écrire :

```

public static void main(String[] args) {
    Forme [] tabForme = new Forme[3];
    tabForme[0]=new Rectangle(10, 20);
    tabForme[1]=new Cercle(3);
    tabForme[2]=new Carre(10);

    for(int i=0; i<3; i++)
        System.out.println(tabForme[i].surface() +", " +
            tabForme[i].perimetre());
}

```

Casting

Ajoutons à la classe **Cercle** la méthode « diametre() » :

```

class Cercle implements Forme {
    private double rayon;
    ...
    public double diametre() {
        return 2 * rayon;
    }
}

```


Considérons l'instruction :

```
Forme forme = new Cercle(5);
```

L'instruction :

```
double d = forme.diametre();
```

génère une erreur de compilation. Pour éviter cette erreur, il faut faire un cast comme suit :

```
double d = ((Cercle) forme).diametre();
```

6.2.4 Héritage et interfaces

Une interface peut hériter d'une ou plusieurs interfaces.

Exemple

```
interface I1 {  
    final static int MAX = 20;  
    void meth1();  
    void meth2();  
}
```

```
interface I2 {  
    void meth3();  
    void meth4();  
}
```

```
interface I3 extends I1, I2 {  
    void meth5();  
}
```

L'instruction `interface I3 extends I1, I2`, est normalement équivalente à :

```
interface I3 {  
    final static int MAX = 20;  
    void meth1();  
    void meth2();  
    void meth3();  
    void meth4();  
    void meth5();  
}
```

6.3 Packages

Un package est un ensemble de classes. Il sert à mieux organiser les programmes. Si on n'utilise pas de packages dans nos programmes, alors on travaille automatiquement dans le package par défaut (default package).

Par exemple, pour la saisie à partir du clavier, nous avons utilisé la classe **Scanner**, pour cela nous avons importé le package **java.util**.

6.3.1 Création d'un package

Pour créer un package, on utilise le mot clé **package**. Il faut ajouter, au début de chaque fichier, l'instruction :

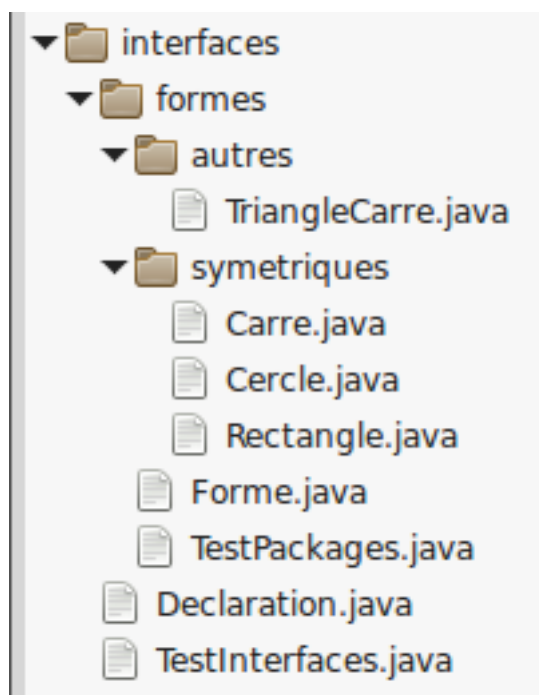
package nomPackage;

Pour qu'elle soit utilisable dans un package, une classe publique doit être déclarée dans un fichier séparé.

On accède à une classe publique en utilisant le nom du package.

6.3.2 Exemple

Considérons la hiérarchie suivante :



Nous avons la structure suivante :

- le répertoire **interfaces** contient le répertoire **formes** et les fichiers **Declaration.java** et **TestInterfaces.java** ;
- le répertoire **formes** contient les répertoires **symetriques** et **autres**, et les fichiers **Forme.java** et **TestPackages.java** ;

- le répertoire **autres** contient le fichier **TriangleCarre.java** ;
- le répertoire **symetriques** contient les fichiers **Carre.java**, **Cercle.java** et **Rectangle.java**.

Puisque la classe **TriangleCarre** se trouve dans le répertoire **autres**, donc on doit inclure, au début l'instruction :

```
package interfaces . formes . autres ;
```

Et comme la classe **TriangleCarre** hérite de la classe **Rectangle**, donc, on doit inclure l'instruction :

```
import interfaces . formes . symetriques . Rectangle ;
```

La classe **TriangleCarre** aura la structure suivante :

```
package interfaces . formes . autres ;

import interfaces . formes . symetriques . Rectangle ;

public class TriangleCarre extends Rectangle {
    private double cote ;

    public TriangleCarre ( double larg , double longu , double cote ) {
        super ( larg , longu ) ;
        this . cote = cote ;
    }

    public double surface () {
        return super . surface () / 2 ;
    }
}
```

La classe **TestPackages** sert pour tester les différentes classes, donc on doit inclure, au début les instructions :

```
package interfaces . formes ;
```

```
import interfaces . formes . autres . TriangleCarre ;
```

```
import interfaces . formes . symetriques . Carre ;
```

```
import interfaces . formes . symetriques . Cercle ;
```

```
import interfaces . formes . symetriques . Rectangle ;
```

Pour simplifier, on peut remplacer les trois dernières instructions par :

```
import interfaces . formes . symetriques . * ;
```

La structure de la classe **TestPackages** est comme suit :

```
package interfaces . formes ;

import interfaces . formes . autres . TriangleCarre ;
import interfaces . formes . symetriques . * ;

public class TestPackages {

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

```

    Forme[] tabForme = new Forme[4];
    tabForme[0] = new Rectangle(10, 20);
    tabForme[1] = new Cercle(3);
    tabForme[2] = new Carre(10);
    tabForme[3] = new TriangleCarre(3,4,5);

    for (int i = 0; i < 3; i++)
        System.out.println(tabForme[i].surface()+" , "+
                           tabForme[i].perimetre());
    }
}

```

6.3.3 Classes du même package

Pour les classes qui sont dans le même package, on n'a pas besoin de faire des importations et on n'a pas besoin de mettre une classe par fichier si on veut que cette classe ne soit pas visible pour les autres packages.

Soit le fichier **Carre.java** qui contient les 2 classes **Carre** et **Cube** :

```

package interfaces.formes.symetriques;

public class Carre extends Rectangle {
    private double largeur;

    public Carre(double largeur) {
        super(largeur, largeur);
    }
    double getLargeur() {
        return largeur;
    }
}

class Cube extends Carre {
    public Cube(double largeur) {
        super(largeur);
    }
    public double volume() {
        return surface() * super.getLargeur();
    }
}

```

Remarques

1. Les classes **Rectangle** et **Carre** sont dans le même package, donc on n'a pas besoin de faire des importations.

2. La classe **Cube** est invisible dans les autres packages. Dans la classe **TestPackages**, une instruction comme :
`Cube cube = new Cube();`
génère une erreur de compilation, du fait que la classe **TestPackages** se trouve dans le package **interfaces.formes**

6.3.4 Fichiers jar

Un fichier **jar** (**J**ava **A**Rchive) est un fichier qui contient les différentes classes (compilées) sous format compressé.

Pour générer le fichier **jar** d'un projet sous eclipse, il faut cliquer avec la souris (bouton droit) sur le nom du projet puis cliquer sur **export**, choisir après **JAR** dans la section **java**, puis cliquer sur **next**, et choisir un nom pour le projet dans la partie **JAR file** (par exemple test.jar) puis cliquer deux fois **next** et choisir la classe principale. Cliquer, enfin, **Finish** pour terminer l'exportation.

En se positionnant dans le répertoire qui contient le fichier **test.jar**, ce dernier peut être exécuté en utilisant la commande :

```
java -jar test.jar
```

6.4 Exercices

Exercice 1

Corrigez le programme suivant et justifiez chaque correction :

```
interface I {  
    double m11();  
    int m2(){}  
}  
  
public class ClasseA {  
    public static void main(String[] args) {  
        I i1 = new I();  
        I i2;  
    }  
}
```

Exercice 2

Considérons les classes **ClasseA** et **ClasseB** définies comme suit :

```
package p1;

public class ClasseA {
    protected int a;
    private int b;
    public int c;
    final public int X = 7;
}
```

```
package p2;

public class ClasseB {
    public static void main(String[] args) {
        ClasseA objA = new ClasseA();

        objA.a = 20;           // q1
        ClasseA.a = 20;        // q2
        objA.b = 10;           // q3
        objA.c = 30;           // q4
        objA.X = 50;           // q5
        int D = objA.X;        // q6
    }
}
```

Dans **ClasseB**, citez les déclarations qui sont valides et celles qui ne sont pas valides ? Justifiez.

Exercice 3

- Créez le package **banque.personnes** et créez dans ce package une classe **Personne** qui contient :
 - les attributs **privés** : **nom**, **prenom** de type **String** et **age** de type **int** ;
 - un constructeur pour initialiser les différents attributs ;
 - redéfinissez la méthode **toString()** pour afficher le nom, le prénom et l'âge.
- Créez le package **banque.comptes** et créez dans ce package une classe **CompteBancaire** qui doit posséder :
 - les attributs privés **numCompte** de type **int** et **solde** de type **double** ;
 - un constructeur pour initialiser les différents attributs ;
 - la méthode **deposer(double s)** : pour ajouter une somme au solde ;
 - la méthode **retirer(double s)** : pour retirer une somme du solde ;
 - la méthode **info()** qui retourne les informations concernant le compte, comme suit :
« compte numéro : 121, Solde : 15000 Dhs ».
- Dans le package **banque.personnes**, créez une classe **Client** qui hérite de la classe **Personne** :
 - on suppose que chaque client dispose d'un seul compte ;
 - redéfinissez la méthode **toString()** pour qu'elle affiche aussi les informations concernant le compte.

Indication : ajoutez à la classe **Client** un attribut de type **CompteBancaire** qui sera initialisé dans un constructeur.
- Dans le package **banque**, écrivez une classe **Banque** qui contient une méthode **main()**. Dans la méthode **main()** :
 - déclarez un tableau de taille 4 de type **Personne**. Le tableau contiendra les données de 4 clients qui auront les caractéristiques suivantes :

	Nom	prénom	Age	Solde initial
Client 1	Oujdi	Ali	30	1000Dh
Client 2	Sami	Lina	27	2000Dh
Client 3	Berkani	Karim	35	3000Dh
Client 4	Othmani	Karima	40	4000Dh

Pour le numéro de compte, le premier client aura le numéro **1000** et pour chaque nouveau client il faut ajouter **1** au numéro du client précédent.

- Le client 1 dépose une somme d'argent saisie au clavier.
- La cliente 4 retire une somme d'argent saisie au clavier.
- Retirez 1000dh du client 3 et la déposer au cliente 4.
- Affichez les informations concernant les différents clients.

Chapitre 7

Gestion des exceptions

7.1 Introduction

Une **exception** est un signal indiquant que quelque chose d'exceptionnelle (comme une erreur) s'est produit. Elle interrompt le flot d'exécution normal du programme. L'objectif est de traiter les conditions **anormales** à part pour ne pas compliquer le code du traitement **normal**. Le traitement **normal** apparaît ainsi plus simple et plus lisible.

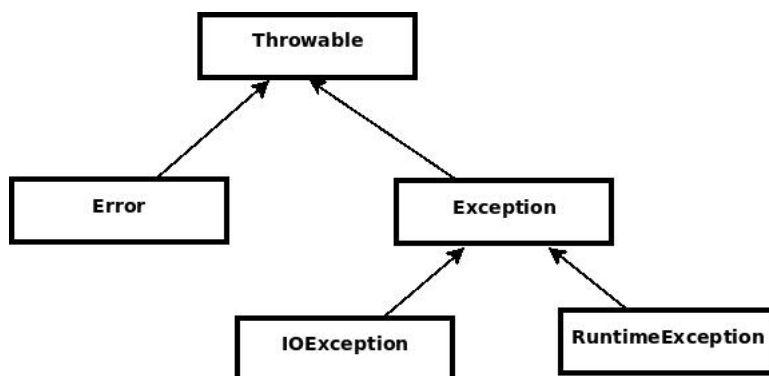
Voici quelques exemples des exceptions qu'on peut rencontrer :

- division par zéro ;
- une mauvaise valeur entrée par l'utilisateur (une chaîne de caractères au lieu d'un entier) ;
- dépassement des capacités d'un tableau ;
- lecture ou écriture dans un fichier qui n'existe pas, ou pour lequel le programme n'a pas les droits d'accès ;
- ...

7.2 Exceptions

En java, il y a deux classes pour gérer les erreurs : **Error** et **Exception**.

Les deux classes sont des sous-classe de la classe **Throwable** comme le montre la figure suivante :



- La classe **Error** représente les erreurs graves qu'on ne peut pas gérer. Par exemple, il

n'y a pas assez de mémoire pour exécuter un programme.

- La classe **Exception** représente les erreurs les moins graves qu'on peut gérer dans les programmes. La classe **Exception** a deux sous-classes qui sont : la classe **IOException** et la classe **RuntimeException**.

7.3 Types d'exceptions

Dans ce qui suit, nous donnons quelques types d'exceptions :

Exception	Description
ArithmeticException	Erreur arithmétique, comme une division par zéro.
ArrayIndexOutOfBoundsException	Indice d'un tableau qui dépasse les limites du tableau. Par exemple : <code>double [] tab = new double[10];</code> <code>tab[10]=1.0;</code> ou bien <code>tab[-1]=1.0;</code>
NegativeArraySizeException	Tableau créé avec une taille négative. Par exemple : <code>double [] tab = new double[-12];</code> .
ClassCastException	Cast invalide. Voir un exemple ci-dessous.
NumberFormatException	Mauvaise conversion d'une chaîne de caractères vers un type numérique. Par exemple : <code>String s = "tt";</code> <code>x = Double.parseDouble(s);</code>
StringIndexOutOfBoundsException	Indice qui dépasse les limites d'une chaîne de caractères. Par exemple : <code>String s = "tt";</code> <code>char c = s.charAt(2);</code> ou bien <code>char c = s.charAt(-1);</code>
NullPointerException	Mauvaise utilisation d'une référence. Par exemple utilisation d'un tableau d'objets créé mais non initialisé : <code>A [] a = new A[2]; // A est une classe</code> <code>a[0].x = 2; // l'attribut x est public</code>

Exemple:

```
class Animal {  
}  
  
class Chien extends Animal {  
    int taille = 80;  
}  
  
public class Conversion {  
    public static void main(String args[]) {  
        Animal animal = new Animal();  
        Chien chien = new Chien();  
        chien = (Chien) animal;  
    }  
}
```

On aura l'exception suivante : *Exception in thread "main" java.lang.ClassCastException : Animal cannot be cast to Chien.*

7.4 Méthodes des exceptions

Dans ce qui suit, une liste de méthodes disponibles dans la classe **Throwable** :

Méthode	Description
<code>public String getMessage()</code>	Retourne un message concernant l'exception produite.
<code>public String toString()</code>	Retourne le nom de la classe concaténé avec le résultat de « <code>getMessage()</code> »
<code>public void printStackTrace()</code>	Affiche le résultat de « <code>toString()</code> » avec la trace de l'erreur .

Exemple :

```
public class TestException {
public static void main(java.lang.String [] args) {
    int i = 3, j = 0;
    try {
        System.out.println("resultat = " + (i / j));
    } catch (ArithmeticException e) {
        System.out.println("1." + e.getMessage());
        System.out.println("2." + e.toString());
        System.out.print("3.");
        e.printStackTrace();
    } } }
```

Execution :

1. / by zero
2. java.lang.ArithmeticException: / by zero
3. java.lang.ArithmeticException: / by zero at TestException.main(TestException.java:5)

7.5 Capture des exceptions

Pour les différents tests, le programme s'est arrêté de façon brutale. Il est possible de capturer (to catch) ces exceptions et continuer l'exécution du programme en utilisant les 5 mots clés **try**, **catch**, **throw**, **throws** et **finally**.

La forme générale d'un bloc **try** est :

```
try {
    //traitement normale
    //Code succésible de generer une erreur
} catch (TypeException1 excepObj) {
```

```

    //traitement en cas d'exception de type TypeException1
} catch (TypeException2 excepObj) {
    //traitement en cas d'exception de type TypeException2

// ...
finally{
//code a executer avant la fin du bloc try
}
//code apres try

```

TypeException est le type d'exception généré. **excepObj** est un objet.

Remarque :

TypeException peut-être une classe prédéfinie de Java ou une classe d'exception créée par l'utilisateur.

7.6 Utilisation du bloc try-catch

7.6.1 Un seul catch

Lors de l'exécution du programme :

```

public class DiviseZero {
    public static void main(String[] args) {
        int n = 0;
        System.out.println("1/" + n + " = " + 1/n);
    }
}

```

on a obtenu l'exception **ArithmeticException**. Pour capturer cette exception, on pourra modifier le programme en utilisant le bloc **try** de la façon suivante :

```

public class DiviseZero {
    public static void main(String[] args) {
        int n = 0;
        try {
            System.out.println("1/" + n + " = " + 1 / n);
        } catch (ArithmeticException exOb) {
            System.out.println("Division par zero");
        }
        System.out.println("Reste du programme");
    }
}

```

7.6.2 plusieurs catch

Voici un exemple qui génère au moins deux exceptions :

```
import java.util.*;

public class Exceptions2 {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);

        try {
            System.out.print("Saisir un entier : ");
            n = clavier.nextInt();
            System.out.println("1/" + n + " = " + 1 / n);
        } catch (ArithmeticException e) {
            System.out.println("Impossible de diviser par 0");
            System.out.println(e.getMessage());
        } catch (InputMismatchException e) {
            System.out.println("Vous n'avez pas saisi un entier");
            System.out.println(e); //équivalent a e.toString()
            //pour recuperer le retour a la ligne
            clavier.nextLine();
        }

        System.out.println("Fin du programme");
        clavier.close();
    }
}
```

Remarque :

Puisque les classe **ArithmeticException** et **InputMismatchException** sont des sous classes de la classe **Exception**, on peut les combiner dans bloc **catch** générique qui utilise la classe **Exception** :

```

public class ExceptionsGeneriques {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);
        try {
            System.out.print("Saisir un entier : ");
            n = clavier.nextInt();
            System.out.println("1/" + n + " = " + 1 / n);
        } catch (Exception e) {
            System.out.println("Erreur\n" + e.toString());
        }
        System.out.println("Fin du programme");
        clavier.close();
    }
}

```

7.6.3 Bloc finally

Le bloc **finally** (optionnel) est toujours exécuté, même si aucune exception ne s'est produite.

Exemple :

```

public class BlocFinally {
    public static void main(String[] args) {
        int tab[] = new int[2];
        try {
            tab[2] = 1;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception : ");
            e.printStackTrace();
            //ou bien e.printStackTrace(System.out);
        } finally {
            tab[0] = 6;
            System.out.println("tab[0] = " + tab[0]);
            System.out.println("Le bloc finally est execute");
        }
    }
}

```

7.7 Exceptions personnalisées

7.7.1 Déclenchement d'une exception

Considérons la classe `Point` qui a un constructeur à deux arguments (position du point dans le plan) et une méthode `deplace()` qui permet de déplacer le point dans le plan.

Le programme s'arrêtera si au moins une des coordonnées du point est négatif.

On peut déclarer deux classes personnalisées `ErrConst` et `ErrDepl` qui héritent de la classe `Exception` comme suit :

```
class ErrDepl extends Exception{ }  
class ErrConst extends Exception{ }
```

Une méthode déclare qu'elle peut lancer (déclencher) une exception par le mot clé **throws**.

Par exemple :

```
public void deplace(int dx, int dy) throws ErrDepl {  
    // Declare que la methode deplace() peut generer une exception  
    ...  
}
```

Ensuite la méthode lance une exception, en créant une nouvelle valeur (un objet) d'exception en utilisant le mot clé **throw**.

```
public void deplace (int dx, int dy) throws ErrDepl {  
    if ((x+dx <0) || (y+dy<0)) throw new ErrDepl();  
    // Detection de l'exception et Creation  
    // d'une nouvelle valeur d'exception  
    x = x+dx ; y = y+dy;    // traitement normal  
}
```

7.7.2 Exemple complet

```
class ErrConst extends Exception {}  
class ErrDepl extends Exception {}  
  
class Point {  
    private int x, y;  
    public Point(int x, int y) throws ErrConst {  
        if ((x < 0) || (y < 0)) throw new ErrConst();  
        this.x = x ; this.y = y;  
    }  
    public void deplace(int dx, int dy) throws ErrDepl{  
        if ((x+dx < 0) || (y+dy < 0)) throw new ErrDepl();  
        x = x+dx ; y = y+dy;  
    }  
}  
  
public class Test {  
    public static void main(String[] argv) {  
        try {  
            Point a = new Point(1,4);  
            a.deplace(0,1);  
            a = new Point(7, 4);  
            a.deplace(3,-5);  
        }  
    }  
}
```

```

        a.deplace(-1,-5);
    }
    catch (ErrConst e) {
        System.out.println("Erreur de Construction");
        System.exit(-1);
    }
    catch (ErrDepl ed) {
        System.out.println("Erreur de Deplacement");
        System.exit(-1);
    }
}
}

```

7.8 Exercices

Exercice 1

Créez un programme qui permet de calculer la racine carrée d'un nombre réel (méthode `Math.sqrt()`). Gérez une exception si l'utilisateur entre un nombre négatif.

Exercice 2

Reprenez l'exercice 3 du chapitre 6 et ajoutez à la question 4 les points suivants :

1. gérez les exceptions si l'utilisateur saisi autre chose qu'un **double** ;
2. gérez les exceptions si le compte ne contient pas assez d'argent (le client veut retirer une somme d'argent supérieure à son solde).

Chapitre 8

Flux d'entrées/sorties en JAVA

8.1 Introduction

Dans les programmes des précédents chapitres, nous avons affiché des informations dans la fenêtre console en utilisant la méthode `System.out.println`. En fait, **out** est un *flux de sortie*. En Java, un flux de sortie désigne n'importe quel canal susceptible de recevoir de l'information. Il peut s'agir d'un périphérique d'affichage, comme c'était le cas pour **out**, mais il peut également s'agir d'un fichier ou encore d'une connexion à un site distant, voire d'un emplacement en mémoire centrale.

De façon comparable, il existe des flux d'entrée, c'est-à-dire des canaux délivrant de l'information sous forme d'une suite d'octets (ou de caractères). Là encore, il peut s'agir d'un périphérique de saisie (clavier), d'un fichier, d'une connexion ou d'un emplacement en mémoire centrale.

Java fournit un paquetage `java.io` qui permet de gérer les flux de données en entrée et en sortie, sous forme de caractères (exemple fichiers textes) ou sous forme binaire (octets, byte).

Java fournit quatre hiérarchies de classes pour gérer les flux de données.

1. Pour les flux binaires :
 - La classe `InputStream` et ses sous-classes pour lire des octets (`FileInputStream`) ;
 - La classe `OutputStream` et ses sous-classes pour écrire des octets (`FileOutputStream`).
2. Pour les flux de caractères :
 - La classe `Reader` et ses sous-classes pour lire des caractères (`BufferedReader`, `FileReader`) ;
 - La classe `Writer` et ses sous-classes (`BufferedWriter`, `FileWriter`).

Le cas illustré dans ce chapitre sera le cas des accès en lecture ou écriture à un fichier en caractère. Pour le flux binaire on se contente de donner quelques classes utiles.

8.2 Flux binaires

8.2.1 Flux d'entrée

Un flux d'entrée est une instance d'une sous-classe de `InputStream`. Les classes les plus couramment utilisées sont :

- `ByteArrayInputStream` permet de lire le flux d'entrée sous la forme d'octets (byte) ;
- `DataInputStream` permet de lire le flux d'entrée sous la forme de types de données primitifs de Java.
- `FileInputStream` est utilisé pour lire le contenu d'un fichier.
- `ObjectInputStream` permet de lire des objets (c-à-d des instances de classes Java) à partir du flux d'entrée, si ces objets implémentent les interfaces `java.io.Serializable` ou `java.io.Externalizable`.
- `Reader` n'est pas une sous-classe de `InputStream`, mais représente un flux d'entrée pour les chaînes de caractères.
- `Scanner` n'est pas une sous-classe de `InputStream`, mais un `Iterator` qui permet de lire un flux (fichier ou chaîne de caractères par exemple) mot par mot en définissant le délimiteur entre les mots (espace par défaut).

8.2.2 Flux de sortie

Un flux de sortie est une instance d'une sous-classe de `OutputStream`. Les classes les plus couramment utilisées sont :

- `ByteArrayOutputStream` permet d'écrire des octets vers le flux de sortie ;
- `DataOutputStream` permet d'écrire des types de données primitifs de Java vers le flux de sortie.
- `FileOutputStream` est utilisé pour écrire dans un fichier.
- `ObjectOutputStream` permet d'écrire des objets vers le flux de sortie, si ces objets implémentent les interfaces `Serializable` ou `Externalizable` .
- `Writer` n'est pas une sous-classe de `OutputStream`, mais représente un flux de sortie pour les chaînes de caractères.

8.3 Flux de caractères

8.3.1 Flux d'entrée

Classe	Description
<i>Reader</i>	classe de base
<i>InputStreamReader</i>	flux texte d'entrée
<i>FileReader</i>	fichiers texte d'entrée
<i>BufferedReader</i>	filtre pour ajouter un tampon à un flux texte d'entrée
<i>CharArrayReader</i>	flux texte d'entrée en mémoire

8.3.2 Flux de sortie

Classe	Description
<code>Writer</code>	classe de base
<code>OutputStreamWriter</code>	flux texte de sortie
<code>FileWriter</code>	fichiers texte de sortie
<code>PrintWriter</code>	flux texte de sortie avec formatage des types primitifs
<code>BufferedWriter</code>	filtre pour ajouter un tampon à un flux texte
<code>CharArrayWriter</code>	flux texte de sortie en mémoire

8.4 Démarche de lecture et d'écriture de données

La lecture de données à partir d'un flux d'entrée suit le déroulement suivant :

- **Ouverture du flux.** Elle se produit à la création d'un objet de la classe `InputStream` (`InputStreamReader`). Lors de l'appel au constructeur, on doit préciser quel élément externe est relié au flux (par exemple un nom de fichier ou un autre flux).
- **Lecture de données.** Des données provenant du flux sont lues au moyen de la méthode `read()` ou d'une méthode équivalente. La méthode précise à employer dépend du type de flux ouvert.
- **Fermeture du flux.** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

L'écriture de données vers un flux de sortie suit le même déroulement que la lecture d'un flux d'entrée :

- **Ouverture du flux.** Elle se produit à la création d'un objet de la classe `OutputStream` (`OutputStreamWriter`).
- **Écriture de données.** Des données sont écrites vers le flux au moyen de la méthode `write()` ou d'une méthode équivalente.
- **Fermeture du flux.** Quand le flux n'est plus nécessaire, il doit être fermé par la méthode `close()`.

8.5 Lecture dans un fichier

8.5.1 Lecture des données à partir d'un fichier

Supposons qu'on veut lire à partir d'un fichier texte nommé : «Test2.txt» qui contient un message quelconque, par exemple : Je suis un étudiant. Le code suivant répond à notre besoin :

```
import java.io.*;
public class LireFichier{

    public static void main(String args[]) throws IOException{
        String nomfich = "Test2.txt";
        String ligneLue;    //pour stocker chaque ligne du fichier
        try{
            BufferedReader b = new BufferedReader(new FileReader(nomfich));
            while((ligneLue = b.readLine()) != null){
                System.out.println(ligneLue);
            }
            b.close();

        }catch (Exception e){
            System.out.println(e.toString());
        }
    }
}
```

Remarque : Dans ce code, on a supposé que le fichier Test2.txt se trouve dans le même répertoire que notre programme, sinon on doit spécifier le chemin complet.

8.5.2 Exemple : découper une chaîne de caractère

Pour découper une chaîne de caractères, on utilise la fonction `split`. Soit la chaîne de caractère suivante : `Je suis un étudiant`. On veut extraire tous les mots séparés par un `espace`, i.e. “Je” puis “suis” puis “un” et enfin “étudiant”. Alors voici un morceau du code qui répond à ce besoin :

```
String ligneLue = "Je suis un étudiant"; //la chaîne a decouper
String delimiter = " "; //le delimitateur
String [ ] temp = ligneLue.split(delimiter);
                //temp est un tableau pour stocker chaque mot
for(int i=0;i<temp.length;i++){
    System.out.println(temp[i]);
}
```

Maintenant, on peut utiliser ce code en lisant d'un fichier.

Exemple complet :

```
import java.io.*;
public class LireFichier{

    public static void main(String args[]) throws IOException{
        String fich = "Test2.txt";
        String ligneLue; //pour stocker chaque ligne du fichier
        String delimiter = " ";
        String [ ] temp;
        try{
            BufferedReader b = new BufferedReader(new FileReader(fich));
            while((ligneLue = b.readLine()) != null){
                temp = ligneLue.split(delimiter);
                for(int i=0;i<temp.length;i++){
                    System.out.println(temp[i]);
                }
            }
            b.close();
        }catch (Exception e){
            System.out.println(e.toString());
        }
    }
}
```

8.6 La classe Random

La classe `java.util.Random` fournie par l'API Java permet la génération de nombres pseudo-aléatoires. Elle propose, entre autres, les méthodes suivantes :

`nextInt()`; `nextFloat()`; `nextBoolean()`; `nextInt(int n)`

8.6.1 La méthode nextInt()

`nextInt()` : calcule et renvoie le prochain nombre pseudo-aléatoire sous forme d'un nombre entier

```
Random r = new Random();  
int a = r.nextInt();
```

8.6.2 La méthode nextFloat()

`nextFloat()` : calcule et renvoie le prochain nombre pseudo-aléatoire sous forme d'un nombre réel entre 0 et 1.

```
Random r = new Random();  
float b = r.nextFloat();
```

8.6.3 La méthode nextBoolean()

`nextBoolean()` : calcule et renvoie une valeur pseudo-aléatoire sous forme d'une valeur `true` ou `false`

```
Random r = new Random();  
boolean c = r.nextBoolean();
```

8.6.4 La méthode nextInt(int n)

`nextInt(int n)` : calcule et renvoie une valeur pseudo-aléatoire entre 0 et n (exclu)

```
Random r = new Random();  
int d = r.nextInt(10);
```

8.6.5 Exemple

```
import java.util.*;  
public class Aleatoire{  
    public static void main(String [] args){  
        Random r = new Random();  
        int a = r.nextInt();  
    }  
}
```

```

    float b = r.nextFloat();
    boolean c = r.nextBoolean();
    int d = r.nextInt(10);
    System.out.println(a+"; "+b+"; "+c+"; "+d);
    for(int i=0;i<9;i++)
        System.out.print(r.nextInt(2)); //0 ou 1
    System.out.println();
}
}

```

L'exécution de ce programme affiche :

```

702106123; 0.52694154; true; 4
001111101

```

8.7 Ecriture dans un fichier

Supposons maintenant qu'on veut écrire le message « Bonjour » dans un fichier texte nommé «Test.txt», alors on doit utiliser `PrintWriter` comme suit :

```

import java.io.*;
public class EcrireFichier{

    public static void main(String args[]) throws IOException{
        String nomfich = "Test.txt";
        PrintWriter s = new PrintWriter(new FileWriter(nomfich));
        s.print("Bonjour");
        s.close();
    }
}

```

Le fichier «Test.txt» sera donc créé dans le même répertoire où se trouve notre programme, et il contient le message `Bonjour`.

Exemple :

On veut écrire dix nombres réels générés aléatoirement entre 0 et 1 dans un fichier texte : Test3.txt. Pour ceci on utilise `Math.random()`.

```

import java.io.*;
public class EcrireFichier{

    public static void main(String args[]) throws IOException{
        String nomfich = "Test3.txt";
        PrintWriter s = new PrintWriter(new FileWriter(nomfich));
        int i;
        double a;
        for(i=0;i<10;i++){

```

```

        a = Math.random();
        s.print(a+";");
    }
    s.close();
}

```

Le fichier «Test3.txt» sera donc créé et il contient une ligne comme ceci :

0.6368808900737176;0.40074971393945025;0.9257407606689951;...

8.8 Exercice

Dans une classe nous avons 20 étudiants avec leurs notes dans six modules.

1. Écrire les 120 notes générées aléatoirement entre 0 et 20 dans un fichier texte : **Notes.txt**. Chaque ligne correspond à un étudiant, et chaque colonne à un module.
2. Dans le programme principal, lire les notes à partir du fichier **Notes.txt** et calculer la moyenne de chaque étudiant.

```

10 11.5 8.4 15 17 5
7.5 10 14.6 18 9 12
....

```

Chapitre 9

De UML à JAVA

9.1 Introduction

UML (Unified Modeling Language) a été créé en 1997 pour être le langage standard de modélisation orientée objet.

UML contient différents diagrammes utilisés pour décrire de nombreux aspects du logiciel. Parmi ces diagrammes, on trouve le *diagramme de classes*.

Le *diagramme de classes* représente la structure statique du logiciel. Il décrit l'ensemble des classes qui sont utilisées ainsi que leurs associations.

Java permet de programmer tout modèle représenté sous forme de diagramme de classe UML. Dans ce qui suit, nous présentons les principales correspondances entre *UML* et *Java*.

9.2 Classe et membres

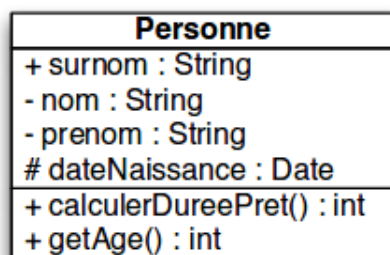


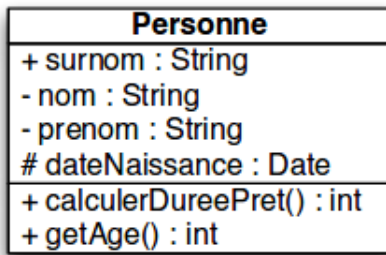
FIGURE 9.1 – Représentation d'une classe en UML

En *UML*, une classe est représentée par un rectangle, éventuellement divisé en trois parties : son nom, ses attributs et ses méthodes (voir la figure 9.1).

Un niveau de visibilité est attribué à chaque membre de la classe. *UML* utilise trois niveaux de visibilité :

- **public** (noté par +), le membre est visible par toutes les classes ;
- **privé** (noté par -), le membre n'est visible par aucune classe sauf celle qui le contient ;
- **protégé** (noté par #), le membre est visible par toutes les sous-classes de celle qui le contient et éventuellement par les classes de même package.

La figure 9.2 illustre un code Java correspondant à une classe `Personne`.



```

public class Personne {
    public String surnom;
    private String prenom;
    private String nom;
    protected Date dateNaissance;
    public int calculerDureePret() {...}
    public int getAge() {...}
}
  
```

FIGURE 9.2 – Code Java correspondant à une classe UML

9.3 Classe abstraite

En UML, le mot-clé `abstract` est accolé aux classes et méthodes abstraites. La figure 9.3 représente une telle classe. Un programme Java correspondant à cette classe abstraite peut être

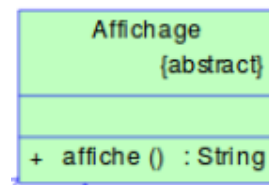


FIGURE 9.3 – Représentation d’une classe abstraite en UML

défini comme suit :

```

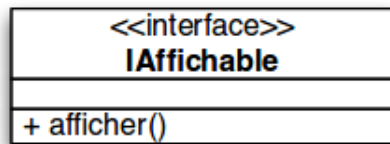
public abstract class Affichage {
    public abstract String affiche();
}
  
```

9.4 Interface

En UML, une interface est décrite par le mot-clé `interface` dans le bloc d’entête, comme présenté dans la figure 9.4.

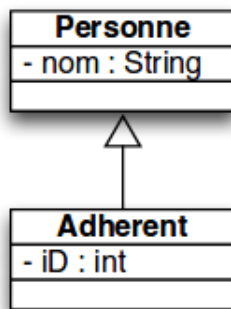
9.5 Héritage

L’héritage est représenté, en UML, par une flèche à la pointe creuse. La figure 9.5 décrit une classe mère et une classe fille, et le code Java correspondant à cette situation.



```
interface IAffichable {
    void afficher();
}
```

FIGURE 9.4 – Une interface et son code Java



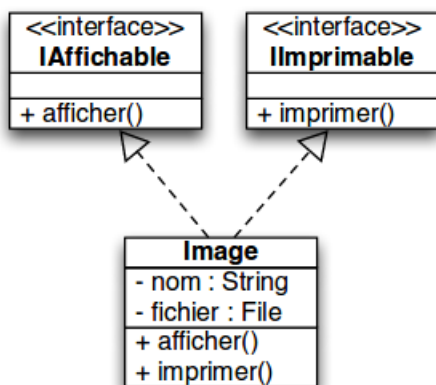
```
public class Adherent extends Personne {
    private int iD;
}
```

FIGURE 9.5 – Représentation d'héritage en UML et le code correspondant en Java

9.6 Réalisation (Implémentation)

La réalisation (implémentation) d'une interface par une classe se représente par une flèche pointillée à pointe creuse, comme illustré dans la figure 9.6.

La réalisation en Java se fait par le mot clé `implements`.



```
public class Image implements IAffichable, IImprimable {
    private String nom;
    private File fichier;
    public void afficher(){...}
    public void imprimer(){...}
}
```

FIGURE 9.6 – Code Java implémentant deux interfaces

9.7 Association

Une association peut être identifiée par un nom. Chacune de ses extrémités définit le nombre d'instances (multiplicité) des classes reliées qui sont impliquées dans cette association.

Les associations peuvent être dirigées, ce qui contraint la visibilité et la navigation dans le modèle. La direction se représente par une flèche classique. Par défaut, s'il n'y a pas de flèche, l'association est bidirectionnelle.

La figure 9.7 montre que seul *A1* connaît le *B1* auquel il est associé mais pas l'inverse. Ceci

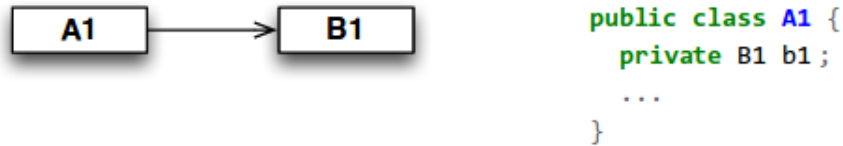


FIGURE 9.7 – Association dirigée vers B1 avec la multiplicité 1

se traduit en Java par l'ajout d'un attribut de type *B1* dans la classe *A1*.

La figure 9.8 montre que seul *A2* connaît tous les *B2* auxquels il est associé mais pas l'inverse. Ceci se traduit en Java par l'ajout d'un tableau de type *B2* dans la classe *A2*.

La figure 9.9 présente une association bidirectionnelle, ce qui se traduit en Java par l'ajout



FIGURE 9.8 – Association dirigée vers B2 avec la multiplicité * (plusieurs)

d'un attribut de type *Femme* dans la classe *Homme* et d'un attribut de type *Homme* dans la classe *Femme*.

Enfin, la figure 9.10 présente une association réflexive et son code en Java.

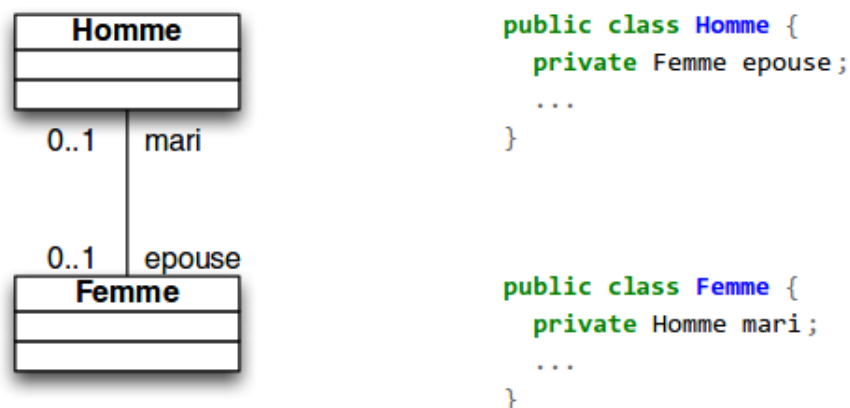
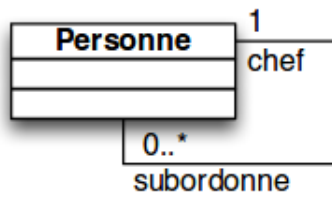


FIGURE 9.9 – Association bidirectionnelle et son code en Java

Une association est dite réflexive si les deux extrémités de l'association pointent vers la même classe.



```
public class Personne {  
    private ArrayList<Personne> subordonnes ;  
    private Personne chef ;  
    ...  
}
```

FIGURE 9.10 – Association réflexive et son code en Java

Chapitre 10

Introduction à Java graphique avec NetBeans

10.1 Introduction

Les applications informatiques nécessitent des échanges d'informations entre l'utilisateur et la machine. Cet échange d'informations peut s'effectuer avec une interface utilisateur (UI en anglais) en mode texte (ou console) ou en mode graphique.

Une interface graphique est composée de plusieurs fenêtres qui contiennent divers composants graphiques tels que boutons, listes déroulantes, menus, champ texte, etc.

Les interfaces graphiques sont souvent appelés **GUI** (pour Graphical User Interface en Anglais).

Java offre des facilités pour construire des interfaces graphiques grâce aux trois bibliothèques :

- **AWT** (Abstract Window Toolkit, JDK 1.1) ;
- **Swing** (JDK/SDK 1.2) ;
- **JavaFX**.

Actuellement **JavaFX** est la bibliothèque graphique remplaçante de **Swing** et de **AWT**. La première version de **JavaFX** 1.0 a été créée en 2008.

Dans les versions courantes 1.8, les trois bibliothèques sont toujours accessibles, mais seule **JavaFX** va, dans le futur, avoir de nouvelles fonctionnalités.

NetBeans constitue une plate forme qui permet le développement d'applications spécifiques (bibliothèque Swing).

Dans ce qui suit, on s'intéresse à cet IDE (environnement de développement intégré) pour développer quelques applications graphique en Java SE (Java standard).

Pour installer **NetBeans** avec JDK, vous pouvez consulter :

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

et choisissez **NetBeans with JDK 8**.

10.2 Création d'un projet

Après l'installation de NetBeans IDE, créez un nouveau projet en cliquant **File + New Project + Java + Java Application** puis en entrant le nom du projet.

Notre projet initial se compose uniquement d'un fichier `JavaApplication1.java`. Il s'agit par défaut d'une application console comme l'illustre la figure 10.1.

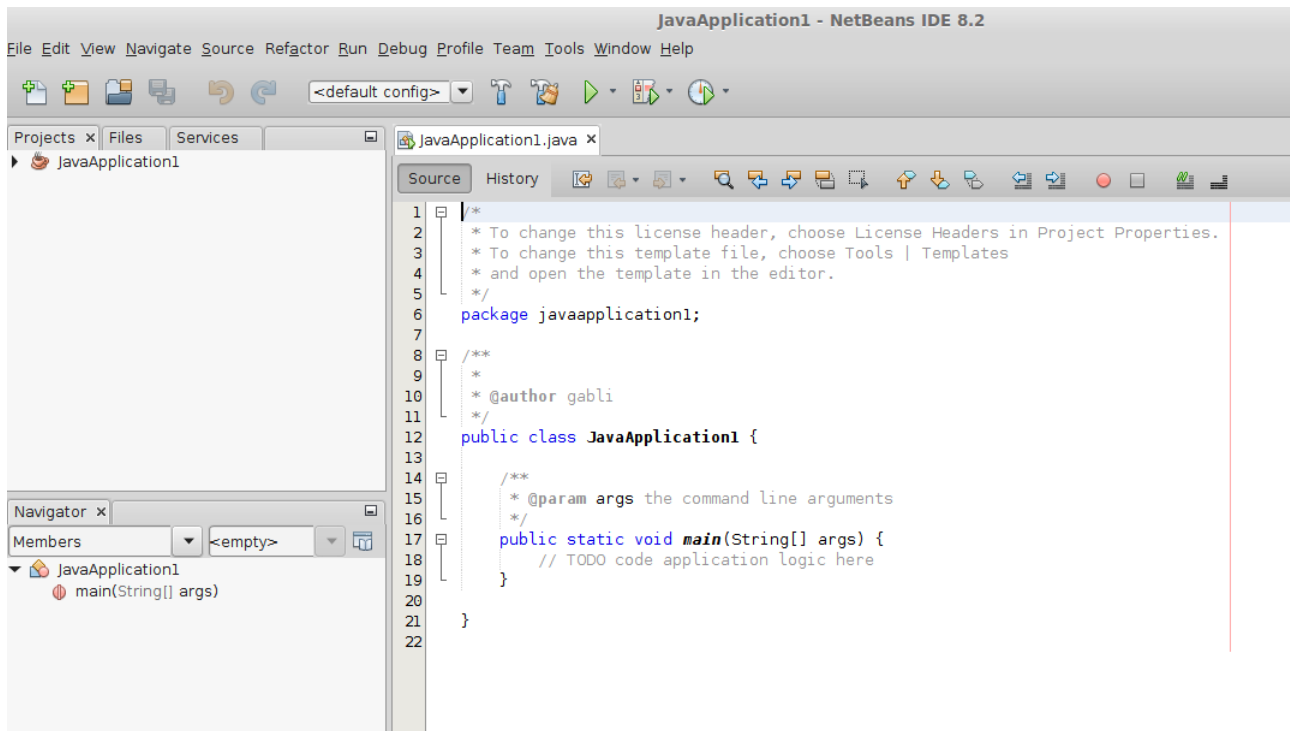


FIGURE 10.1 – Premier projet en NetBeans

10.3 Ajout d'une interface graphique dans le projet

10.3.1 Ajouter un composant visuel à votre projet

Il y a de nombreux types différents de composants visuels. Nous utiliserons ici un composant simple : `JDialog Form`.

Pour pouvoir faire de design, on doit procéder comme suit : **File + New File + Swing GUI Forms + JDialog Form** puis on choisit le nom de notre fichier, par exemple `Test.java`.

Le projet en cours est mis à jour et possède maintenant deux fichiers : `JavaApplication1.java` et `Test.java` (voir la figure 10.2).

10.3.2 Suppression d'un fichier du projet

Souhaitons maintenant supprimer le fichier `JavaApplication1.java`. Pour ceci, on fait un click droit sur le fichier `JavaApplication1.java`, et on choisit **Delete** dans le menu contextuel.

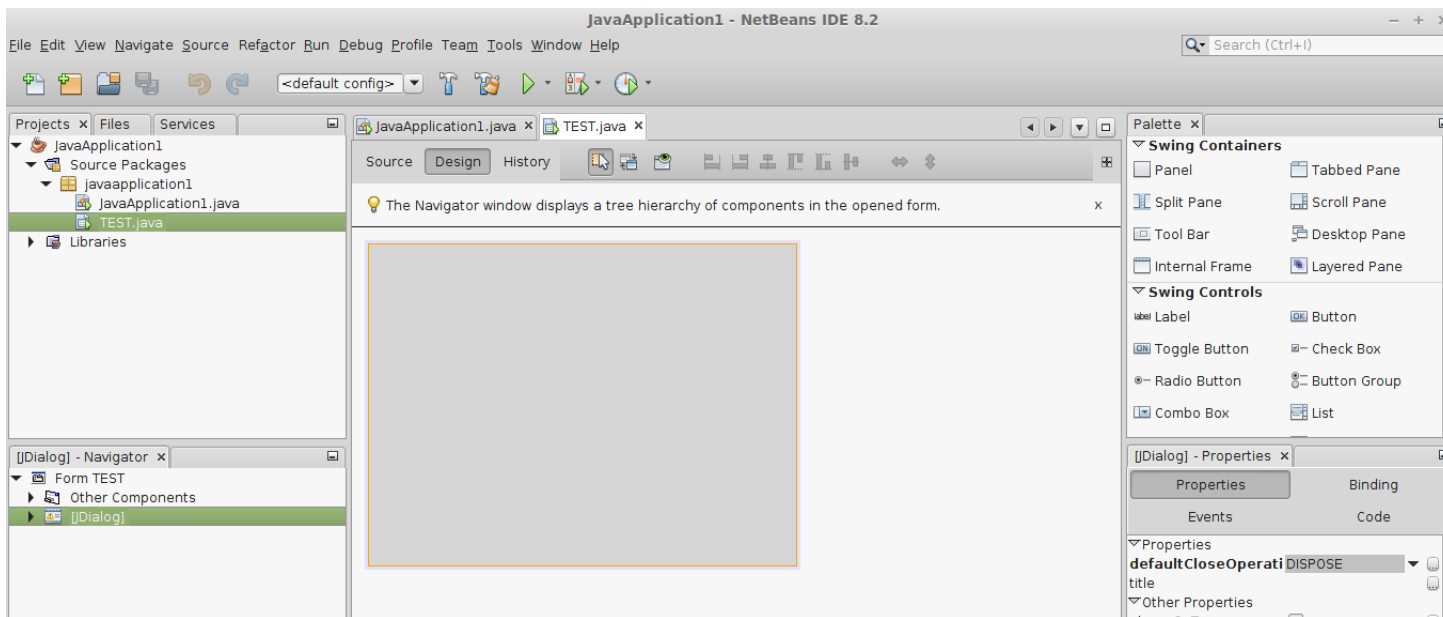


FIGURE 10.2 – Ajout d'un composant visuel

10.3.3 Compilation du projet

On clique sur le bouton **Run Project**. NetBeans vous demandera de sélectionner la classe principale. En validant **Test** comme étant la classe principale, on obtient comme résultat une fenêtre vide (sans nom, sans bouton....) comme illustré dans la figure 10.3).

10.3.4 Éléments de base d'une fenêtre

Une fenêtre graphique peut se manipuler sous forme graphique (l'onglet **Design**) ou sous forme textuelle (l'onglet **Source**). La construction d'une interface se fait en mode **Design**. Tous les objets graphiques nécessaires à une interface sont regroupés dans l'onglet **Palette** (voir figure 10.4).

Pour faire le design, il suffit de cliquer sur l'objet voulu (bouton par exemple) et sans lâcher la souris on met l'objet sur la partie design.

10.4 Attachement d'un évènement à un objet

Les évènements sont des actions déclenchées par l'utilisateur en utilisant l'interface graphique, par exemple : taper au clavier, appuyer sur un bouton, redimensionner une fenêtre ...

En Java, chaque source d'évènement (le bouton appuyé par exemple) envoie un message à un objet qui va gérer cet évènement. Cet objet est nommé **écouteur d'évènement** (**event listener**).

La classe ancêtre gérant les événements est **EventObject**.

Il existe des sous classes qui s'occupent d'évènements particuliers comme **ActionEvent** (évènements des boutons, ...), **WindowEvent** (évènements des fenêtres), ...

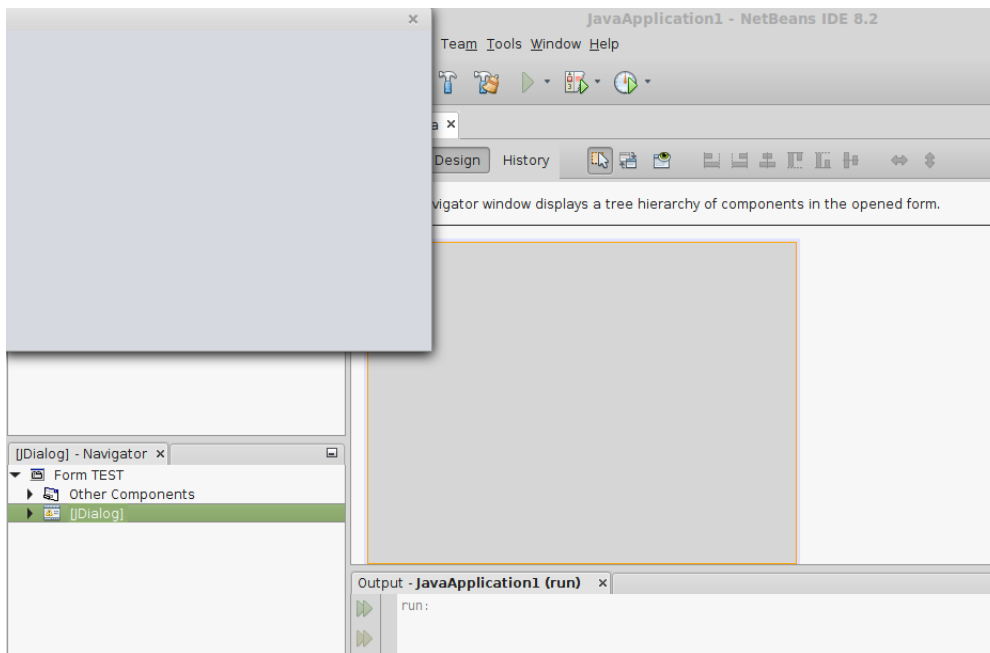


FIGURE 10.3 – Compilation du premier programme

Exemple1

Supposons qu'on veut afficher un message dans un `TextArea` après un clic sur le bouton. Pour ceci, on développe deux parties : une partie **Design** et une partie **Source**.

Le design :

dans la palette on choisit un bouton (`Button`) et une zone de texte (`TextArea`). Si on veut changer le label du bouton, on sélectionne le bouton, et dans la partie propriété (**Properties**) on choisit la propriété `Text`, et on met le label choisi, par exemple : "Afficher".

La source :

On fait d'abord un double clic sur le bouton, puis on entre le code suivant :

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
    JTextArea1.setText("Bonjour") ;
}
```

Après la compilation et si on fait un clic sur le bouton "Afficher", on obtient le message "Bonjour" dans la zone de texte comme illustré dans la figure 10.5.

Exemple2

Dans cet exemple on veut faire la somme des deux entiers entrés par l'utilisateur.

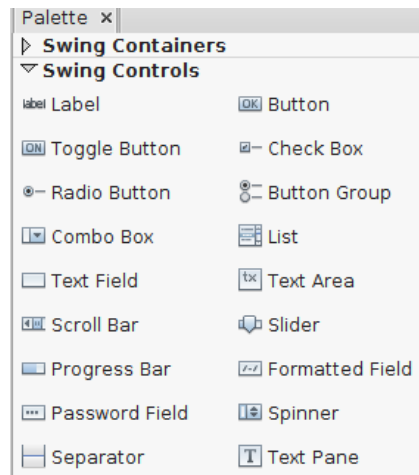


FIGURE 10.4 – Une partie de l’onglet Palette

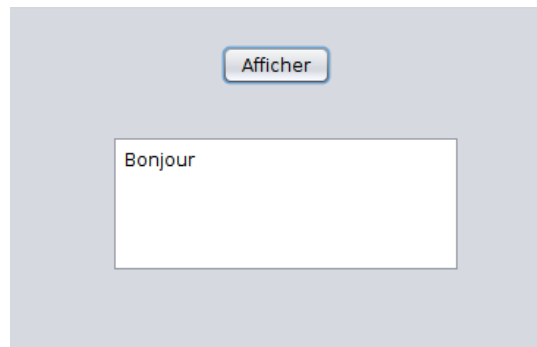


FIGURE 10.5 – Illustration du premier exemple

Le design :

On utilise deux TextFields, un TextArea et un Button.

La source :

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
    int a = Integer.parseInt(jTextField1.getText())+
        Integer.parseInt(jTextField2.getText());
    String b = String.valueOf(a);
    jTextArea1.setText(b);
}
```

L’exécution de ce code est illustrée par la figure 10.6.

Exemple3

Lire à partir d’un fichier.

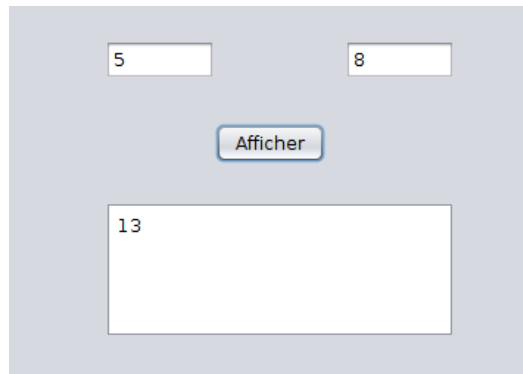


FIGURE 10.6 – Illustration du deuxième exemple

Le design :

On utilise un TextArea et un Button.

La source :

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
    String ligne;
    try{
        BufferedReader b = new BufferedReader(new FileReader("Test.txt"));
        while((ligne=b.readLine())!=null){
            jTextArea1.append(ligne+"\r\n");
        }
        b.close();
    }catch(IOException e){
        System.out.println("Erreur de lire le fichier!");
    }
}
```

Il faut remarquer que le fichier “Test.txt” doit être dans le même repertoire que le code source.

10.5 Exercices

Exercice 1

1. Dessiner les radio boutons et les checkboxes comme indiqué dans la figure 10.7.
2. lorsqu’on clique sur le bouton :
 - la fenêtre prend la couleur de la case cochée;
 - Dans le textArea on affiche le résultat du produit des checkboxes cochés et du radio-button coché.

FIGURE 10.7 – Design de l'exercice 1.

Exercice 2

Nous avons cinq étudiants et quatre modules.

1. réaliser le design comme illustré dans la figure 10.8.
2. chaque fois, on fait entrer les notes d'un étudiant dans le textArea comme suit :14;12;15;11 et on clique sur enregistrer pour les sauvegarder dans un tableau.
3. en cliquant sur le label **Moyennes**, le programme doit afficher la moyenne de chaque étudiant.
4. en cliquant sur le bouton **Trier**, le programme affichera les moyennes dans un ordre croissant.

FIGURE 10.8 – Design de l'exercice 2.

Livres, documents et liens utiles

- Programmer en Java, Claude Delannoy, Eyrolles, 5 édition.
- Initiation à la programmation orientée-objet avec le langage Java, Gauthier Picard et Laurent Vercoeur, École Nationale Supérieure des Mines.
- Cours de la programmation orientée objet avec Java, Abdelhak Lakhouaja, Faculté des Sciences, Oujda.
- <http://blog.paumard.org/cours/java/>
- <http://www.emse.fr/picard/cours/1A/java/livretJava.pdf>