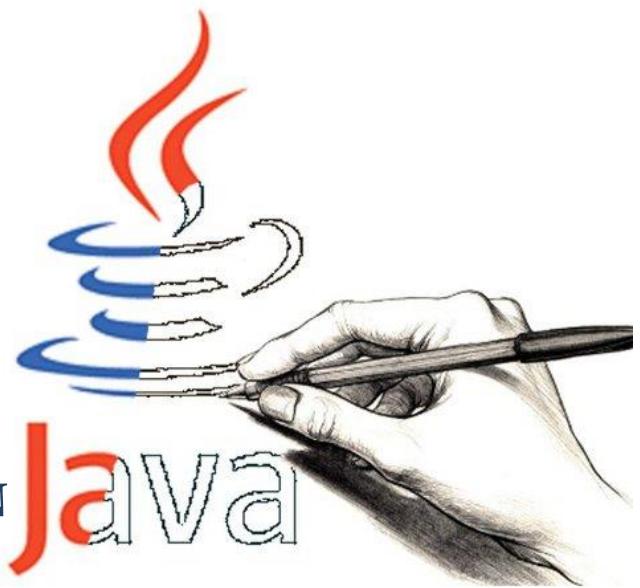




INITIATION A LA PROGRAMMATION ORIENTEE OBJET EN **Java**



Dernière révision : 27- Février -2022
Version du document : V 2.4

Préparé par : Pr. Tarik BOUDAA
Département Mathématiques et Informatique
✉ t.boudaa@uae.ac.ma
↗ <https://boudaa.github.io/>

Avant-propos

Ce cours a été conçu pour les étudiants débutants en Programmation Orientée Objet avec Java des filières Génie Informatique et Ingénierie des données de l'Ecole Nationale des Sciences Appliquées d'Al Hoceima. Il couvre les notions de base de la programmation orientée objet avec le langage Java.

Ce document est un support de cours, l'étudiant y trouvera une quantité d'informations importante lui permettant la plupart du temps de travailler seul.

Il est préparé principalement en utilisant les références citées ci-dessous :

- *Claude Delannoy. Programmer en Java. Editions Eyrolles.*
- *Joshua Bloch - Effective Java-Addison. Wesley Professional*
- *Cay Horstmann, et al. Au coeur de Java, 8ème Ed. Volume 1 Notions fondamentales 1*
- *Baland, Marie-Cécile, et al. Au Coeur de Java 2: Fonctions avancées*

Ce document comporte probablement des erreurs : toute suggestion constructive est la bienvenue à l'adresse t.boudaa@uae.ac.ma.

Table des matières

Avant-propos	2
Chapitre 1 : Les bases du langage Java.....	9
1. Introduction	9
2. Java et Portabilité	10
3. Plateforme de développement et d'exécution.....	11
3.1. JVM, JRE, et JDK	11
3.2. Environnement de développement intégré IDE.....	11
4. Types primitifs	11
4.1. Les types entiers	11
4.2. Types réels.....	12
4.3. Type char	12
4.4. Type boolean :	13
5. Déclaration et initialisation d'une variable.....	13
5.1. Déclaration d'une variable	13
5.2. Déclaration d'une constante	13
5.3. Initialisation d'une variable.....	14
6. Opérateurs	14
6.1. Les opérateurs arithmétiques :	14
6.2. Les opérateurs relationnels	14
6.3. Les opérateurs logiques	14
6.4. L'opérateur d'affectation usuel	15
6.5. Les opérateurs d'incrémentation et de décrémentation	15
6.6. Les opérateurs d'affectation élargie	15
6.7. Les opérateurs de manipulation de bits	15
6.8. L'opérateur conditionnel ou opérateur ternaire.....	15
6.9. Priorité des opérateurs	15
7. Expressions et conversions.....	16
7.1. Les conversions implicites dans les expressions	16
7.2. Conversions implicite par affectation.....	17
7.3. Conversion forcée avec l'opérateur de cast (transtypage).....	18
8. La déclaration var.....	18

9.	Premier exemple de programme Java.....	18
10.	Les commentaires.....	19
11.	Les conditions avec l'instruction if	19
11.1.	Syntaxe de l'instruction if	19
11.2.	Syntaxe de l'instruction If else.....	20
11.3.	L'instruction else if.....	21
12.	Les conditions avec l'instruction switch	21
12.1.	L'instruction switch.....	21
12.2.	Nouvelle syntaxe de l'instruction switch.....	24
12.3.	L'instruction <i>yield</i>	26
13.	Les boucles	27
13.1.	Boucle <i>while</i>	27
13.2.	Boucle <i>do ... while</i>	27
13.3.	Boucle <i>for</i>	27
13.4.	Boucle <i>for each</i>	28
13.5.	Branchement inconditionnel.....	28
Chapitre 2 : Les classes et les objets	29	
1.	Notion d'objet et classe	29
2.	Construction d'objet	30
2.1.	Les constructeurs.....	30
2.2.	L'opérateur new.....	31
2.3.	Initialisation par défaut.....	31
2.4.	Etapes de création d'initialisation d'un objet	32
3.	L'autoréférence : <i>this</i>	32
4.	Affectation entre références d'objets.....	32
5.	Comparaison des objets.....	33
6.	Portée des variables locales	35
7.	Principe d'encapsulation	35
7.1.	Encapsulation et ses avantages :	35
7.2.	Niveaux de visibilité des membres d'une classe :	37
7.3.	Mutateurs et accesseurs	38
8.	Champs et méthodes de classe	39
9.	Notion de paquetage (<i>package</i>)	42
10.	Passage de paramètres à une méthode.....	42

11.	Surcharge des méthodes (<i>Method Overloading</i>)	44
12.	La récursivité des méthodes	45
13.	Le mot clé final.....	46
14.	Classes enveloppes	46
14.1.	Création des objets des classes enveloppes	46
14.2.	Emballage et déballage (<i>boxing/ unboxing</i>) automatique.....	47
15.	Les classes internes (<i>inner-class</i>)	48
15.1.	Classes définies à l'intérieur d'une autre classe	48
15.2.	Classes internes locales	50
15.3.	Classes internes statiques	50
15.4.	Droits d'accès aux classes	50
16.	Java et la ligne de commande.....	51
16.1.	Arguments d'une méthode <i>main</i>	51
16.2.	Compilation et exécution d'un programme en ligne de commande	51
16.3.	Les archives jar.....	52
17.	Lecture des données au clavier avec la classe Scanner	52
Chapitre 3 : Tableaux, Listes et Enumérations	55	
1.	Caractéristique des tableaux JAVA.....	55
2.	Tableaux à une dimension	55
2.1.	Déclaration et mémorisation d'un tableau.....	55
2.2.	Création d'un tableau	56
2.3.	Utilisation d'un tableau	56
2.4.	Affichage d'un tableau de caractères.....	58
2.5.	Affectation de tableaux.....	58
2.6.	Utilisation de la boucle for...each	59
3.	Tableaux à deux dimensions	59
3.1.	Déclaration :	60
4.	Quelques méthodes utiles pour la manipulation des tableaux	61
4.1.	Méthode <i>Arrays.toString</i> et <i>Arrays.deepToString</i>	62
4.2.	Méthode <i>Arrays.equals</i>	62
4.3.	Méthode <i>Arrays.fill</i>	63
4.4.	Méthode <i>Arrays.sort</i>	64
4.5.	Méthode <i>Arrays.binarySearch</i>	65
4.6.	Méthode <i>System.arraycopy</i>	65

4.7.	Autres méthodes sur les tableaux	66
5.	Les listes ArrayList et LinkedList	66
5.1.	Méthodes propres à ArrayList:	67
5.2.	Méthodes propres à LinkedList:	67
5.3.	Méthodes communes à ArrayList et LinkedList :	67
5.4.	Choix de la structure de données à utiliser :	72
6.	Tableau en argument ou en retour d'une méthode	73
7.	Méthode avec nombre d'arguments variable, notation ellipse	73
8.	Les énumérations.....	75
8.1.	Définition d'un type énuméré.....	75
8.2.	Méthodes des énumérations	76
Chapitre 4 : Les chaînes de caractères	80	
1.	La classe String	80
1.1.	Déclaration et initialisation :	80
1.2.	Constructeurs de la classe String.....	80
1.3.	longueur d'une chaîne de caractères.....	81
1.4.	Concaténation de chaînes	81
1.5.	Comparaison des chaînes de caractères.....	82
1.6.	Accéder aux caractères d'une chaîne avec la méthode <i>charAt</i>	83
1.7.	Recherche dans une chaîne de caractères	83
1.8.	Remplacement de caractères	84
1.9.	Extraction de sous-chaîne	85
1.10.	Conversion entre type primitif et une chaîne.....	86
1.11.	Autres méthodes utiles	86
2.	Autres classes pour la gestion des chaînes de caractères.....	87
3.	Simplification de l'écriture des blocs de texte.....	89
4.	Résumé chaînes de caractères	90
Chapitre 5 : Héritage et Polymorphisme.....	94	
1.	Généralités sur le concept de l'héritage.....	94
2.	Mise en œuvre de l'héritage en JAVA	96
2.1.	Règles d'accès d'une classe dérivée aux membres de sa classe de base	96
2.2.	Construction et initialisation des objets dérivés	97
2.3.	Initialisation d'un objet dérivé.....	99
3.	La surdéfinition et l'héritage :	99

4.	La redéfinition des membres :	102
5.	La classe Class et la classe Object.....	106
5.1.	La classe Class.....	106
5.2.	La classe Object.....	106
5.3.	Les méthodes de la classe Object	106
6.	Principe général du polymorphisme	108
7.	Polymorphisme dans le langage JAVA	110
7.1.	La compatibilité par affectation entre un type classe et un type ascendant.....	110
7.2.	la ligature dynamique des méthodes.....	112
8.	Les classes abstraites	113
9.	Les interfaces.....	113
9.1.	Définition d'une interface.....	114
9.2.	Implémenter une interface	114
9.3.	Héritage, polymorphisme et interface :	115
9.4.	Classe interne anonymes	116
Chapitre 6 : Gestion des exceptions.....	118	
1.	Généralités sur le traitement des erreurs	118
2.	Déclencher une exception avec throw	119
3.	Utilisation d'un gestionnaire d'exception : l'instruction try/catch	120
4.	Exceptions standards et exception définies par l'utilisateur.....	121
5.	Transmission d'information au gestionnaire d'exception	123
6.	Re-déclenchement d'une exception.....	124
7.	Gestion de plusieurs exceptions	124
8.	Le bloc finally	124
9.	Les bonnes pratiques	130
Chapitre 7 : Accès aux bases de données.....	132	
1.	Accès aux bases de données avec JDBC	132
1.1.	Driver JDBC.....	132
1.2.	Etablir une connexion à une base de données	132
1.3.	Traitement des commandes SQL avec JDBC.....	133
1.4.	Les transactions	135
2.	Les injections SQL	136
3.	Utilisation d'un ORM pour la gestion de la persistance.....	137
3.1.	Différence en Framework et API.....	138

3.2.	Framework Hibernate	139
3.2.1.	Le fichier hibernate.cfg	139
3.2.2.	Contenu d'une (simple) application Hibernate.....	140
3.2.3.	Session Hibernate et SessionFactory.....	140
3.2.4.	Le cycle de vie d'un objet manipulé avec Hibernate.....	141
4.	L'API JPA	142
Chapitre 9 : La généricité dans Java		143
Chapitre 10 : Les collections.....		144
Chapitre 11 : Les Threads		145
Chapitre 12 : Initiation à la programmation graphique avec Java		146

Chapitre 1 : Les bases du langage Java

1. Introduction

Java est un langage de programmation à usage général, moderne, évolué et orienté objet, il est développé par la société Sun Microsystems, cette dernière a été ensuite rachetée par la société Oracle qui détient et maintient désormais Java. Ce langage est actuellement l'un des plus populaires et les plus utilisés dans les projets de développement informatique à travers le monde. Il existe trois éditions de Java pour des cibles distinctes selon les besoins des applications à développer ; à savoir : Java Standard Edition (Java SE), Java Enterprise Edition (Java EE) et Java Micro Edition (Java ME). Avec ces différentes éditions, les types d'applications qui peuvent être développées en Java sont nombreux et variés : Applications desktop, Applications web, Applications pour appareils mobiles et embarqués, Applications pour carte à puce, Applications temps réel, Applications pour TV, etc...

Les programmes écrit en Java s'exécute dans une machine virtuelle (JVM) qui s'agit d'un véritable processeur virtuel qui définit et implémente les éléments nécessaires au bon fonctionnement des programmes Java (allocations mémoire, interpréteur, etc...) et qui fait abstraction du matériel sous-jacent et du système d'exploitation, et permet ainsi à un programme de s'exécuter de la même manière sur des plate-formes différents.

Le succès de Java est dû à un ensemble de caractéristiques et avantages :

- Fortement typé : il garantit que les types de données employés décrivent correctement les données manipulées.
- Langage de programmation objet : l'approche objet du langage permet d'avoir un code modulaire, réutilisable et robuste.
- Gestion de la mémoire simplifiée : Java possède un mécanisme nommé ramasse-miettes (*Garbage Collector*) qui détecte automatiquement les objets inutilisés et ainsi libérer la mémoire qu'ils occupent.
- Multitâche : grâce aux threads, Java permet de programmer l'exécution simultanée et synchronisée de plusieurs traitements.
- Bibliothèque très riche : la bibliothèque fournie en standard avec Java couvre de nombreux domaines (gestion de collections, accès aux bases de données, interface utilisateur graphique, accès aux fichiers et au réseau, utilisation d'objets distribués, XML...). En plus des bibliothèques standards, il existe plusieurs Frameworks et bibliothèques pour JAVA qui facilitent le développement de différents types d'applications.
- Exécutable portable (*Write Once, Run Anywhere*) : les exécutables produits après la compilation d'un programme sont portables et peuvent s'exécuter sur n'importe quel système prenant en charge Java.
- Java est très bien documenté et dispose d'une grande communauté active de développeurs.
- Java est sécurisé : Java s'exécute dans une machine virtuelle contraignante qui contrôle l'accès à la mémoire et contrôle les opérations qui peuvent présenter des risques. Java avec ces concepts permet de minimiser les erreurs liées à la gestion de la mémoire, qui constituent dans certains cas des sources de vulnérabilité.
- Gratuit : il existe des outils gratuits de qualité pour le développement (IDE comme Eclipse, Netbeans,..) et l'exécution des applications Java

2. Java et Portabilité

Les langages comme C et C++ ne sont portables qu'au niveau code (en anglais *cross-platform in source form*), en effet, les exécutables produits après la compilation ne sont pas portables et dépendent du système d'exploitation ainsi que de l'architecture du processeur. L'un des principaux atouts de JAVA par rapport aux langages compilés comme C ou C++ est qu'il est un langage indépendant de la plate-forme, ce qui signifie qu'un même programme peut fonctionner sur différentes architectures (processeurs) et sous différents systèmes d'exploitation. En effet, à la compilation d'un programme écrit en C par exemple, le compilateur traduit le fichier source en code machine (des instructions spécifiques au processeur de l'ordinateur utilisé). Ainsi pour utiliser le même programme sur une autre plate-forme, il faut transférer le code source vers la nouvelle plate-forme et le recompiler pour produire du code machine spécifique à ce système. Contrairement, la compilation d'un programme Java ne produit pas directement un code machine mais un code intermédiaire appelé pseudo-code (ou ByteCode) similaire au code machine produit par d'autres langages, mais il n'est pas propre à un processeur donné. Il ajoute un niveau entre la source et le code machine. Les programmes JAVA reposent sur une machine virtuelle, qui convertit le pseudo-code en commandes intelligibles pour un système d'exploitation (cf. figure 1).

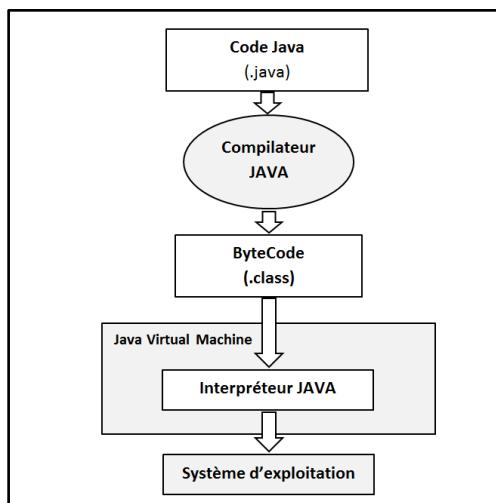


Figure 1

Le même programme semi-compilé, qui se présente sous un format pseudo-code (ByteCode), peut fonctionner sur n'importe quelle plate-forme et sous n'importe quel système d'exploitation possédant une machine Java virtuelle (cf. figure 2).

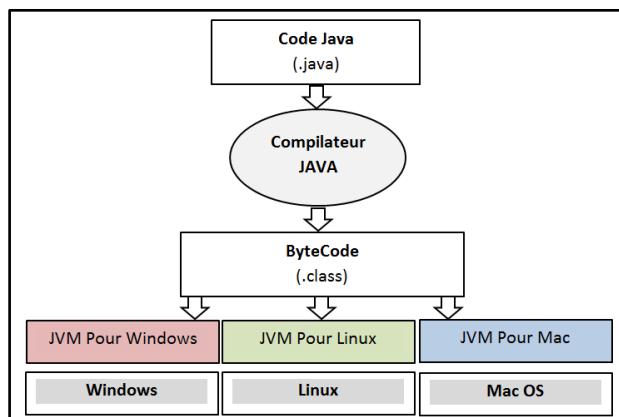


Figure 2

3. Plateforme de développement et d'exécution

3.1. JVM, JRE, et JDK

JVM (Java Virtual Machine) est un composant logiciel qui constitue une machine abstraite qui permet à un ordinateur d'exécuter le ByteCode .

Le **JRE** (Java Runtime Environment) constitue la plateforme d'exécution des programmes Java et se compose d'une machine virtuelle JVM et des bibliothèques logicielles utilisées par les programmes Java. (JRE = JVM + Library classes).

Le **JDK** « Java Development Kit » est utilisé pour le développement des applications Java, il regroupe l'ensemble des éléments permettant le développement, la mise au point et l'exécution des programmes Java, il inclut de nombreux outils de développement ainsi que l'ensemble de l'API Java dont le développeur dispose pour construire ses programmes. (JDK = JRE + Developpent Tools).

La figure ci-dessous résume les relations entre JVM, JRE et JDK.

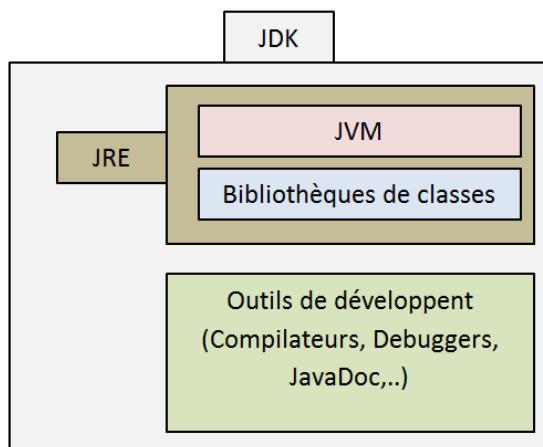


Figure 3

3.2. Environnement de développement intégré IDE.

Un environnement de développement intégré (EDI ou IDE en anglais pour *Integrated Development Environment*) est un programme regroupant un ensemble d'outils pour le développement de logiciels. Il existe de nombreux IDE pour le développement Java, citons par exemple :

- IntelliJ IDEA
- Eclipse IDE
- Apache NetBeans
- Oracle JDeveloper
- MyEclipse

4. Types primitifs

4.1. Les types entiers

Java dispose de quatre types entiers, le tableau ci-dessous représente les différents types d'entiers et leurs caractéristiques

Type	Occupation en mémoire	Intervalle (limites incluses)
int	4 octets	de -2 147 483 648 à +2 147 483 647
short	2 octets	de -32768 à +32767
long	8 octets	de -2^{63} à $+2^{63} - 1$
byte	1 octet	de -128 à +127

Remarque : La taille des types entiers primitifs peut être insuffisante dans certaines applications, par exemple l'algorithme de cryptage RSA travaille avec de très grands nombres (1024, 2048 voire 4096 bits), la classe BigInteger du package java.math peut être utilisée dans ce cas (Voir la documentation officielle <http://download.oracle.com/javase/10/docs/api/java/math/BigInteger.html>)

4.2. Types réels

Il existe deux types de réels :

Type	Occupation en mémoire	Intervalle (limites incluses)
float	4 octets	de $-1.4 * 10^{-45}$ à $+3.4 * 10^{38}$
double	8 octets	de $4.9 * 10^{-324}$ à $+1.7 * 10^{308}$ (en valeur absolue)

Les nombres de type float ont pour suffixe f, par exemple 4.114f. Les nombres réels exprimés sans ce suffixe f sont considérés comme étant du type double. Les constantes flottantes peuvent s'écrire suivant l'une des deux notations décimale ou exponentielle.

Exemples :

Notation décimale : 52.43 ; -0.38 ; -.75 ; 5. ; .97

Notations exponentielle : La notation exponentielle utilise la lettre e (ou E) (puissance de 10). Ci-dessous quelques exemples (les exemples d'une même ligne sont équivalents) :

4.25E4	4.25e+4	42.5E3
54.27E-32	542.7E-33	5427e-34
48e13	48.e13	48.0E13

Remarque : En cas de besoin d'une grande précision la classe BigDecimal du package java.math peut être utilisée. Cf. la documentation sur le lien ci-dessous :

<http://download.oracle.com/javase/10/docs/api/java/math/BigDecimal.html>

4.3. Type char

Le type caractère en Java est représenté par char qui est codé sur deux octets afin de supporter l'Unicode. Outre les caractères qu'on peut saisir au clavier, on peut entrer d'autres caractères non disponibles sur le clavier en passant par le code Unicode en hexadécimal du caractère et ceci avec la syntaxe suivante \uxxxx où xxxx représente le code. Par exemple l'alphabet arabe est entre les codes hexadécimal 0600 et 06FF.

Le type char pouvant accepter des valeurs numériques, il possède lui aussi une capacité et des limites représentées par le tableau ci-dessous.

Taille (en octets)	Valeur minimale	Valeur maximale
2	0	65 536

4.4. Type boolean :

Le type boolean (booléen) peut posséder deux valeurs, *false* ou *true*. Une variable de type booléen se déclare en utilisant le mot-clé *boolean*.

5. Déclaration et initialisation d'une variable

5.1. Déclaration d'une variable

Une variable est identifiée par un nom et se rapporte à un type de données. Le nom d'une variable Java a n caractères, le premier alphabétique, les autres alphabétiques ou numériques. La syntaxe de déclaration d'une ou plusieurs variables est :

IDENTIFICATEUR_DE_TYPE variable1, variable2, ..., variableN ;

où IDENTIFICATEUR_DE_TYPE est un type prédéfini ou bien un type objet défini par le programmeur.

Les variables peuvent être initialisées lors de leur déclaration.

Exemples :

Déclaration de deux entiers et d'une chaîne de caractères non initialisée :

```
int i, j=1 ;
String s = null;
```

Exemples d'identificateurs valides :

```
int _a;
int $c;
int _____2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

Exemples d'identificateurs invalides :

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

Java fait la différence entre majuscules et minuscules. Ainsi les variables code et Code sont différentes.

En Java les déclarations peuvent apparaître à n'importe quel emplacement du programme. Exemples :

```
System.out.println("ENSA");
int i =2;
System.out.println("Al-Hoceima");
int j,k,l =1;
float f=1.0F;
```

5.2. Déclaration d'une constante

La syntaxe de déclaration d'une constante est la suivante :

```
final IDENTIFICATEUR_DE_TYPE Nom_Constante = valeur;
```

Exemple : Déclaration de la constante Pi : `final float PI=3.14f;`

5.3. Initialisation d'une variable

Une variable peut recevoir une valeur initiale au moment de sa déclaration, comme dans :

```
int lCoeff = 2 ;
```

Cette instruction est équivalent deux instructions suivantes :

```
int lNbr ;
lNbr = 16 ;
```

En Java on peut initialiser une variable avec une expression autre qu'une constante. Voici un exemple

```
int lCoeff = 2 ;
int lNote = 13*lCoeff;
double lNbr = 0.5* lCoeff ;
```

6. Opérateurs

6.1. Les opérateurs arithmétiques :

Opérateurs	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Ces opérateurs ne sont définis que pour deux opérandes ayant le même type parmi int, long, float ou double et ils fournissent un résultat de même type que leurs opérandes.

Java utilise également deux opérateurs unaires correspondant à l'opposé noté - (comme dans -2) et à l'identité noté + (comme dans +2).

6.2. Les opérateurs relationnels

Opérateurs	Signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

6.3. Les opérateurs logiques

Opérateurs	Signification
!	négation
&	et

\wedge	ou exclusif
$ $	ou inclusif
$\&\&$	et (avec court-circuit)
$\ $	ou inclusif (avec court-circuit)

6.4. L'opérateur d'affectation usuel

- L'opérateur d'affectation en Java est $=$
- En Java les expressions peuvent avoir une valeur, ainsi une affectation telle que : $x = 2$ est une expression de valeur 2 et l'expression : $i = j = 2$ sera interprétée par $i = (j = 2)$ autrement dit, elle affectera à j la valeur 2 puis elle affectera à i la valeur de l'expression $j = 2$, c'est-à-dire 2.

6.5. Les opérateurs d'incrémentation et de décrémentation

- L'opérateur d'incrémentation en Java est $++$. Ainsi, l'expression : $++i$ incrémente de 1 la valeur de i . Noter que dans ce cas la valeur de l'expression $(++i)$ est celle de i après incrémantation.
- Lorsque l'opérateur $++$ est placé après son unique opérande, la valeur de l'expression correspondante est celle de la variable avant incrémentation.
- De la même manière, il existe un opérateur de décrémentation noté $--$ ayant les mêmes caractéristiques que $++$.

6.6. Les opérateurs d'affectation élargie

D'une manière générale, Java permet de condenser les affectations de la forme

« *variable = variable opérateur expression* » en une expression « *variable opérateur = expression* »

Ci-dessous la liste complète de tous ces opérateurs d'affectation élargie :

$+= \quad -= \quad *= \quad /= \quad %= \quad |= \quad ^= \quad &= \quad <<= \quad >>= \quad >>>=$

6.7. Les opérateurs de manipulation de bits

Opérateurs	Signification
$\&$	et (bit à bit)
$ $	ou inclusif (bit à bit)
\wedge	ou exclusif (bit à bit)
$<<$	décalage à gauche
$>>$	décalage arithmétique à droite
$>>>$	décalage logique à droite
\sim (unaire)	complément à un (bit à bit)

6.8. L'opérateur conditionnel ou opérateur ternaire

$varibale = expression_logique ? valeur1 : valeur2$ est équivalent à :

si *expression_logique* alors $varibale = valeur1$ sinon $varibale = valeur2$.

6.9. Priorité des opérateurs

La priorité d'un opérateur sur un autre et son associativité déterminent dans quel ordre seront appliqués les opérateurs. Comme en mathématiques, une opération est rendue prioritaire grâce à l'usage de parenthèses. Le tableau ci-dessous récapitule les opérateurs et leur priorité. Les opérateurs figurant sur une même ligne ont la même priorité ; ils sont rangés ligne par ligne du plus prioritaire au moins prioritaire (source <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>).

Priorité des opérateurs	
Opérateurs	priorité
postfix	<i>expr++ expr--</i>
unary	<i>++expr --expr +expr -expr ~ !</i>
multiplicative	<i>* / %</i>
additive	<i>+ -</i>
shift	<i><< >> >>></i>
relational	<i>< > <= >= instanceof</i>
equality	<i>== !=</i>
bitwise AND	<i>&</i>
bitwise exclusive OR	<i>^</i>
bitwise inclusive OR	<i> </i>
logical AND	<i>&&</i>
logical OR	<i> </i>
ternary	<i>? :</i>
assignment	<i>= += -= *= /= %= &= ^= = <<= >>= >>>=</i>

7. Expressions et conversions

7.1. Les conversions implicites dans les expressions

En Java on peut écrire des expressions dans lesquelles interviennent des opérandes de types différents. Dans ce cas des conversions de types peuvent être faites automatiquement par le compilateur. On distingue deux cas :

Les conversions d'ajustement de type : Elles se font selon cette hiérarchie :

int → long → float → double

Les promotions numériques : Les opérateurs numériques ne sont pas définis pour les types byte, char et short. Pour régler ce problème Java prévoit que toute valeur de l'un de ces types apparaissant dans une expression est d'abord convertie en int.

Exemples :

Si n est de type int, p de type long et x de type float, l'expression : n * p + x sera évaluée suivant le schéma ci-dessous :

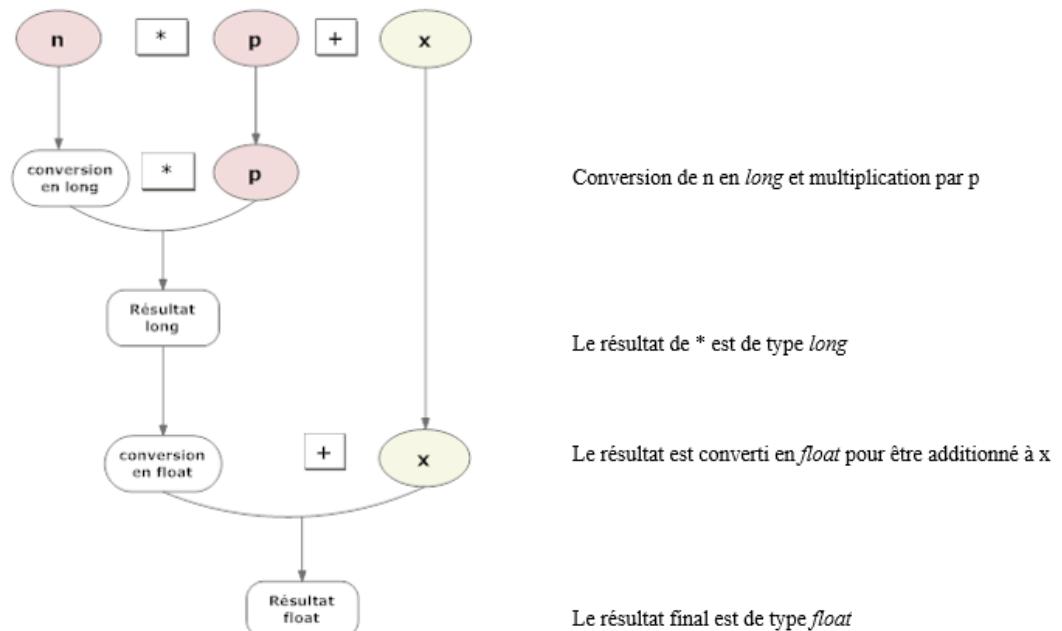


Figure 4

Dans l'exemple ci-dessous, si `n`, `m` sont de type `short`, `p` de type `int` et `x` de type `float`, l'expression : `n * p + m + x` est évaluée ainsi :

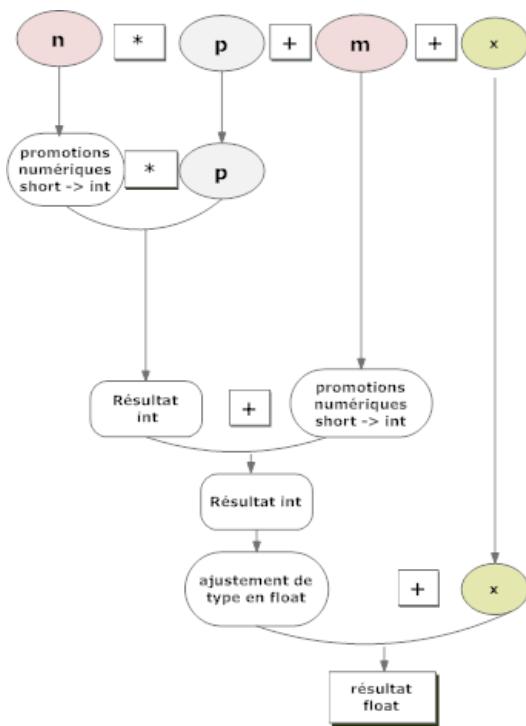


Figure 5

7.2. Conversions implicite par affectation

Les différentes possibilités sont :

`byte → short → int → long → float → double` et `char → int`

Cas particulier des constantes : On peut affecter n'importe quelle expression constante entière à une variable de type byte, short ou char, à condition que sa valeur soit représentable dans le type destinataire.
Exemple :

```
short p ;  
p = 47 ; (Ici on affecte une constante de type int à short)
```

7.3. Conversion forcée avec l'opérateur de cast (transtypage)

Le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'un opérateur nommé cast.

Exemple : double d = (double) 3 / 2 ;// force la conversion 3 → 3.0

Le résultat sera donc évalué de la façon suivante : (double) 3 / 2 → 3.0 / 2 → 3.0 / 2.0 → 1.5

8. La déclaration var

Depuis la version 10 du Java, il est devenu possible de laisser le compilateur décider du type d'une variable, quand il dispose des informations nécessaires et ceci en utilisant le mot clé *var*.

Par exemple, au lieu de : `int n = 11 ;` on peut utiliser :

```
var n = 11 ; // n sera dans ce cas de type int
```

Ici le compilateur infère le type de variable en se basant sur le type de la constante 11.

On considère un autre exemple :

```
long a = 3 ; float x = 3.5f  
  
var test = a * x + 3.22 ;
```

ici `a * x` est de type *float* et puisque 3.22 est de type *double* le résultat de `a * x + 3.22` est de type *double* ainsi, le compilateur attribuera le type *double* à la variable *test*.

9. Premier exemple de programme Java

Ci-dessous un exemple très simple de programme qui se contente d'afficher dans la fenêtre console le texte : "ENSA d'Al-Hoceima" :

```
public class FirstProgram {  
    public static void main(String args[]) {  
        System.out.println("ENSA d'Al-Hoceima");  
    }  
}
```

La structure générale du programme précédent correspond à la définition d'une classe nommée *FirstProgram*. La première ligne **package** *test* permet de définir le package dont on veut mettre notre classe *FirstProgram*. La deuxième ligne identifie cette classe ; elle est suivie de la définition de la méthode principale (Méthode *main*).

L'instruction **System.out.println** permet d'afficher le texte passé en paramètre sur la console.

10. Les commentaires

Il existe 3 types de commentaires :

Les commentaires usuels ayant la même syntaxe du langage C. Ces commentaires sont formé d'un texte quelconque inséré entre les deux symboles : /* et */ . Ces commentaires peuvent apparaître à n'importe quel endroit et ils peuvent s'étaler sur plusieurs lignes.

Exemple :

```
public class Main {  
    public static void main(String[] args) {  
        /*  
         * Un exemple de  
         * commentaire sur  
         * plusieurs ligne  
         */  
    }  
}
```

Un autre type de commentaire est indiqué par le symbole //. Ces commentaires s'écrivent sur une seule ligne.

Exemple :

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Bonjour"); //Affichage de bon jour sur l'écran  
        //Affichage d'un message sur l'écran  
        System.out.println("Affichage");  
    }  
}
```

Le dernier type est les commentaires utilisés pour écrire la documentation du code (commentaires javadoc). Généralement ce type de commentaires est utilisé pour documenter une classe et ses membres. Ils commencent par /** et se terminent par */. L'intérêt de ces commentaires est la possibilité de les transformer par des outils automatique en une documentation sous un format HTML par exemple.

Exemple :

```
/**  
 * Cette classe est un exemple de cours  
 *  
 * @author Tarik BOUDAA  
 *  
 */  
public class Main {  
    public static void main(String[] args) {  
    }  
}
```

11. Les conditions avec l'instruction if

11.1. Syntaxe de l'instruction if

Pour le cas d'une seule instruction on peut utiliser l'une des syntaxes ci-dessous :

```
if(booleanExpression)  
    instruction ;
```

ou

```
if(booleanExpression) {
    instruction ;
}
```

Pour le cas d'un bloc d'instructions il faut utiliser les accolades :

```
if(booleanExpression) {
    instruction1;
    instruction2;
    ...
    instructionN;
}
```

Attention : En Java, la condition de l'instruction *if* doit être obligatoirement une expression de type *boolean*. Une condition comme *if(var=0)* provoque donc une erreur de compilation.

Exemple

```
public class Main {
    public static void main(String[] args) {
        int x = 200;
        int y = 180;
        if (x > y) {
            System.out.println("x est plus grand que y");
        }
    }
}
```

A l'exécution on obtient :



The screenshot shows a Java application running in an IDE. The console tab displays the output of the program: "x est plus grand que y". The tabs at the top include "Problems", "Javadoc", "Declaration", and "Console". The status bar at the bottom indicates the application is terminated.

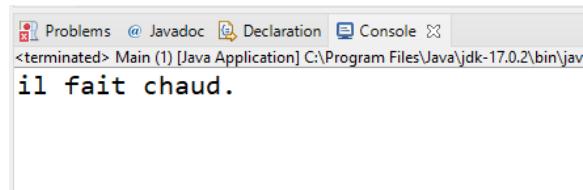
11.2. Syntaxe de l'instruction If else

```
if(booleanExpression) {
    instruction1;
    ...
    instructionN;
} else{
    ...
}
```

Exemple

```
public class Main {
    public static void main(String[] args) {
        int temperature = 40;
        if (temperature < 28) {
            System.out.println("il fait beau temps.");
        } else {
            System.out.println("il fait chaud.");
        }
    }
}
```

A l'exécution on obtient :



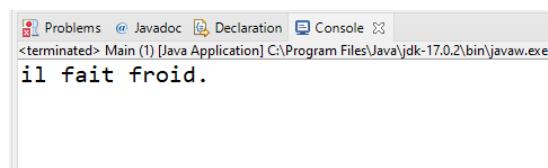
The screenshot shows a Java IDE interface with tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of a Java application named 'Main'. The output is: '<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\jav il fait chaud.'

11.3. L'instruction else if

```
if (condition1) {
    // bloc de code à exécuter si condition1 est vraie
} else if (condition2) {
    // bloc de code à exécuter si la condition1 est fausse et la condition2
    est vraie
} else {
    // bloc de code à exécuter si la condition1 est fausse et la condition2
    //est fausse
}

public class Main {
    public static void main(String[] args) {
        int temperature = 20;
        if (temperature < 28 && temperature > 22) {
            System.out.println("il fait beau temps.");
        } else if (temperature > 40 ) {
            System.out.println("il fait très chaud.");
        }
        else if (temperature > 28) {
            System.out.println("il fait chaud.");
        }
        else{
            System.out.println("il fait froid.");
        }
    }
}
```

A l'exécution on obtient :



The screenshot shows a Java IDE interface with tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of a Java application named 'Main'. The output is: '<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe il fait froid.'

12. Les conditions avec l'instruction switch

12.1. L'instruction switch

```
switch (expression)
{
    case constante_1 : [ suite_d'instructions_1 ]
    case constante_2 : [ suite_d'instructions_2 ]
    .....
    case constante_n : [ suite_d'instructions_n ]
    [ default : suite_d'instructions ]
}
```

- *expression* est une expression de l'un des types *byte*, *short*, *char* ou *int*, énuméré, ou *String* (depuis la version 7 du Java).
- *constante_i* est une expression constante d'un type compatible par affectation avec le type de *expression*,
- *suite_d'instructions_i* est une séquence d'instructions quelconques.

N.B. : les crochets [et] signifient que ce qu'ils renferment est facultatif.

Exemple :

```
public class Main {  
    public static void main(String[] args) {  
        int n = 2;  
        switch (n) {  
            case 1:  
                System.out.println("Si 1");  
                System.out.println("Dans ce cas c'est un");  
                break;  
            case 2:  
                System.out.println("Si 2");  
                System.out.println("Dans ce cas c'est Deux");  
                break;  
            case 4:  
                System.out.println("Si 4");  
                System.out.println("Dans ce cas c'est Quatre");  
                break;  
            default:  
                System.out.println("Si aucune");  
                System.out.println("Dans ce cas aucune des valeurs");  
                break;  
        }  
    }  
}
```

A l'exécution on obtient :

```
Problems @ Javadoc Declaration Console  
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe  
Si 2  
Dans ce cas c'est Deux
```

Remarque :

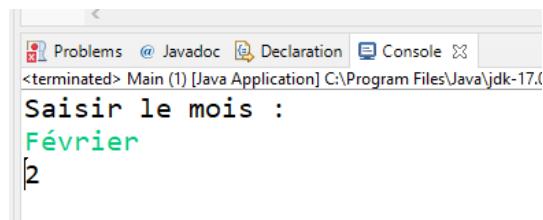
L'instruction *switch* ne permettait pas dans les anciennes versions du Java de tester autre que les valeurs numériques entières (ou des caractères). Les nouvelles versions de Java permettent de tester également des constantes de type chaînes de caractères.

Exemple :

```
import java.util.Scanner;  
public class Test {  
    public static void main(String[] args) {  
        System.out.println("Saisir le mois : ");  
        //Lire une chaîne de caractère à la console  
        Scanner sc= new Scanner(System.in);  
        String month = sc.nextLine();
```

```
int monthNumber = 0;
switch (month) {
    case "Janvier":
        monthNumber = 1;
        break;
    case "Février":
        monthNumber = 2;
        break;
    case "Mars":
        monthNumber = 3;
        break;
    case "Avril":
        monthNumber = 4;
        break;
    case "Mai":
        monthNumber = 5;
        break;
    case "Juin":
        monthNumber = 6;
        break;
    case "Juillet":
        monthNumber = 7;
        break;
    case "Août":
        monthNumber = 8;
        break;
    case "Septembre":
        monthNumber = 9;
        break;
    case "Octobre":
        monthNumber = 10;
        break;
    case "Novembre":
        monthNumber = 11;
        break;
    case "Décembre":
        monthNumber = 12;
        break;
    default:
        monthNumber = 0;
        break;
}
if (monthNumber == 0) {
    System.out.println("Mois invalide");
} else {
    System.out.println(monthNumber);
}
```

A l'exécution on obtient :



```
Saisir le mois :
Février
```

12.2. Nouvelle syntaxe de l'instruction switch

A partir de la version 14, Java propose une syntaxe plus moderne pour cette instruction dans laquelle il n'est plus besoin d'indiquer les *break* et qui simplifie les situations d'étiquettes multiples (plusieurs valeurs dans l'instruction *case*). La nouvelle syntaxe utilise le symbole "*->*" à la place de ":".

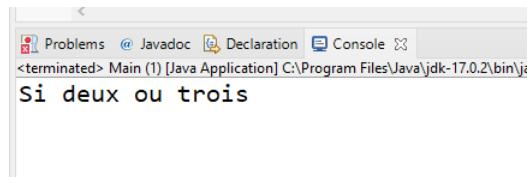
Syntaxe :

```
switch (expression)
{ case suite_de_constants_1 -> instruction_1
    case suite_de_constants_2 -> instruction_2
    .....
    case suite_de_constants_n -> instruction_n
    [ default -> instruction ]
}
```

Exemple :

```
public class Main {
    public static void main(String[] args) {
        int n = 2;
        switch (n)
        {
            case 1 -> System.out.println("Si Un");
            case 2, 3 -> System.out.println("Si deux ou trois");
            case 4 -> System.out.println("si quatre");
            default -> System.out.println ("Rien");
        }
    }
}
```

A l'exécution on obtient :



Dans le cas de plusieurs instructions associées à un *case* il faut utiliser les accolades pour délimiter le bloc d'instruction qui lui est associé.

Syntaxe :

```
switch (expression)
{
    case suite_de_constants_1 -> {
        instruction_1_1 ;
        ...
        instruction_1_n ;
    }
    case suite_de_constants_2 -> {
        instruction_2_1 ;
        ...
        instruction_2_n ;
    }
    .....
    case suite_de_constants_n -> {
        instruction_n_1 ;
        ...
    }
}
```

```

                instruction_n_n ;
            }

        [ default ->
            {
                instruction_d_1 ;
                ...
                instruction_d_n ;
            }
        ]
    }
}
```

Exemple :

```

public class Main {
    public static void main(String[] args) {
        int n = 2;
        switch (n) {
            case 1, 2, 3 -> {
                System.out.println("Un");
                System.out.println("Deux");
                System.out.println("Trois");
            }
            case 0 -> {
                System.out.println("Zéro");
                System.out.println("Rien");
            }
            case 4 -> System.out.println("Quatre");
            default -> {
                System.out.println("Par défaut");
                System.out.println("Aucune valeur");
            }
        }
    }
}
```

A l'exécution on obtient :

The screenshot shows a Java application window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab displays the output of the program: "Un", "Deux", and "Trois".

Remarque :

Java 14 permet d'utiliser *switch* pour écrire une expression dont la valeur est conditionnée par la valeur d'une variable.

Considérons, par exemple, le code suivant dont l'instruction *switch* permet d'affecter à la variable entière *response* la valeur 1 ou 0 en fonction de la valeur de la variable *test*

```

public class Main {
    public static void main(String[] args) {
        String test ="Vrai";
        int response ;
        switch (test)
        { case "Vrai" , "Ok" , "Yes" -> response = 1 ;
          default -> response = 0 ;
        }
    }
}
```

```

        System.out.println("La réponse est :" +response);
    }
}

```

A l'exécution on obtient :

The screenshot shows a Java application running in an IDE. The title bar says 'Main (1) [Java Application] C:\Program Files\Java\jdk-11\bin'. The 'Console' tab is selected, displaying the output: 'La réponse est :1'.

On peut réécrire le code précédent de la façon suivante :

```

public class Main {
    public static void main(String[] args) {
        String test = "Vrai";
        int response = switch (test) {
            case "Vrai", "Ok", "Yes" -> response = 1;
            default -> response = 0;
        }; //Attention à ne pas oublier ce point-virgule
        System.out.println(response);
    }
}

```

12.3. L'instruction *yield*

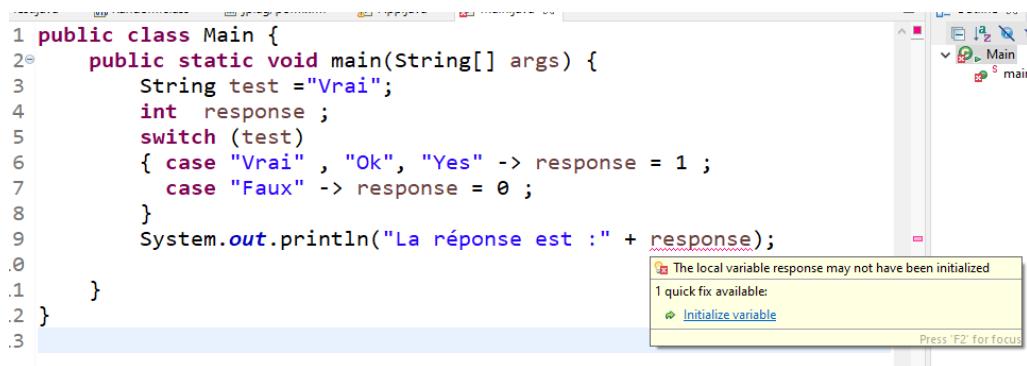
Le mot clé *yield* peut être utilisé pour retourner une valeur quand le bloc d'instructions associé à un *case* contient plus qu'une simple affectation d'une valeur à la variable à retourner par l'expression *switch*:

```

public class Main {
    public static void main(String[] args) {
        String test = "Vrai";
        int response = switch (test) {
            case "Vrai", "Ok", "Yes" -> {
                System.out.println("OK");
                yield 1;
            }
            default -> {
                System.out.println("NO");
                yield 0;
            }
        };
        System.out.println(response);
    }
}

```

Remarque : Dans une instruction *switch* (quelle que soit la syntaxe utilisée), il n'est pas obligatoire que les étiquettes indiquées dans les instructions *case* couvrent toutes les valeurs possibles. En revanche, cela le devient obligatoire dans une expression *switch*, afin d'éviter le risque de variable non définie. Dans le cas contraire, on obtient une erreur de compilation. Par exemple dans le cas suivant si la valeur de la variable *test* est égale à une valeur différente de "Vrai" et "Faux" la variable *reponse* ne sera pas initialisée c'est pour cela on obtient l'erreur indiquée dans le code ci-dessous :



A screenshot of an IDE showing a Java file named Main.java. The code contains a switch statement where the variable 'response' is assigned a value based on the string 'test'. A tooltip appears over the line 'System.out.println("La réponse est :" + response);' indicating a warning: 'The local variable response may not have been initialized' with a quick fix 'Initialize variable' available.

```
1 public class Main {  
2     public static void main(String[] args) {  
3         String test = "Vrai";  
4         int response ;  
5         switch (test)  
6             { case "Vrai" , "Ok", "Yes" -> response = 1 ;  
7              case "Faux" -> response = 0 ;  
8             }  
9         System.out.println("La réponse est :" + response);  
.0  
.1     }  
.2 }  
.3
```

13. Les boucles

13.1. Boucle while

```
while(booleanExpression)  
    instruction1 ;
```

Et dans le cas d'un bloc d'instructions :

```
while(booleanExpression) {  
    instruction1;  
    instruction2;  
    ..  
    instructionN;  
}
```

13.2. Boucle do ... while

```
do    instruction;  
while(booleanExpression);
```

Et dans le cas d'un bloc d'instructions :

```
do{  
    instruction1;  
    ..  
    instructionN;  
}while(booleanExpression);
```

13.3. Boucle for

```
for(expressionInitial,expressionBooleene; expressionIncrementation){  
    instruction ;
```

Et dans le cas d'un bloc d'instructions :

```
for(expressionInitial,expressionBooleene; expressionIncrementation){  
    instruction1;  
    instruction2;  
    ..  
    instructionN;  
}
```

13.4. Boucle *for each*

A partir de JDK 5.0 java dispose d'une autre boucle nommée *for... each*. Elle ne peut être utilisée qu'au parcours des éléments d'une collection ou d'un tableau. Cette boucle s'utilise en lecture seule, ainsi, on ne pas l'utiliser pour modifier un tableau ou une collection.

13.5. Branchement inconditionnel

Instruction	Effet
<code>break ;</code>	Sortir de la boucle courante
<code>continue ;</code>	Retour à l'évaluation de la condition du while ou de l'instruction <i>expressionIncrementation</i> du for courant sans terminer les instructions du bloc.
<code>return ; ou return value ;</code>	Sortir de la boucle courante et de la méthode courante

Chapitre 2 : Les classes et les objets

1. Notion d'objet et classe

Un objet est une entité qui comprend :

- une partie structurelle qui décrit son état (caractéristiques) et ses liens avec d'autres objets (attributs ou propriétés)
- une partie opérationnelle qui décrit ses comportements (méthodes).

Un objet est créé selon un modèle ou un prototype qu'on appelle une classe. Les objets sont des instances d'une classe.

Exemple : On peut modéliser un étudiant par la classe Etudiant suivante :

```
public class Etudiant {  
  
    // attributs (état)  
    private String nom;  
    private String prenom;  
    private String cin;  
    private int age;  
  
    // Constructeur sans arguments  
    public Etudiant() {  
        // instructions d'initialisation  
    }  
    // Constructeur avec arguments  
    public Etudiant(String pNom, String pPrenom, String pCin, int pAge)  
    {  
        nom = pNom;  
        prenom = pPrenom;  
        cin = pCin;  
        age = pAge;  
    }  
    //Méthodes  
    public void afficheEtudiant() {  
        System.out.println("Nom :" + nom);  
        System.out.println("Prénom :" + prenom);  
        System.out.println("CIN :" + cin);  
        System.out.println("Age :" + age);  
    }  
    public int incrementAge() {  
        return ++age;  
    }  
}
```

Le mot-clé *private* précise que les attributs nom, prenom, cin et age ne seront pas accessibles en dehors des méthodes propres aux instances de la classe. (Encapsulation).

La définition d'une méthode se compose d'un bloc d'instructions et d'un en-tête qui détermine le mode d'accès (*private/public/protected/package*), le type de retour, le nom de la méthode, les arguments et leurs types. Par exemple dans la classe précédente deux méthodes ont été définies *afficheEtudiant* et *incrementAge*. Le mode d'accès de la méthode *incrementAge* est : public, donc cette méthode peut être appelée depuis l'extérieur de cette classe (i.e. par des objets qui ne sont pas forcément des instances de cette classe), son type de retour est int et elle est sans arguments.

2. Construction d'objet

2.1. Les constructeurs

Un constructeur est une méthode spécifique portant le même nom que la classe ; il peut posséder des arguments ou non, mais il est toujours sans valeur de retour. Il sert à instancier des objets

La classe précédente a deux constructeurs (un sans arguments et l'autre avec arguments). Une initialisation d'un objet Etudiant peut se faire donc par l'un des constructeurs comme dans les deux exemples ci-dessous :

```
//En utilisant le premier constructeur :  
Etudiant ls1 = new Etudiant();  
  
//En utilisant le deuxième constructeur :  
Etudiant ls2 = new Etudiant("boudaa", "mohamed", "R11111", 19);
```

Une classe qui ne déclare aucun constructeur explicitement en possède en fait toujours un : le constructeur vide (qui ne fait rien) par défaut, qui ne prend aucun paramètre.

Exemple :

```
public class Point {  
    // attributs  
    private float x;  
    private float y;  
    private float z;  
}
```

Un objet de type Point peut être instancié en faisant appel au constructeur par défaut ainsi :

```
Point lPt = new Point();
```

Si on définit un constructeur explicite (avec arguments), alors ce constructeur vide par défaut n'existe plus; on doit le déclarer explicitement si l'on veut encore l'utiliser.

Ainsi le code suivant ne se compile pas :

```
public class Etudiant {  
    // attributs  
    private String nom;  
    private String prenom;  
    private String cin;  
    private int age;  
    public Etudiant (String pNom, String pPrenom, String pCin, int pAge) {  
        nom = pNom;  
        prenom = pPrenom;  
        cin = pCin;  
        age = pAge;  
    }  
    public static void main(String[] args) {  
        Etudiant lStdt = new Etudiant (); // Erreur de compilation  
    }  
}
```

L'erreur disparaît après redéfinition explicite du constructeur par défaut :

```
public class Etudiant {  
    // attributs  
    private String nom;  
    private String prenom;  
    private String cin;  
    private int age;
```

```

public Etudiant () {
}
public Etudiant (String pNom, String pPrenom, String pCin, int pAge) {
    nom = pNom;
    prenom = pPrenom;
    cin = pCin;
    age = pAge;
}
public static void main(String[] args) {
    Etudiant lStdnt = new Etudiant ();
}
}

```

2.2. L'opérateur new

L'allocation d'un emplacement pour un objet en mémoire se fait par l'appel à l'opérateur nommé *new*.

Concéderons la suite d'instructions

```

Etudiant lStudent1 = new Etudiant ();
Etudiant lStudent2 = new Etudiant ();
Etudiant lStudent3 = null; //ou Etudiant lStudent1; )

```

La troisième instruction n'instancie aucun objet, il se contente juste de réserver un emplacement mémoire pour y stocker la référence à un objet.

La première instruction et la deuxième instruction, chacune instancient un objet. L'instruction `Etudiant lStudent1 = new Etudiant ();`

Effectue deux choses :

- créer un objet de type Etudiant par l'appel du constructeur par : `new Etudiant ()`
- la référence renvoyée est affecté à la variable `lStudent1`

La situation en mémoire est présentée par la figure ci-dessous :

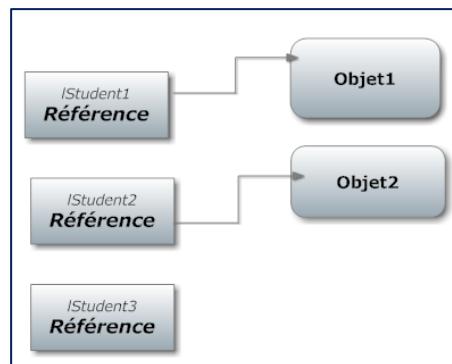


Figure 6

2.3. Initialisation par défaut

Les champs d'un objet peuvent être initialisés explicitement au moment de la déclaration. Sinon ces champs seront initialisés par défaut. Ces valeurs par défaut sont données dans le tableau ci-dessous

Type champ	Valeur par défaut
boolean	false
char	Caractère du code nul
Les types entiers	0
Les réels	0.0
Les objets	null

Contrairement à ce qui se passe pour les champs d'un objet les variables locales ne sont pas initialisées automatiquement, ainsi le code ci-dessous ne se compile pas :

```
public static void main(String[] args) {
    int i;
    System.out.println(i); // Erreur de compilation par ce que la
                           // variable i n'est pas initialisée
}
```

2.4. Etapes de création d'initialisation d'un objet

La création d'un objet passe toujours ; dans l'ordre ; par les opérations suivantes :

- Une initialisation (par défaut et explicite) de tous les champs de l'objet,
- L'exécution des instructions du corps du constructeur.

Ainsi, les champs usuels se trouvent initialisés : d'abord à une valeur par défaut, ensuite à une valeur fournie (éventuellement) lors de leur déclaration, enfin par le constructeur.

3. L'autoréférence : *this*

Dans une méthode, on peut accéder à la référence de l'instance sur laquelle la méthode est appelée par le mot clé *this*. Ce mot clé est utilisé principalement :

- Dans un constructeur, pour appeler un autre constructeur de la même classe. **Dans ce cas l'appel *this(...)* doit obligatoirement être la première instruction du constructeur.**

```
class MaClasse {
    public MaClasse(int a, int b) { ... }

    public MaClasse(int c) {
        this(c, 0); // Appelle le constructeur MaClasse(int a, int b)
    }

    public MaClasse() {
        this(10); // Appelle le constructeur MaClasse(int c)
    }
}
```

- lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode. (cette instruction n'est pas obligatoire qu'il soit la première de la méthode)
- Parfois juste pour lever une ambiguïté. Comme dans l'exemple suivant :

```
public class Etudiant {
    private String nom;
    Etudiant (String nom) {
        this.nom = nom; // Pour lever l'ambiguïté sur le mot « nom » et
                      // différentier le nom du paramètre et de l'attribut
    }
}
```

4. Affectation entre références d'objets

Considérons les instructions suivantes :

```
Etudiant lStudent1 = new Etudiant ();
Etudiant lStudent2 = new Etudiant ();
lStudent2 = lStudent1;
```

Les deux premières instructions peuvent être schématisées par la figure suivante :

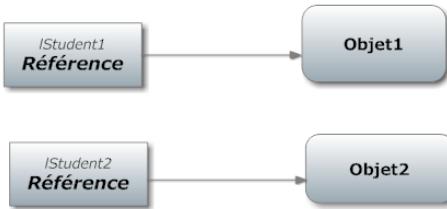


Figure 7

La troisième instruction (l'affectation) recopie dans ISStudent2 la référence contenue dans ISStudent1, ainsi on aboutit à la situation schématisée dans la figure ci-dessous :

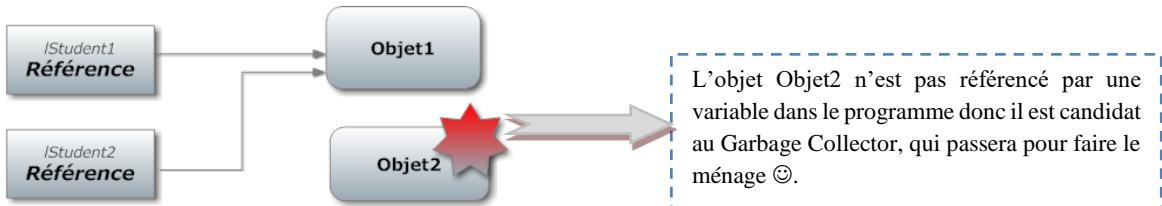


Figure 8

5. Comparaison des objets

Les opérateurs == et != permettent de comparer les références d'objets. Pour comparer les valeurs des objets il faudrait redéfinir et utiliser la méthode *equals*. Cette méthode sera traitée avec plus de détails dans le chapitre suivant.

Exemple 1: comparaison des références

On considère le programme ci-dessous :

```

public class ExempleProgramme1 {
    public static void main(String[] args) {
        String s1 = new String("Boudaa");
        String s2 = new String("Boudaa");
        if(s1==s2) System.out.println("OK")
        else System.out.println("KO");
    }
}
  
```

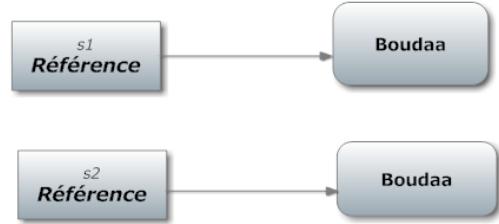


Figure 9

Ce programme affiche : KO. Ce qu'est toute à fait normal car s1 et s2 ne référence pas le même objet donc les références ne sont pas égales (cf. figure 4).

Exemple 2 : cas particulier des chaînes constantes

Considérons maintenant le code suivant :

```

public class ExempleProgramme2 {
    public static void main(String[] args) {
        String s1 = "Boudaa";
        String s2 = "Boudaa";
        String s3 = "Boudaa";
        if(s1==s2 && s2==s3) System.out.println("OK");
        else System.out.println("KO");
    }
}
  
```

Le programme affiche : OK. En effet pour des raisons d'optimisation dans le cas des chaînes constantes ayant même valeurs la machine virtuelle crée une seule instance mémoire (cf. figure 5).

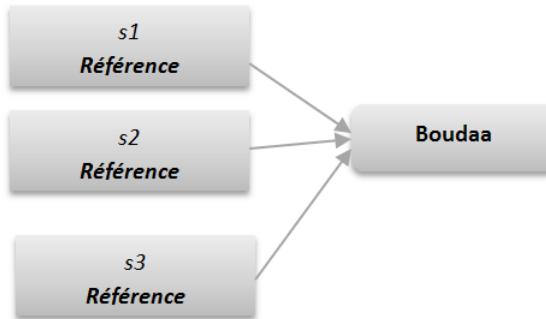


Figure 10

Bonne pratique :

Pour gagner en performance Il est donc claire qu'il faudrait instancier les chaînes constantes de la même manière que dans ExempleProgramme2 pour optimiser le nombre d'instances. Voici un exemple de ce qu'il ne faut pas faire :

```
String s = new String("ENSA AL-Hoceima");
```

Cette instruction crée une nouvelle instance inutile de la classe String à chaque exécution, si une telle instruction se trouve dans une boucle ou une partie de code fréquemment exécutée, un nombre important d'instances seront inutilement créées. Une version améliorée est tout simplement :

```
String s = " ENSA AL-Hoceima";
```

Exemple 3: comparaison du contenu des objets avec la méthode *equals*

La classe String redéfinie déjà la méthode *equals* on peut donc l'utiliser directement comme dans le programme suivant :

```
public class ExempleProgramme3{
    public static void main(String[] args) {
        String s1 = new String("Boudaa");
        String s2 = new String("Boudaa");
        if(s1.equals(s2)) System.out.println("OK");
        else System.out.println("KO");
    }
}
```

Le programme affiche OK, car la méthode *equals* compare les instances.

Exemple 4: Comparaison des objets *Etudiant*

On considère la classe *Etudiant* définie précédemment et le programme suivant :

```
public class ExempleProgramme {
    public static void main(String[] args) {
        Etudiant ls1 = new Etudiant("Baladi","Faiz","t",14);
        Etudiant ls2 = new Etudiant("Baladi","Faiz","t",14);
        if(ls1 == ls2) System.out.println("OK");
        else System.out.println("KO");
    }
}
```

Le programme affiche bien évidemment KO par ce qu'il compare les références de deux instances différentes. Pour comparer les valeurs il faut redéfinir la méthode *equals* pour la classe *Etudiant*. Nous reviendrons avec plus de détails sur ce sujet dans le chapitre suivant.

6. Portée des variables locales

La portée d'une variable locale définie à l'intérieur d'une méthode est limitée au bloc constituant la méthode où elle est déclarée. De plus, une variable locale ne doit pas posséder le même nom qu'un argument de la méthode :

```
void exempleMethode(int n)

{
    float x ;      // variable locale à f

    float n ;      // ERREUR : interdit en Java

    .....

}
```

L'emplacement mémoire d'une variable locale est alloué au moment où l'on entre dans la fonction et il est libéré lorsqu'on en sort.

Contrairement aux attributs d'une classe, les variables locales ne sont pas initialisées de façon implicite. Toute variable locale, y compris une référence à un objet, doit être initialisée avant d'être utilisée.

Les variables locales à une méthode sont en fait des variables locales au bloc constituant la méthode. En effet, on peut aussi définir des variables locales à un bloc. Dans ce cas, leur portée est tout naturellement limitée à ce bloc ; leur emplacement est alloué à l'entrée dans le bloc et il disparaît à la sortie. Il n'est pas permis qu'une variable locale porte le même nom qu'une variable locale d'un bloc englobant.

```
void f()
{ int n ;      // n est accessible de tout le bloc constituant f
  .....
  for (...)
  { int p ;      // p n'est connue que dans le bloc de for
    int n ;      // interdit : n existe déjà dans un bloc englobant
    .....
  }
  .....
  //bloc d'instruction « artificiel »
  { int p ;      // p n'est connue que dans ce bloc ; elle est allouée ici
    .....
  }              // et n'a aucun rapport avec la variable p ci-dessus
  .....
}
```

Notez qu'on peut créer artificiellement un bloc, indépendamment d'une quelconque instruction structurée comme if, for. C'est le cas du deuxième bloc interne à la fonction *f* ci-dessus. :

7. Principe d'encapsulation

7.1. Encapsulation et ses avantages :

Dès le début ce cours, nous déclarons toujours les attributs avec le modificateur d'accès *private* pour respecter le principe d'encapsulation. Ce paragraphe explique la notion d'encapsulation et son intérêt et pourquoi il est important d'une manière générale de masquer autant que possible les données de l'objet en définissant le niveau de visibilité adéquat. Pour illustration, prenons l'exemple de la classe *Note* définie ci-dessous :

```
public class Note {
    public double noteFinale;
    public boolean valide ;
}
```

Les attributs de cette classe ont été définis avec le modificateur d'accès *public*. Avec cette définition de la classe *Note*, aucun contrôle ne peut être fait sur les valeurs de ses attributs. Nous pouvons dans une autre classe affecter des valeurs quelconques aux attributs des objets *Note* :

```
public class Main {
    public static void main(String[] args) {
        Note note = new Note();
        note.noteFinale = -1; //valeur invalide mais possible
        note.valide = true; //valeur invalide mais possible
    }
}
```

En rendant les attributs *private* nous pouvons imposer une validation sur les valeurs à affecter aux attributs. Dans ce cas les autres classes ne pourront pas modifier directement les valeurs des attributs, l'affectation de la note ne peut se faire que via la méthode *public affecterNote*. Cette dernière contrôle les valeurs à affecter aux attributs en fonction des règles de gestion :

```
public class Note {

    private double noteFinale;
    private boolean valide;

    public void affecterNote(double noteFinale) {

        // On contrôle la valeur de la note qui doit être dans l'intervalle [0,20]
        if (noteFinale >= 0 && noteFinale <= 20) {
            this.noteFinale = noteFinale;

            //On contrôle la valeur de l'attribut "valide"
            if (this.noteFinale < 12) {
                this.valide = false;
            } else {
                this.valide = true;
            }
        } else {
            // On écrit ici le code indiquant une erreur
            System.out.println("La note doit être définie dans l'intervalle [0,20]");
        }
    }
}
```

Après avoir rendu les attributs *private* les autres classes n'ont plus accès directe à aux attributs des objets de type *Note*. Ainsi, le code suivant ne compile pas :

```
3 public class Main {
40     public static void main(String[] args) {
5
6         Note note = new Note();
7         note.noteFinale = -1;
8         note.valide = true;
9
10    }
11 }
```

L'affectation de la note doit passer par une méthode publique contrôlant les valeurs à donner aux attributs :

```
public class Main {  
    public static void main(String[] args) {  
  
        Note note = new Note();  
        // la note sera affecté en respectant les règles de gestion interne  
        //aux objets de type Note  
        note.affecterNote(19);  
  
    }  
}
```

Disons que nous décidons de modifier les règles de validation de telle sorte que la note de validation doit être supérieure à 10 au lieu de 12. L'implémentation à l'intérieur de l'objet doit être changée

```
if (this.noteFinale < 10) {  
    this.valide = false;  
} else {  
    this.valide = true;  
}
```

Mais le monde extérieur n'est pas affecté. La façon dont les méthodes sont appelées reste exactement la même toujours:

```
Note note = new Note();  
note.affecterNote(19);
```

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les méthodes proposées. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet. Ainsi, si l'on veut protéger des informations contre une modification inattendue, on doit se référer au principe d'encapsulation.

L'utilisateur d'une classe n'a pas forcément besoin de savoir de quelle manière sont structurées les données dans l'objet, soit leur implémentation. Ainsi, en interdisant à l'utilisateur de modifier directement les attributs, et en l'obligeant à utiliser les fonctions définies pour les modifier, on est capable de s'assurer de l'intégrité des données. On pourra par exemple s'assurer que le type des données fournies est conforme aux attentes, ou encore que les données se trouvent bien dans l'intervalle attendu.

7.2. Niveaux de visibilité des membres d'une classe :

L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque. Il existe quatre niveaux de visibilité pour un membre d'une classe (champ ou méthode) :

- **Visibilité privée** : indiquée avec le mot clé **private** dans ce cas le membre (méthode ou attribut) est accessible uniquement aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé
- **Visibilité publique** : indiquée avec le mot clé **public** dans ce cas le membre (méthode ou attribut) est accessible depuis l'intérieur de sa classe et de l'extérieur de sa classe. Il s'agit du niveau de protection le plus bas.

- **Visibilité par défaut** : si aucun modificateur de visibilité n'est indiqué, dans ce cas, l'accès au membre (méthode ou attribut) est limité aux classes du même paquetage (on parle d'accès de paquetage).
- **Visibilité protégée** : indiquée avec le mot clé ***protected*** dans ce cas le membre sera accessible depuis les classes du même package et les sous-classes (voir le chapitre de l'héritage et polymorphisme).

7.3. Mutateurs et accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées *private* à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Les accesseurs et les mutateurs sont des méthodes qui respectent certaines conventions et elles sont destinées pour répondre à ce besoin.

Un accesseur est une méthode publique qui donne l'accès en lecture à une variable d'instance privée.

Un mutateur est une méthode publique utilisée pour modifier ou définir la valeur d'un champ privé.

Par convention les accesseurs et les mutateurs doivent respecter des conventions de nommage. Pour un attribut nommé *nomVar* de type *typeVar* voici comment on peut écrire ces méthodes pour respecter les conventions :

Mutateur	<pre>public void setNomVar(typeVar nomVar){ // éventuellement d'autres instructions ... this.nomVar = ... ; }</pre>
Accesseur	<pre>public typeVar getNomVar(){ ... return ...; }</pre> <p>Remarque : Pour un attribut de type booléen, il est possible de faire commencer l'accesseur par <i>is</i> au lieu de <i>get</i>.</p> <pre>public boolean isNomVar(){ ... return ...; }</pre>

Exemple :

On reprend l'exemple précédent et on l'adapte pour respecter les conventions des mutateurs et des accesseurs.

```
public class Note {

    private double noteFinale;
    private boolean valide;

    //setters
```

```

public void setNoteFinale(double noteFinale) {

    // On contrôle la valeur de la note qui doit être dans l'intervalle[0,20]
    if (noteFinale >= 0 && noteFinale <= 20) {
        this.noteFinale = noteFinale;

    } else {
        // On écrit ici le code indiquant une erreur
        System.out.println("La note doit être définie dans l'intervalle [0,20]");
    }

}

//getters
public void setValide(boolean valide) {
    if ((this.noteFinale < 12 && valide) || (this.noteFinale >= 12 && !valide)) {
        System.out.println("valeur incorrecte");
    } else {
        this.valide = valide;
    }
}

public boolean isValide() {
    return valide;
}

public double getNoteFinale() {
    return noteFinale;
}
}

```

8. Champs et méthodes de classe

En utilisant le mot clé *static* on peut définir des champs partagés par toutes les instances d'une même classe, ce type de champs sont appelés champs de classe ou champs statiques. De même, on peut définir des méthodes de classe (ou *méthodes statiques*) qui peuvent être appelées indépendamment de toute instance. L'initialisation des champs statiques suit les deux étapes suivantes :

- i. l'initialisation par défaut,
- ii. l'initialisation explicite éventuelle.

Exemple 1 : Champ de classe

```

public class MyClass {

    public static int nbrInstance;

    public MyClass() {
        nbrInstance++;
    }

    public static void main(String[] args) {
        System.out.println("Nombre d'objets crée est : "+MyClass.nbrInstance);
        MyClass c1 = new MyClass();
        System.out.println("Nombre d'objets crée est : "+MyClass.nbrInstance);
        MyClass c2 = new MyClass();
        System.out.println("Nombre d'objets crée est : "+MyClass.nbrInstance);
    }
}

```

Le programme affiche :

```
Nombre d'objets crée est : 0
Nombre d'objets crée est : 1
Nombre d'objets crée est : 2
```

Le champ *nbrInstance* est partagé par toutes les instances de la classe *MyClass*. La figure suivante permet de schématiser la situation :

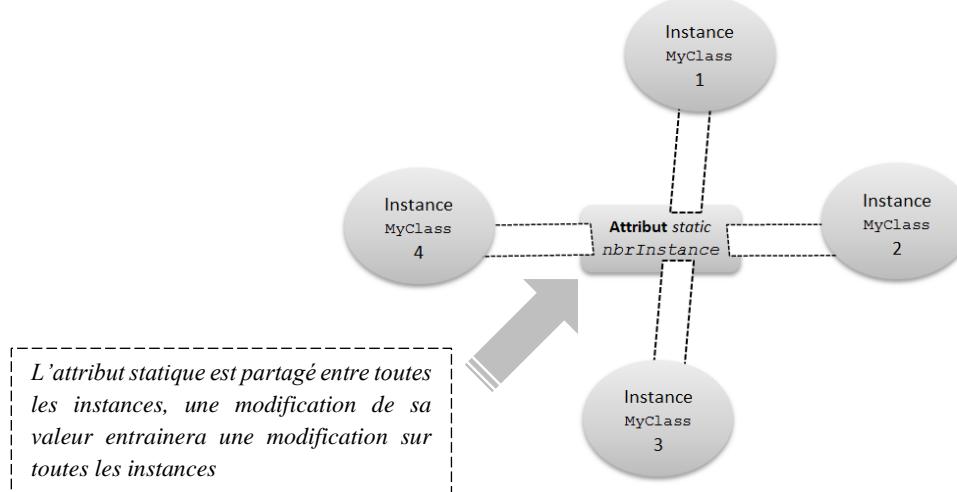


Figure 11

Exemple 2 : Méthode de classe :

On définit une classe *Complex* qui permet de présenter les nombres complexes

```
public class Complex {
    private double re;
    private double im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    public double getRe() { return re; }
    public double getIm() { return im; }
}
```

Et une classe *ComplexArithmetic* suivante :

```
public class ComplexArithmetic {
    public static Complex add(Complex z1, Complex z2) {
        return new Complex(z1.getRe() + z2.getRe(),
                           z1.getIm() + z2.getIm());
    }
}
```

Le programme ci-dessous illustre un exemple d'utilisation la méthode statique *add* :

```
public static void main(String[] args) {
    Complex z1 = new Complex(1.2, 2.2);
    Complex z2 = new Complex(3.2, 5.2);
    Complex z3 = ComplexArithmetic.add(z1, z2);
    System.out.println("z1+z2=" + z3.getRe() + "+" + z3.getIm() + "i");
}
```

Exemple 3 : Le Design Pattern SINGLETON

Les patrons de conception (*design patterns*) regroupent un ensemble de modèles de conception de classes pour répondre aux problèmes les plus courants rencontrés en programmation objet. Parmi les plus connus on peut citer les design patterns *singleton*, *factory*, *iterator* et *observer*.

L'objectif du patron de conception singleton est de garantir qu'une classe ne possède qu'une seule instance et de fournir un point d'accès global à celle-ci. On présente ci-dessous deux approches permettant d'implémenter un singleton, toutes les deux se basent sur un constructeur privé et sur un variable d'instance publique et statique qui présente l'unique instance de la classe.

Exemple avec la première approche : Le membre statique et public est un champ final

```
public class MySingleton {  
    public static final MySingleton INSTANCE = new MySingleton();  
    private MySingleton() {  
        // ... code du constructeur  
    }  
    // ... Reste de la classe  
}
```

Le constructeur privé n'est appelé qu'une seule fois pour initialiser le champ final statique et public MySingleton.INSTANCE l'absence de constructeur public garantit l'unicité des objets de la classe car on ne peut pas appeler ce constructeur de l'extérieur de la classe.

Exemple avec la deuxième approche : Dans la deuxième approche, une méthode statique remplace le champ final statique et public

```
public class MySingleton {  
    private static final MySingleton INSTANCE = new MySingleton();  
    private MySingleton() {  
        // ... code du constructeur  
    }  
    public static MySingleton getInstance() {  
        return INSTANCE;  
    }  
    // ... Reste de la classe  
}
```

Une autre implémentation de la deuxième approche :

```
public class MySingleton {  
    private static MySingleton INSTANCE;  
  
    private MySingleton() {  
        // ... code du constructeur  
    }  
    public static MySingleton getInstance() {  
        if (INSTANCE == null)  
            INSTANCE = new MySingleton();  
        return INSTANCE;  
    }  
    // ... Reste de la classe  
}
```

Questions :

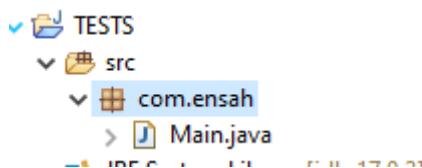
1. Quelle sont les avantages et les inconvénients e chaque approche ?
2. Est-ce que ces approches sont toujours adaptées à une application multithread ? Pourquoi ?

9. Notion de paquetage (package)

Un paquetage (package) est un ensemble de classes. En fait, c'est un ensemble de dossiers et de sous-dossiers contenant une ou plusieurs classes.

Les packages permettent de faire une organisation logique et physique des classes. Un paquetage portant le nom X.Y.Z doit se trouver intégralement sur le système de fichier du système d'exploitation dans un sous-répertoire de nom X/Y/Z. En revanche, s'il existe un paquetage de nom X.Y.U, il pourra se trouver dans un sous-répertoire X/Y/U.

La figure ci-dessous montre un exemple d'une classe définie à l'intérieur d'un package nommé *com.ensah*



10. Passage de paramètres à une méthode

Java emploie toujours un passage par valeur. Ainsi quand un objet est transféré comme paramètre d'une méthode la valeur transférée n'est pas une copie de l'objet mais une copie de la référence à l'objet, ainsi la méthode peut donc modifier l'objet correspondant.

La même règle s'applique à la valeur de retour d'une méthode.

Exemple 1 : passage en valeur cas des arguments de type primitifs

Une méthode ne peut pas modifier la valeur d'un argument d'un type primitif.

```
public class Exemple1 {
    public static void Echange(int a, int b)
    { System.out.println("Etat initiale : " + a + " " + b);
      int c;
      c = a;
      a = b;
      b = c;
      System.out.println("Etat finale : " + a + " " + b);
    }

    public static void main(String args[])
    {
        int n = 12, p = 40;
        System.out.println("avant appel : " + n + " " + p);
        Exemple1.Echange(n, p);
        System.out.println("apres appel : " + n + " " + p);
    }
}
```

Résultat d'exécution :

```
avant appel : 12 40
Etat initiale : 12 40
Etat finale : 40 12
apres appel : 12 40
```

Un échange a bien eu lieu ; mais il a porté sur les valeurs des arguments muets a et b de la méthode Échange. Les valeurs des arguments effectifs n et p de la méthode main n'ont pas été influencé par l'appel de la méthode Échange.

Exemple 2 : Cas des arguments de type objet

On test dans le programme ci-dessous le passage par valeur appliqué à une référence d'un objet :

```
public class Exemple2 {  
    public static void ChangeObject(Etudiant pS) {  
        System.out.println("Etat initiale CIN : " + pS.getCin());  
        pS.setCin("B121212");  
        System.out.println("Etat finale : " + pS.getCin());  
    }  
    public static void main(String args[]) {  
        Etudiant lS = new Etudiant ("A121211");  
        System.out.println("avant appel : " + lS.getCin());  
        Exemple2.ChangeObject(lS);  
        System.out.println("apres appel : " + lS.getCin());  
    }  
}
```

Résultat d'exécution :

```
avant appel : A121211  
Etat initiale CIN : A121211  
Etat finale : B121212  
apres appel : B121212
```

L'objet a été bien modifié.

Exemple 3 : valeur de retour

```
public class Etudiant {  
    // attributs  
    private String nom;  
    private String prenom;  
    private String cin;  
    private int age;  
    public Etudiant (String nom, String prenom, String cin, int age) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.cin = cin;  
        this.age = age;  
    }  
    public void displayStudentInfos() {  
        System.out.println("Nom :" + nom);  
        System.out.println("Prenom :" + prenom);  
        System.out.println("CIN :" + cin);  
        System.out.println("Age :" + age);  
    }  
    public static Etudiant incrementAge(Etudiant pSt){  
        pSt.age++;  
        return pSt;  
    }  
    public static void main(String[] args) {  
        Etudiant lStd = new Etudiant ("BOUDAA", "Mohamed", "A175232",12);  
        incrementAge(lStd);  
        lStd.displayStudentInfos();  
    }  
}
```

Dans le cas de la valeur de retour, on reçoit toujours la copie de la valeur fournie par une méthode. Une méthode peut aussi renvoyer un objet. Dans ce cas, elle fournit une copie de la référence à l'objet concerné.

11. Surcharge des méthodes (*Method Overloading*)

On parle de surcharge ou surdéfinition d'une méthode, quand elle est définie plusieurs fois dans une même classe, avec le même identificateur mais avec des paramètres de type différent, ou de même type mais dans un ordre différent. Cette règle s'applique également aux constructeurs.

La surcharge ne s'applique pas au type de retour seul. Ainsi il est interdit de créer deux méthodes qui ne diffèrent que dans le type de retour.

Exemple 1 : Surcharge du constructeur

Considérons cet exemple, dans lequel nous avons doté la classe Etudiant de 4 constructeurs :

```
public class Etudiant {
    // attributs
    private String nom;
    private String prenom;
    private String cin;
    private String age;

    public Etudiant () {
    }
    public Etudiant (String cin) {
        this.cin = cin;
    }
    public Etudiant (String nom, String prenom, String cin, String age) {
        this.nom = nom;
        this.prenom = prenom;
        this.cin = cin;
        this.age = age;
    }
    public Etudiant (String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
}
```

Exemple 2 : Surcharge de méthodes

On considère cette fois-ci un exemple extrait du code de la classe Math, la méthode abs possède 4 implémentations :

```
public static int abs(int a) {
    return (a < 0) ? -a : a;
}
public static long abs(long a) {
    return (a < 0) ? -a : a;
}
public static float abs(float a) {
    return (a <= 0.0F) ? 0.0F - a : a;
}
public static double abs(double a) {
    return (a <= 0.0D) ? 0.0D - a : a;
}
```

Exemple 3 : Surcharge de méthodes et cas d'ambiguïté

On considère la classe Point suivante :

```
public class Point {  
    private double x;  
    private double y;  
    public void deplace(double dx, float dy) {  
        x += dx;  
        y += dy;  
    }  
    public void deplace(float dx, double dy) {  
        x += dx;  
    }  
}
```

Considérons alors ces instructions :

```
public static void main(String[] args) {  
    Point a = new Point();  
    double d = 1.2;  
    float f = 2.3f;  
    a.deplace(d, f); // OK : appel de deplace (double, float)  
    a.deplace(f, d); // OK : appel de deplace (float, double)  
    a.deplace(d, d); // erreur : ambiguïté  
    a.deplace(f, f); // erreur : ambiguïté  
}
```

Les deux derniers appels seront refusés par le compilateur.

12. La récursivité des méthodes

Java autorise la récursivité des appels de méthodes. Celle-ci peut être directe lorsque une méthode comporte, dans sa définition, au moins un appel à elle-même ; et croisée lorsque l'appel d'une méthode entraîne l'appel d'une autre méthode qui, à son tour, appelle la méthode initiale (le cycle pouvant d'ailleurs faire intervenir plus de deux méthodes). La récursivité s'applique aux méthodes d'instance et aux méthodes statiques.

Exemple : calcul du factoriel récursivement

On peut appliquer la récursivité aussi bien aux méthodes usuelles qu'aux méthodes de classes (statiques). Voici un exemple classique d'une méthode statique calculant une factorielle de façon récursive :

```
public class Main {  
    public static int factoriel(int n) {  
        if (n == 0) {  
            return 1;  
        }  
        return n * factoriel(n - 1);  
    }  
    public static void main(String[] args) {  
        int fact = Main.factoriel(4);  
        System.out.println(fact);  
    }  
}
```

13. Le mot clé final

Le mot clé *final* empêche généralement la modification, ainsi :

- ✓ Si *final* est placé sur une classe : empêche l'héritage de la classe
- ✓ Si *final* est placé sur une méthode : empêche la redéfinition de la méthode
- ✓ Si *final* est placé devant un attribut : empêche sa modification. En plus, un attribut final doit être initialisé explicitement à sa déclaration ou dans tous les constructeurs.
- ✓ Si *final* est placé sur une variable : empêche la modification de la variable
- ✓ Si *final* est placé devant un paramètre fictif de la méthode : empêche la modification de la valeur de ce paramètre dans le corps de la méthode.

14. Classes enveloppes

Les classes enveloppes (*wrappers* en anglais) permettent de manipuler les types primitifs comme des objets. Il est à noter que ces classes sont finales (on ne peut pas hériter de ces classes). Les objets des classes enveloppes sont des objets immuables (toutes modifications de la valeur aboutissent à un nouvel objet) qui encapsule un type primitif. Grâce aux classes enveloppes on peut manipuler les types primitifs à l'aide de méthodes définies dans chaque type enveloppe. Ci-dessous la liste des types enveloppes intégrés en JAVA :

Type enveloppe	Type primitif associé
<i>Byte</i>	<i>byte</i>
<i>Short</i>	<i>short</i>
<i>Integer</i>	<i>int</i>
<i>Long</i>	<i>long</i>
<i>Float</i>	<i>float</i>
<i>Double</i>	<i>double</i>
<i>Character</i>	<i>char</i>
<i>Boolean</i>	<i>boolean</i>

14.1. Création des objets des classes enveloppes

Toutes les classes enveloppes disposent d'une méthode statique *valueOf(type_primitif)* qui retourne une instance représentant la valeur de type primitif spécifiée en paramètre. Bien que ces classes aient des constructeurs, il est déconseillé de les utiliser pour construire des objets de type enveloppe, c'est la méthode *valueOf* qu'est recommandée.

Exemple :

```
// Retourne une instance représentant la valeur entière de type primitif 25
Integer n = Integer.valueOf(25);

// Retourne une instance représentant la valeur réelle (double) de type primitif 0.25
Double d = Double.valueOf(0.25);

// Retourne une instance représentant la valeur réelle (float) de type primitif 0.25
Float f = Float.valueOf(0.25f);

// Retourne une instance représentant le caractère passé en paramètre
Character c = Character.valueOf('A');
```

Toutes les classes enveloppes disposent d'une méthode de la forme `xxxValue` (`xxx` représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif correspondant.

Exemple :

```
Integer intObj = Integer.valueOf(25);
int n = intObj.intValue();           // n contient 25

Double doubleObj = Double.valueOf(0.25);
double x = doubleObj.doubleValue(); // x contient 0.25
```

Remarque :

Pour comparer deux objet de type enveloppe on utilise la méthode `equals`. Il ne faut pas oublier que l'opérateur `==` appliqué à des objets se contente d'en comparer les adresses.

14.2. Emballage et déballage (*boxing/ unboxing*) automatique

Le JDK 5.0 a introduit des possibilités de conversions mises en place automatiquement par le compilateur ; on parle alors d'emballage ou de déballage automatique (*autoboxing* ou, encore *boxing* et *unboxing* en anglais). Ainsi, nous pouvons écrire les instructions suivantes :

```
Integer nObj = 25      // au lieu de : Integer.valueOf(25) ;
Double xObj= 0.25 ;    // au lieu de : Double.valueOf(0.25) ;
                      .....
int n = nObj ;        // au lieu de :      = nObj.intValue() ;
double x = xObj       // au lieu de :      = xObj.doubleValue() ;
```

Dans la première affectation, la valeur entière 12 est convertie en objet de type `Integer` et sa référence est affectée à `nObj`. De même, dans la troisième affectation (`n = nObj`), la valeur entière encapsulée dans `nObj` est affectée à `n`.

Dans des expressions arithmétiques, ces conversions s'ajoutent aux conversions implicites usuelles comme dans ces exemples où l'on suppose que `nObj1` et `nObj2` sont de type `Integer` :

Dans le cas suivant, `nObj2` est converti en `int` auquel on ajoute 2, le résultat est converti en `Integer` :

```
nObj1 = nObj2 + 2 ;
```

Dans le cas suivant `nObj1` est converti en `int`, auquel on ajoute 1 et le résultat est converti en `Integer` :

```
nObj1++ ;
```

On notera que de telles opérations arithmétiques, effectuées apparemment directement sur des types enveloppes, peuvent s'avérer relativement peu efficaces compte tenu des conversions supplémentaires qu'elles entraînent lors de l'exécution du code.

Ces conversions ont des limitations, en effet, elles ne sont possibles qu'entre un type enveloppe et son type primitif correspondant. Ainsi, cette instruction est incorrecte :

```
// 5, de type int, ne peut pas être converti en Double
Double xObj = 5 ;
```

De même, ceci est incorrect :

```
Integer nObj ;  
Double xObj = nObj ; // erreur de compilation
```

15. Les classes internes (*inner-class*)

Il existe 4 possibilités pour la définition des classes internes, dans cette section nous étudions 3 cas, le 4^{ème} cas qu'est celui des classes internes anonymes étudié dans le chapitre dédié à l'héritage et polymorphisme.

15.1. Classes définies à l'intérieur d'une autre classe

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe.

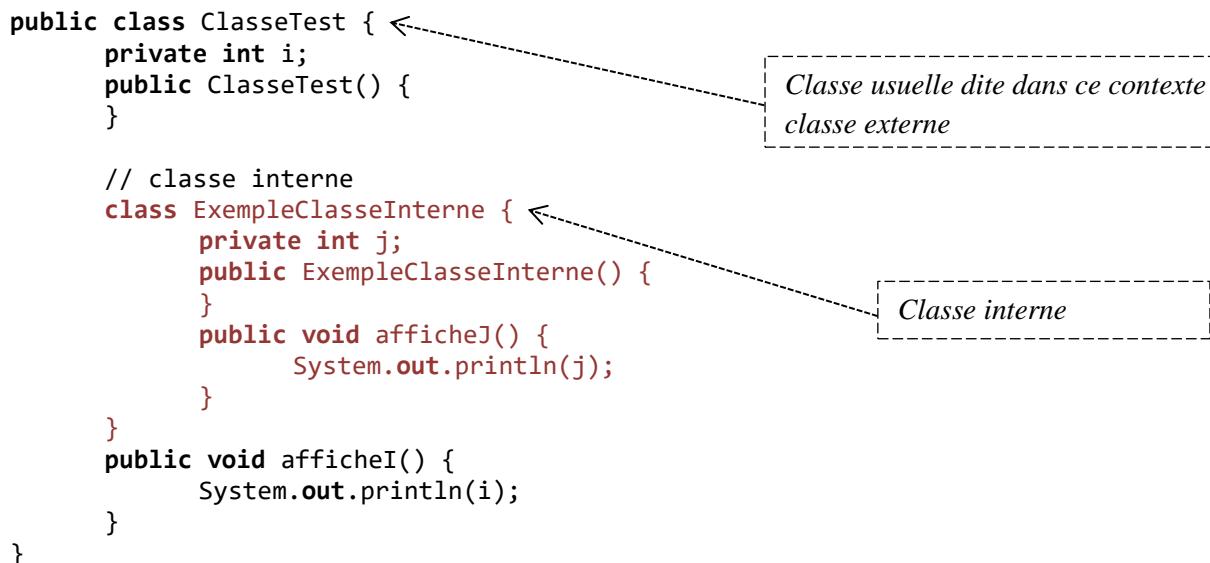
Ce type de classe interne est utilisé quand deux classes sont très liées l'une à l'autre au niveau de leur fonctionnement. Dans ce type de situation, le couplage entre les deux classes est renforcé pour faciliter la communication entre leurs membres.

Syntaxe :

```
[modifiers] class OuterClassName {  
    code...  
    [modifiers] class InnerClassName {  
        code....  
    }  
}
```

Exemple :

```
public class ClasseTest { <  
    private int i;  
    public ClasseTest() {  
    }  
  
    // classe interne  
    class ExempleClasseInterne { <  
        private int j;  
        public ExempleClasseInterne() {  
        }  
        public void afficheJ() {  
            System.out.println(j);  
        }  
    }  
    public void afficheI() {  
        System.out.println(i);  
    }  
}
```



Classe usuelle dite dans ce contexte classe externe

Classe interne

Ci-dessous un exemple d'instanciation et d'utilisation des objets de la classe interne à l'intérieur de classe :

```
public class ClasseTest {  
  
    private int i;  
  
    //déclaration d'un attribut de type classe interne
```

```

private ExempleClasseInterne exAttribut;

public ClasseTest() {

    //exemple d'instanciation d'une classe interne dans la classe externe
    ExempleClasseInterne ex = new ExempleClasseInterne();

    exAttribut =ex;
}

// classe interne
class ExempleClasseInterne {
    private int j;
    public ExempleClasseInterne() {
    }
    public void afficheJ() {
        System.out.println(j);
    }
    public void afficheI() {
        System.out.println(i);
    }
}

```

Règles appliquées aux classes internes :

- Les classes internes peuvent être définies avec l'un des modificateurs d'accès : *public, protected, private, ou default (package)*.
- Un objet d'une classe interne a toujours accès aux champs et méthodes (même privés) de l'objet externe lui ayant donné naissance.
- Un objet de classe externe a toujours accès aux champs et méthodes (même privés) d'un objet d'une classe interne auquel il a donné naissance.
- Une méthode statique n'est associée à aucun objet. Par conséquent, une méthode statique d'une classe externe ne peut créer aucun objet d'une classe interne.
- Une classe interne ne peut pas contenir de membres statiques
- Bien que ce soit une option peu utilisée, Java permet d'utiliser une classe interne depuis une classe indépendante (*non englobante*). Dans ce cas il faut rattacher un objet d'une classe interne à un objet de sa classe englobante avec l'utilisation d'une syntaxe particulière de *new* , soit la situation suivante:

```

class E{
    class I{ }
}

```

En dehors de *E*, on peut toujours déclarer une référence à un objet de type *I*, de cette manière :
E.I i ; // référence à un objet de type I (interne à E)

Mais la création d'un objet de type *I* ne peut se faire qu'en le rattachant à un objet de sa classe englobante.

Par exemple, si l'on dispose d'un objet *e* créé ainsi :

```
E e = new E();
```

on pourra affecter à *i* la référence à un objet de type *I*, rattaché à *e*, en utilisant *new* comme suit :

```
// création d'un objet de type I, rattaché à l'objet e, affectation de  
// sa référence à i  
i = new e.I() ;
```

15.2. Classes internes locales

On peut définir une classe interne I dans une méthode f d'une classe E. Dans ce cas, linstanciation d'objets de type I ne peut se faire que dans f. En plus des accès déjà décrits, un objet de type I a alors accès aux variables locales finales de f.

```
public class E  
{  
    ....  
    void f()  
    {  
        final int n=15 ;  
        float x ;  
        class I // classe interne à E, locale à f  
        { .... // ici, on a accès à n mais pas à x  
        }  
        I i = new I() ;  
    }  
}
```

15.3. Classes internes statiques

Les objets des classes internes dont nous avons parlé jusqu'ici étaient toujours associés à un objet d'une classe englobante. On peut créer des classes internes "autonomes" en employant lattribut static :

```
public class E // classe englobante  
{ ....  
    public static class I // définition (englobée dans celle de E)  
    { .... // d'une classe interne autonome  
    }  
}
```

Depuis lextérieur de E, on peut instancier un objet de classe I de cette façon :

```
E.I i = new E.I() ;
```

L'objet i n'est associé à aucun objet de type E. Dans ce cas, la classe I n'a plus accès aux membres de E, sauf s'il s'agit de membres statiques

15.4. Droits d'accès aux classes

Un fichier source pouvait contenir plusieurs classes, mais une seule pouvait avoir le modificateur d'accès *public*. Chaque classe dispose de ce qu'on nomme un droit d'accès (on dit aussi un modificateur d'accès). Il permet de décider quelles sont les autres classes qui peuvent l'utiliser. Il est simplement défini par la présence ou l'absence du mot-clé public :

- ✓ **Avec le mot-clé public**, la classe est accessible à toutes les autres classes
- ✓ **Sans le mot-clé public**, la classe n'est accessible qu'aux classes du même paquetage.

La classe contenant la méthode *main* doit être publique pour que la machine virtuelle y ait accès.

Cas des classes internes :

Pour les classes internes, compte tenu de l'imbrication de leur définition dans celle d'une autre classe :

- **Avec *public***, la classe interne est accessible partout où sa classe externe l'est ;
- **Avec *private*** (qui est utilisable avec une classe interne, alors qu'il ne l'est pas pour une classe externe), la classe interne n'est accessible que depuis sa classe externe ;
- **Sans aucun mot-clé**, la classe interne n'est accessible que depuis les classes du même paquetage.

16. Java et la ligne de commande

16.1. Arguments d'une méthode *main*

L'en-tête de la méthode *main* se présentait ainsi :

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

La méthode reçoit un argument du type tableau de chaînes de caractères destiné à contenir les éventuels arguments fournis au programme lors de son lancement dans la ligne de commande. Lorsque le programme est lancé à partir d'une ligne de commande, ces arguments sont indiqués à la suite de l'appel du programme.

Exemple :

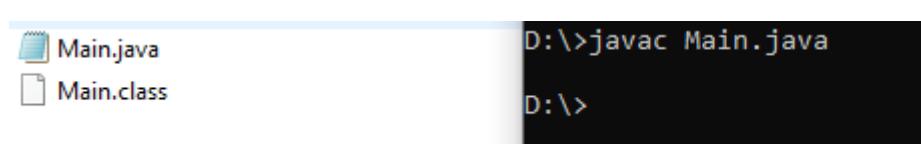
```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Concaténation de" +  
            args[0] + " et " + args[1] + " donne :" + args[0] + args[1]);  
    }  
}
```

Remarque : Les arguments sont toujours récupérés sous forme de chaînes de caractères. Si l'on souhaite récupérer des informations de type numérique, il faudra procéder à une conversion des données.

16.2. Compilation et exécution d'un programme en ligne de commande

Pour compiler une classe en ligne de commande on utilise la commande *javac NomClass.java*

Après la compilation on obtient le fichier .class associé. Voir figure ci-dessous :



Pour exécuter une classe Java on utilise la commande :

```
java nom_du_programme argument_1 argument_2 .... argument_n
```

```
D:\>javac Main.java
D:\>java Main "Boudaa" "Tarik"
Concaténation de Boudaa et Tarik donne :BoudaaTarik
D:\>
```

Résultat de l'exécution

Compilation

Exécution en passant au programme deux arguments qui seront récupérés dans le tableau args de la méthode main

16.3. Les archives jar

Un fichier JAR (*Java archive*) est un fichier ZIP utilisé pour distribuer un ensemble de classes Java constituant une application ou une bibliothèque de classes. Ce format est utilisé pour stocker les définitions des classes, ainsi que des métadonnées, constituant l'ensemble d'un programme.

Généralement la génération d'un jar se fait par l'IDE ou par un outil de *build* comme *Maven* ou *Gradle*. Ces outils seront découverts dans les TPs. Nous pouvons également utiliser la commande suivante.

```
jar cfv jar-file input-file(s)
```

Pour exécuter une application Java constituée d'un archive jar on utilise la commande suivante :

```
java -jar app.jar
```

17. Lecture des données au clavier avec la classe Scanner

Pour lire des données au clavier on peut utiliser la classe *Scanner* qui offre un ensemble de méthodes pour la lecture des données de différents types. Cette classe est définie dans le package *java.util*, donc pour l'utiliser il faut l'importer au début par :

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

Exemple 1 : Lecture d'une chaîne de caractères :

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Veuillez saisir un texte :");
        String str = sc.nextLine();
        System.out.println("Vous avez saisi : " + str);
    }
}
```

A l'exécution on obtient :

```
Veuillez saisir un texte :
Je suis Tarik BOUDAA
Vous avez saisi : Je suis Tarik BOUDAA
```

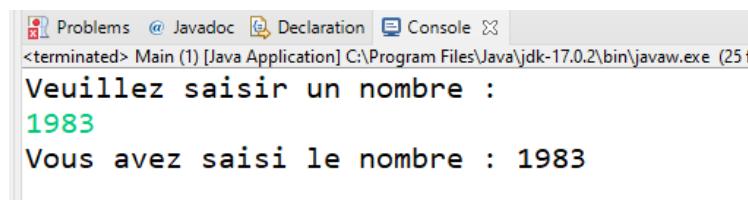
Exemple 2 : Lecture d'une valeur entière :

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Veuillez saisir un nombre :");
        int str = sc.nextInt();
        System.out.println("Vous avez saisi le nombre : " + str);

    }
}
```

A l'exécution on obtient :



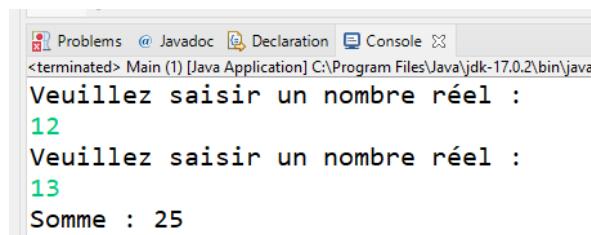
```
Problems @ Javadoc Declaration Console ✎
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe (25)
Veuillez saisir un nombre :
1983
Vous avez saisi le nombre : 1983
```

Exemple 3 : Lecture d'un réel :

Dans ce programme on lit deux nombres réels et on affiche leur somme:

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Veuillez saisir un nombre réel :");
        int r1 = sc.nextInt();
        System.out.println("Veuillez saisir un nombre réel :");
        int r2 = sc.nextInt();
        System.out.println("Somme : " + (r1+r2));
    }
}
```

A l'exécution on obtient :



```
Problems @ Javadoc Declaration Console ✎
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\java
Veuillez saisir un nombre réel :
12
Veuillez saisir un nombre réel :
13
Somme : 25
```

Exemple 3 : Lecture d'un caractère :

Il n'y a pas une méthode spéciale pour lire un caractère. Mais nous pouvons lire une chaîne de caractères et extraire le premier caractère avec la méthode *charAt(int index)* :

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Veuillez saisir un caractère :");
        String s = sc.nextLine();
        char c = s.charAt(0);
        System.out.println("Le caractère saisi est : " + c);

    }
}

```

Le tableau ci-dessous décrit les différentes méthodes de la classe Scanner.

Méthodes	Description
<i>String next()</i>	Renvoie le prochain token complet de l'analyseur
<i>byte nextByte()</i>	Renvoie un byte lu au clavier
<i>double nextDouble()</i>	Renvoie un double lu au clavier
<i>float nextFloat()</i>	Renvoie un float lu au clavier
<i>int nextInt()</i>	Renvoie un int lu au clavier
<i>long nextLong()</i>	Renvoie un long lu au clavier
<i>short nextShort()</i>	Renvoie un short lu au clavier
<i>boolean nextBoolean()</i>	Renvoie un boolean lu au clavier
<i>String nextLine()</i>	Renvoie une chaîne de caractère lu au clavier
<i>boolean hasNext()</i>	Retourne « vrai » si le scanner a une autre donnée dans son entrée
<i>boolean hasNextInt()</i>	Vérifie si la prochaine donnée dans l'entrée peut être interprétée comme un <i>int</i> en utilisant la méthode nextInt() ou non
<i>boolean hasNextFloat()</i>	Vérifie si la prochaine donnée dans l'entrée de ce scanner peut être interprétée comme un <i>float</i> en utilisant la méthode nextFloat() ou non
<i>boolean hasNextLine</i>	Vérifie s'il y a une autre ligne dans l'entrée de ce scanner ou non

Chapitre 3 : Tableaux, Listes et Enumérations

1. Caractéristique des tableaux JAVA

En informatique, un tableau est une structure de données de base qui est un ensemble d'éléments (des variables ou autres entités contenant des données) contiguës en mémoire, auquel on a accès à travers un numéro d'index (ou indice). Comme tous les langages Java permet de manipuler des tableaux, ces derniers sont considérés comme des objets. Les tableaux Java ont les caractéristiques suivantes :

- Les éléments d'un tableau sont tous du même type.
- Ces éléments peuvent être des données de type primitif ou des références sur des objets.
- La taille d'un tableau est déterminée lors de sa création et ne peut être modifiée par la suite. Chaque tableau a un champ public *length* contenant sa taille.
- La position d'un élément dans un tableau est déterminée avec un indice entier. L'indice du premier élément d'un tableau est toujours 0 et l'indice du dernier élément d'un tableau est le nombre d'éléments du tableau moins 1.
- La JVM vérifie à l'exécution que les indices utilisés pour accéder à un élément figurent bien dans l'intervalle autorisé et que le type d'un objet à stocker est compatible avec le type du tableau.
- Les tableaux à plusieurs dimensions en Java s'obtiennent par composition de tableaux.

2. Tableaux à une dimension

2.1. Déclaration et mémorisation d'un tableau

On peut déclarer un tableau avec l'une des syntaxes ci-dessous :

```
TypeElements[] NomTableau ;  
TypeElements NomTableau[] ;
```

Exemple : `int[] Tab1; float Tab2[]; int Tab3[];`

TypeElements peut être un type primitif ou une classe. *TypeElements []* peut être utilisé aussi comme type de paramètre ou comme type de retour d'une méthode renvoyant un tableau. Comme toute référence, une référence de type tableau peut être égale à null ou désigner un objet tableau.

Un tableau qui contient des valeurs de type primitif ; comme celui présenté sur la figure suivante ; stocke directement ces valeurs les unes après les autres.



Tableau dont les éléments sont des primitifs

Un tableau qui contient des objets, comme le tableau présenté ci-dessous, stocke les références sur ces objets.

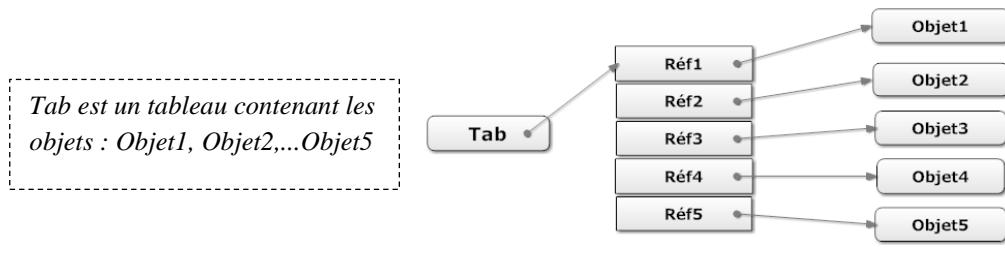


Tableau dont les éléments sont des objets

2.2. Crédation d'un tableau

Après avoir déclaré la variable tableau, il faut allouer les éléments de ce tableau. On utilise pour ce faire l'opérateur `new` en précisant le nombre d'éléments du tableau. Ces derniers sont initialisés automatiquement par des valeurs par défaut (0 pour les entiers, 0.0 pour les flottants, '\0' pour les caractères, false pour les booléen et null pour les objets).

On peut aussi utiliser un initialiseur au moment de la déclaration du tableau, comme on le fait pour une variable d'un type primitif.

Exemple 1 : Tableau dont les éléments sont des types primitifs

```
public class Exemple1 {
    public static void main(String[] args) {
        int n = 1, p = 4;
        //déclaration et création d'un tableau d'entiers
        int NomTableau1[] = new int[12];
        //déclaration et création d'un tableau de float
        float NomTableau2[] = new float[14];
        //Avec initialiseur
        int NomTableau3[] = {1, 2, 4, 5};
        int t[] = {1, n, n+p, 2*p, 12} ;
        NomTableau3 = new int[]{1, 2, 4, 5};
    }
}
```

Exemple 2 : Tableau dont les éléments sont des objets de type Etudiant

```
public class Exemple1 {
    public static void main(String[] args) {

        Etudiant[] tabStudent1 = new Etudiant [19];
        Etudiant tabStudent2[] = new Etudiant [19];
    }
}
```

2.3. Utilisation d'un tableau

Pour pouvoir utiliser un tableau, vous devez respecter les trois étapes suivantes :

- Déclarer une variable pour référencer le tableau.
- Créer un objet de type tableau en initialisant la référence à ce tableau.
- Utiliser et manipuler les éléments de ce tableau.

On désigne un élément particulier en plaçant entre crochets, à la suite du nom du tableau, une expression entière nommée indice indiquant sa position. Le premier élément correspond à l'indice 0.

Exemple 1: parcours des éléments d'un tableau d'entiers

```
public class ExempleTableau {
    public static void main(String[] args) {
        int[] tab1 = {1,3,9,2};
        System.out.println("Les éléments du tableau Tab1");
        for(int i=0;i<tab1.length;i++){
            System.out.println(tab1[i]);
        }
        int[] tab2 = new int[4];
        tab2[0]=12;
        tab2[1]=3;
        tab2[2]=4;
        tab2[3]=7;
        System.out.println("Les éléments du tableau Tab2");
        for(int i=0;i<tab2.length;i++){
            System.out.println(tab2[i]);
        }
    }
}
```

Résultat d'exécution :

```
Les éléments du tableau Tab1
1
3
9
2
Les éléments du tableau Tab2
12
3
4
7
```

Exemple 2: parcours des éléments d'un tableau d'objets

```
public class ExempleTabObjet {
    public static void main(String[] args) {
        Etudiant[] tabStudent = new Etudiant [3];
        tabStudent[0] = new Etudiant (12, "karimi", "kamal");
        tabStudent[1] = new Etudiant (13, "salimi", "ouiam");
        tabStudent[2] = new Etudiant (14, "laili", "sanae");
        System.out.println("La liste des étudiants");
        for (int i = 0; i < tabStudent.length; i++) {
            System.out.println(tabStudent[i].getNom());
        }
    }
}
```

Résultat d'exécution :

```
La liste des étudiants
karimi
salimi
laili
```

Exemple 3 : Incrémentation des éléments d'un tableau

```
public class ExempleTab2 {
    public static void main(String[] args) {
        int[] tab = { 1, 3, 0 };
        System.out.println("Affichage des éléments du tableau :");
        for (int i = 0; i < tab.length; i++) {
            System.out.println(tab[i]);
        }
        // Incrémentation des éléments du tableau
```

```

        for (int i = 0; i < tab.length; i++) {
            tab[i]++;
        }
        System.out.println("Affichage des éléments du tableau après
incrémantation :");
        for (int i = 0; i < tab.length; i++) {
            System.out.println(tab[i]);
        }
    }
}

```

Résultat d'exécution :

```

Affichage des éléments du tableau :
1
3
0
Affichage des éléments du tableau après incrémantation :
2
4
1

```

2.4. Affichage d'un tableau de caractères

Nous pouvons afficher les tableaux de caractères avec la méthode `System.out.println` car ces tableaux disposent d'une méthode `toString` bien redéfinie qui convertit le tableau en une chaîne de caractère. Ceci, n'est pas le cas pour les autres types de tableaux.

On considère le code suivant :

```

public class Main {
    public static void main(String[] args) {
        char[] tabChar = { 'E', 'N', 'S', 'A', 'H' };
        int[] tabInt = { 1, 2, 3, 4 };
        System.out.println(tabChar);
        System.out.println(tabInt);
    }
}

```

A l'exécution on obtient le résultat ci-dessous :

```

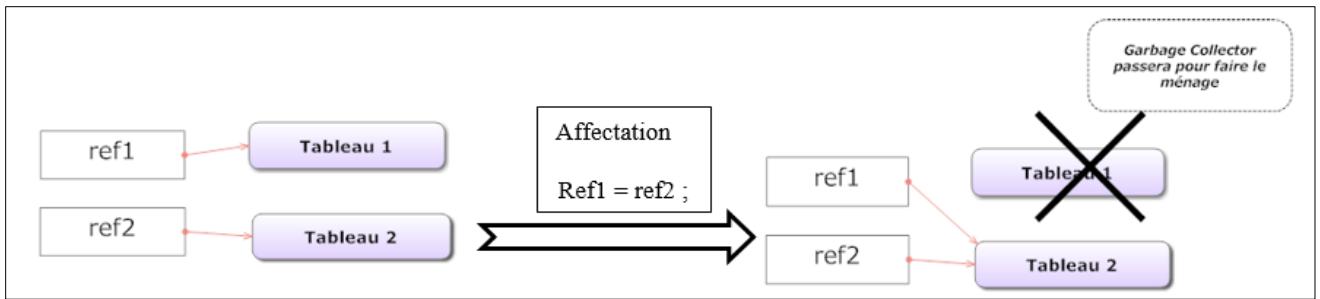
Problems @ Javadoc Declaration Console
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdl
[ENSAH
[I@36baf30c

```

Ce résultat s'explique par le fait que la méthode `toString` est bien redéfinie pour les tableaux de caractères. Elle ne l'est pas pour les autres tableaux, ce qui explique le résultat obtenu pour le tableau d'entiers.

2.5. Affectation de tableaux

Les tableaux sont des objets donc l'affectation entre tableaux est une affectation de référence. La figure ci-dessous schématise deux tableaux représentés par leurs références `ref1` et `ref2` après l'affectation de `ref2` à `ref1` (affectation entre références d'objets) le tableau 1 est devenue non référencé et il sera donc candidat au *Garbage Collector*. Les deux variables `ref1` et `ref2` contiennent la même référence pointant vers le tableau 2.



2.6. Utilisation de la boucle for...each

Pour parcourir un tableau on peut utiliser aussi la boucle for...each comme dans les exemples ci-dessous :

Exemple 1 : Parcours d'un tableau d'entiers :

```
public class ExempleAvecForEach {
    public static void main(String[] args) {
        int tab[]={12,13,16};
        for (int elem : tab){
            System.out.println(elem);
        }
    }
}
```

Exemple 2 : Parcours d'un tableau d'objets :

```
public class ExempleForEachObjet {
    public static void main(String[] args) {
        Student[] ls = new Student[3];
        ls[0] = new Student(12, "karimi", "majd");
        ls[1] = new Student(13, "salimi", "ouiam");
        ls[2] = new Student(14, "laili", "sanae");

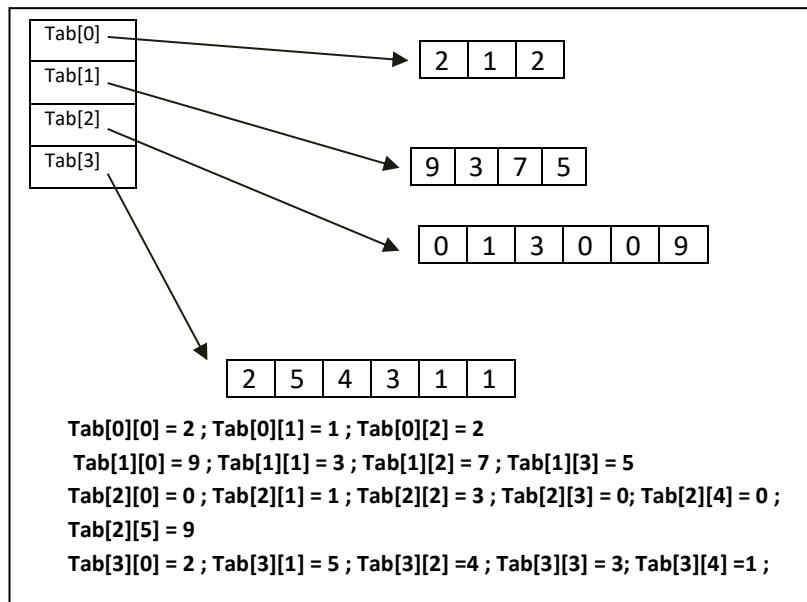
        for (Student itr : ls){
            System.out.println(itr.getNom()+" "+itr.getPrenom());
        }
    }
}
```

Remarque : La structure **for...each** ne s'applique qu'à des consultations de valeurs, et en aucun cas à des modifications.

3. Tableaux à deux dimensions

Les tableaux à plusieurs dimensions en Java s'obtiennent par composition de tableaux (création des tableaux dont les éléments sont eux-mêmes des tableaux) cette approche permet notamment de disposer de tableaux irréguliers, c'est-à-dire dans lesquels les différentes lignes pourront être de taille différente.

Ci-dessous un exemple d'un tableau irrégulier en Java :



3.1. Déclaration :

On peut déclarer un tableau à deux dimensions avec l'une des syntaxes ci-dessous :

```
TypeElements[][] NomTableau ;
TypeElements NomTableau[][] ;
TypeElements[] NomTableau[] ;
```

Exemple 1:

```
public class ExempleTabMultiDim {

    public static void main(String[] args) {
        int tab[][] = { new int[2], new int[4] };
        // L'initialisation des éléments par des valeurs par défaut (int ==>0)
        for (int i = 0; i < tab.length; i++) {
            for (int j = 0; j < tab[i].length; j++) {
                System.out.println("tab["+i+"][" +j+"] = " + tab[i][j]);
            }
            System.out.println("=====");
        }
    }
}
```

Résultat d'exécution :

```
tab[0][0]=0
tab[0][1]=0
=====
tab[1][0]=0
tab[1][1]=0
tab[1][2]=0
tab[1][3]=0
=====
```

Exemple 2:

```
public class ExempleTab2Dim {
    public static void main(String[] args) {
        int tab[][];
```

```

tab = new int[2][]; // création d'un tableau de 2 tableaux d'entiers
int[] tab1 = {1,2,3};
int[] tab2 = {9,7};
// on range les deux références tab1 et tab2 dans tab
tab[0] = tab1;
tab[1] = tab2;

// L'initialisation des éléments par des valeurs par défaut (int ==>0)
for (int i = 0; i < tab.length; i++) {
    for (int j = 0; j < tab[i].length; j++) {
        System.out.println("tab[" + i + "][" + j + "]=" + tab[i][j]);
    }
    System.out.println("=====");
}
}

```

Résultat d'exécution :

```

tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
=====
tab[1][0]=9
tab[1][1]=7
=====
```

Exemple 3:

```

public class ExempleTab2Dim {
public static void main(String[] args) {
    int tab[][] = { { 1, 2, 3 }, { 11, 12, 13, 16 } };
    // L'initialisation des éléments par des valeurs par défaut (int ==>0)
    for (int i = 0; i < tab.length; i++) {
        for (int j = 0; j < tab[i].length; j++) {
            System.out.println("tab[" + i + "][" + j + "]=" + tab[i][j]);
        }
        System.out.println("=====");
    }
}

```

Résultat d'exécution :

```

tab[0][0]=1
tab[0][1]=2
tab[0][2]=3
=====
tab[1][0]=11
tab[1][1]=12
tab[1][2]=13
tab[1][3]=16
=====
```

4. Quelques méthodes utiles pour la manipulation des tableaux

La classe `java.util.Arrays` contient un ensemble de méthodes de classe qui permettent d'effectuer des traitements sur les tableaux de type primitif ou de type objet :

4.1. Méthode `Arrays.toString` et `Arrays.deepToString`

La méthode `Arrays.toString` renvoie une représentation sous forme de chaîne du contenu du tableau spécifié en paramètre. La représentation sous forme de chaîne consiste en une liste d'éléments du tableau, entre crochets ("[]"). Les éléments adjacents sont séparés par les caractères ", " (une virgule suivie d'un espace).

La méthode `Arrays.deepToString` renvoie une représentation sous forme de chaîne du "contenu profond" du tableau spécifié. Si le tableau contient d'autres tableaux comme éléments, la représentation sous forme de chaîne contient leur contenu et ainsi de suite. Cette méthode est conçue pour convertir des tableaux multidimensionnels en chaînes.

Exemple :

```
import java.util.Arrays;

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

        //Tableau à une dimension
        int[] ar1 = {21,22,23,44};
        System.out.println(Arrays.toString(ar1));
        //Tableau à deux dimensions
        int[][] ar2 = {{1,2,3},{11,12},{21,22,23,44}};
        System.out.println(Arrays.deepToString(ar2));
        //Tableau à 3 dimensions
        int[][][] ar3 = {{{1,2,3}, {1,2,3}},{11,12},{44,55}},{{99,10},{2,8}}};
        System.out.println(Arrays.deepToString(ar3));

    }
}
```

A l'exécution on obtient :

```
[21, 22, 23, 44]
[[1, 2, 3], [11, 12], [21, 22, 23, 44]]
 [[[1, 2, 3], [1, 2, 3]], [[11, 12], [44, 55]], [[99, 10], [2, 8]]]
```

4.2. Méthode `Arrays.equals`

La méthode `equals` compare les éléments de deux tableaux. Renvoie true si les tableaux ont la même longueur et les éléments d'indices correspondants possèdent la même valeur

Exemple :

```
import java.util.Arrays;

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

        String[] tab1 = {"ENSAH","AL HOCEIMA","SIDI BOUAFIF"};
        String[] tab2 = {"ENSAH","AL HOCEIMA","SIDI BOUAFIF"};
        //Premier élément en minuscule
        String[] tab3 = {"ensah","AL HOCEIMA","SIDI BOUAFIF"};
        String[] tab4 = {"ensah","AL HOCEIMA"};
        System.out.println(Arrays.equals(tab1, tab2));
        System.out.println(Arrays.equals(tab1, tab3));
        System.out.println(Arrays.equals(tab1, tab4));

    }
}
```

}

A l'exécution on obtient :

```
true  
false  
false
```

4.3. Méthode *Arrays.fill*

La méthode *fill* remplit tout ou partie d'un tableau avec une valeur donnée.

Exemple 1 :

```
import java.util.Arrays;  
  
public class Main {  
    public static void main(String[] args) {  
  
        int[] ar = { 2, 2, 1, 8, 3, 2, 2, 4, 2 };  
        // remplir le tableau avec la valeur 10  
        Arrays.fill(ar, 10);  
        System.out.println("Affichage du tableau: \n" + Arrays.toString(ar));  
  
        // Création d'un tableau vide de 10 éléments  
        ar = new int[10];  
        // remplir le tableau avec la valeur 12  
        Arrays.fill(ar, 12);  
        System.out.println("Affichage du tableau: \n" + Arrays.toString(ar));  
  
        // Remplir à partir de l'indice 1 à l'indice 4 avec la valeur 15  
        // les autres valeurs de 12 ayant les indices 0 et de 5 à 10 seront maintenues  
        Arrays.fill(ar, 1, 5, 15);  
        System.out.println("Affichage du tableau: \n" + Arrays.toString(ar));  
  
    }  
}
```

A l'exécution on obtient :

```
Affichage du tableau:  
[10, 10, 10, 10, 10, 10, 10, 10, 10]  
Affichage du tableau:  
[12, 12, 12, 12, 12, 12, 12, 12, 12]  
Affichage du tableau:  
[12, 15, 15, 15, 15, 12, 12, 12, 12]
```

Exemple 2 :

On peut remplir un tableau multidimensionnel.

```
import java.util.Arrays;  
  
public class Main {  
    public static void main(String[] args) {  
        int[][] ar = new int[3][4];  
        // remplir toutes les lignes avec la valeur 10  
        for (int[] row : ar)  
            Arrays.fill(row, 10);
```

```
        System.out.println(Arrays.deepToString(ar));  
    }  
}
```

A l'exécution on obtient :

```
[[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]]
```

4.4. Méthode `Arrays.sort`

La méthode `sort` trie les éléments d'un tableau dans l'ordre croissant. Cette méthode est fournie en plusieurs versions (surcharge) pour supporter différents type de tableaux. Pour trier un tableau dans l'ordre décroissant on peut inverser l'ordre d'un tableau avec la méthode `reverseOrder`:

Exemple :

```
import java.util.Arrays;  
import java.util.Collections;  
  
public class Main {  
    public static void main(String[] args) {  
  
        // Tableaux de double non trié  
        double[] dArr = { 3.2, 1.2, 9.7, 6.2, 4.5 };  
  
        System.out.println("Avant le tri :" + Arrays.toString(dArr));  
        // Trier le tableau dans l'ordre croissant  
        Arrays.sort(dArr);  
        // Après le tri  
        System.out.println("Après le tri :" + Arrays.toString(dArr));  
  
        // Tableaux de double non trié  
        Double[] dEArr = { 3.2, 1.2, 9.7, 6.2, 4.5 };  
        System.out.println("Avant le tri :" + Arrays.toString(dEArr));  
        // Trier le tableau dans l'ordre décroissant  
        Arrays.sort(dEArr, Collections.reverseOrder());  
        // Après le tri  
        System.out.println("Après le tri :" + Arrays.toString(dEArr));  
  
        // Tableaux de char non trié  
        char[] cArr = { 'x', 'a', 'e', 'b' };  
        System.out.println("Avant le tri :" + Arrays.toString(cArr));  
        // Trier le tableau dans l'ordre croissant  
        Arrays.sort(cArr);  
        // Après le tri  
        System.out.println("Après le tri :" + Arrays.toString(cArr));  
  
        // Tableaux de String non trié  
        String[] sArr = { "AED", "AEA", "AAA", "ABCD" };  
        System.out.println("Avant le tri :" + Arrays.toString(sArr));  
        // Trier le tableau dans l'ordre croissant  
        Arrays.sort(sArr);  
        // Après le tri  
        System.out.println("Après le tri :" + Arrays.toString(sArr));  
    }  
}
```

```
    }  
}
```

A l'exécution on obtient :

```
Avant le tri :[3.2, 1.2, 9.7, 6.2, 4.5]  
Après le tri : [1.2, 3.2, 4.5, 6.2, 9.7]  
Avant le tri :[3.2, 1.2, 9.7, 6.2, 4.5]  
Après le tri : [9.7, 6.2, 4.5, 3.2, 1.2]  
Avant le tri :[x, a, e, b]  
Après le tri : [a, b, e, x]  
Avant le tri :[AED, AEA, AAA, ABCD]  
Après le tri : [AAA, ABCD, AEA, AED]
```

4.5. Méthode *Arrays.binarySearch*

La méthode *binarySearch* renvoie l'indice du premier élément égal à une valeur dans un tableau trié. Cette méthode est fournie en plusieurs versions (surcharge) pour supporter différents type de tableaux

Exemple :

```
import java.util.Arrays;  
  
public class Main {  
    public static void main(String[] args) {  
  
        // Tableau trié  
        double[] doubleArr = { 1.1, 2.2, 3.2, 3.21, 3.3, 15.1, 16.0, 20.11, 21.28 };  
  
        // Valeur à rechercher  
        double searchVal = 3.21;  
  
        int retVal = Arrays.binarySearch(doubleArr, searchVal);  
  
        System.out.println("Indice de la valeur recherchée : " + retVal);  
  
        // Tableau trié  
        String[] stringArr = { "AAA", "ABCD", "AEA", "AED", "BAED", "BBED" };  
  
        // Valeur à rechercher  
        String stringVal = "BAED";  
  
        retVal = Arrays.binarySearch(stringArr, stringVal);  
  
        System.out.println("Indice de la valeur recherchée : " + retVal);  
    }  
}
```

A l'exécution on obtient :

```
Indice de la valeur recherchée : 3  
Indice de la valeur recherchée : 4
```

4.6. Méthode *System.arraycopy*

La méthode statique *arraycopy* de la classe *java.lang.System* permet de copier des éléments d'un tableau dans un autre.

Signature de la méthode :

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Paramètres

- **src** : Le tableau source
- **srcPos** : Il s'agit de la position de départ dans le tableau source.
- **dest** : C'est le tableau de destination.
- **destPos** : Il s'agit de la position de départ dans la destination.
- **length** : C'est le nombre d'éléments à copier du tableau.

```
public class Main {  
    public static void main(String[] args) {  
        int arr1[] = { 0, 1, 2, 3, 4, 5 };  
        int arr2[] = { 5, 10, 20, 30, 40, 50 };  
  
        // copie un tableau à partir du tableau source spécifié  
        System.arraycopy(arr1, 2, arr2, 0, 3);  
        System.out.println(Arrays.toString(arr2));  
    }  
}
```

A l'exécution on obtient :

```
[2, 3, 4, 30, 40, 50]
```

4.7. Autres méthodes sur les tableaux

Pour plus de méthodes utilitaires sur les tableaux vous pouvez consulter la documentation officielle :

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>

5. Les listes ArrayList et LinkedList

Les tableaux Java sont simples d'utilisation mais ont certaines limitations gênantes dans certains cas :

- Les tableaux ne sont pas redimensionnables.
- L'insertion d'un élément au milieu d'un tableau oblige à déplacer tous les éléments qui suivent.
- La recherche d'un élément dans un grand tableau est longue s'il n'est pas trié.

Le Framework Collection contient plusieurs classes de collections utilisées pour gérer un ensemble d'éléments de type objet et résoudre les limitations inhérentes aux tableaux. Chaque classe est prévue pour gérer un ensemble avec des caractéristiques différentes. Le présent chapitre se limite à l'étude des types listes *ArrayList* et *LinkedList*, les autres types seront étudiés dans un autre chapitre plus détaillé sur le Framework Collection.

Ces deux classes gèrent des ensembles ordonnés d'éléments accessibles par leur indice. La classe *ArrayList* mémorise ses éléments dans un tableau Java. Si ce tableau interne est trop petit lors de l'ajout d'un nouvel élément à la collection, il est automatiquement remplacé par un nouveau tableau, plus grand, initialisé avec les références de l'ancien tableau. La classe *LinkedList* mémorise ses éléments avec une liste doublement chaînée, ce qui permet d'insérer plus rapidement un élément dans la collection, mais ralentit l'accès à un élément par son indice. Les sous-sections ci-dessous présentent les principales méthodes des classes *ArrayList* et *LinkedList*.

5.1. Méthodes propres à ArrayList:

<code>ArrayList<T>()</code>	Construit un tableau vide avec une spécification de capacité ou pas.
<code>void ensureCapacity(int taille)</code>	Vérifie que le tableau a la capacité nécessaire pour contenir le nombre d'éléments donné, sans nécessiter la réallocation de son tableau de stockage interne.
<code>void trimToSize()</code>	Réduit la capacité de stockage du tableau à sa taille actuelle.
<code>boolean isEmpty()</code>	Détermine si le tableau est vide ou pas.

Pour plus d'informations vous pouvez consulter le lien :

<https://docs.oracle.com/javase/10/docs/api/java/util/List.html>

5.2. Méthodes propres à LinkedList:

<code>LinkedList<T>()</code>	Construit une liste vide.
<code>void addFirst(T élément)</code> <code>void addLast(T élément)</code>	Ajoute un élément au début ou à la fin d'une liste.
<code>T getFirst()</code> <code>T getLast()</code>	Renvoie l'élément situé au début ou à la fin d'une liste.
<code>T removeFirst()</code> <code>T removeLast()</code>	Supprime et renvoie l'élément situé au début ou à la fin d'une liste.

Pour plus d'informations vous pouvez consulter le lien :

<https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html>

5.3. Méthodes communes à ArrayList et LinkedList :

<code>boolean add(T élément)</code> <code>void add(int indice, T élément)</code>	Ajoute un élément soit à un indice donné soit en fin de collection.
<code>void clear()</code> <code>void remove(int indice)</code> <code>void remove(T élément)</code>	Suppression de tout ou partie des éléments de la collection.
<code>T get(int indice)</code> <code>void set(int indice, T élément)</code>	Interrogation et modification d'un élément de la collection à un indice donné.
<code>boolean contains(T élément)</code> <code>int indexOf(T élément)</code> <code>int lastIndexOf(T élément)</code>	Recherche d'un élément dans la collection, en utilisant la méthode equals() pour comparer les objets.
<code>int size()</code>	Taille de la collection.
<code>Object[] toArray()</code>	Récupération des éléments de la collection dans un tableau
<code>boolean isEmpty()</code>	Détermine si le tableau est vide ou pas.

Pour plus d'informations vous pouvez consulter le lien :

<https://docs.oracle.com/javase/10/docs/api/java/util/ArrayList.html>

5.4. Exemples avec les listes :

Exemple 1 :

```
package com.ensah;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MainTest {
```

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

    // Construire la liste
    List<Etudiant> list = new ArrayList<>();

    //Construction des objets
    Etudiant etd1 = new Etudiant("Test1", "Test1");
    Etudiant etd2 = new Etudiant("Test2", "Test2");
    Etudiant etd3 = new Etudiant("Test3", "Test3");

    //Ajout des données objets dans la liste
    list.add(etd1);
    list.add(etd2);
    list.add(etd3);

}

}
```

Exemple 2 :

```
package com.ensah;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MainTest {

    public static void main(String[] args) {

        // Construire la liste
        List<Etudiant> list = new ArrayList<>();

        // Construction des objets
        Etudiant etd1 = new Etudiant("Test1", "Test1");
        Etudiant etd2 = new Etudiant("Test2", "Test2");
        Etudiant etd3 = new Etudiant("Test3", "Test3");

        // Ajout des données objets dans la liste
        list.add(etd1);
        list.add(etd2);
        list.add(etd3);

        // Tester que la liste est non vide
        if (!list.isEmpty()) {
            System.out.println("La liste n'est pas vide");
        } else {
            System.out.println("La liste est vide");
        }

        //suppresion tous les éléments
        list.clear();
    }
}
```

```
        if (!list.isEmpty()) {
            System.out.println("La liste n'est pas vide");
        } else {
            System.out.println("La liste est vide");
        }
    }

}
```

Exemple 3 :

```
package com.ensah;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MainTest {

    public static void main(String[] args) {

        // Construire la liste
        List<Etudiant> list = new ArrayList<>();

        // Construction des objets
        Etudiant etd1 = new Etudiant("Test1", "Test1");
        Etudiant etd2 = new Etudiant("Test2", "Test2");
        Etudiant etd3 = new Etudiant("Test3", "Test3");

        // Ajout des données objets dans la liste
        list.add(etd1);
        list.add(etd2);
        list.add(etd3);

        // Tester que la liste est non vide
        if (!list.isEmpty()) {
            System.out.println("La liste n'est pas vide");
        } else {
            System.out.println("La liste est vide");
        }

        // suppression tous les éléments
        list.clear();

        if (!list.isEmpty()) {
            System.out.println("La liste n'est pas vide");
        } else {
            System.out.println("La liste est vide");
        }

        list.add(etd1);
        list.add(etd2);

        // Parcourir la liste
        System.out.println("***** Avec for *****");
    }
}
```

```
        for(int i = 0; i< list.size() ; i++) {
            System.out.println(list.get(i).getNom() + " " +
list.get(i).getPrenom());
        }

        System.out.println("***** Avec for each *****");
//Avec foreach
        for(Etudiant it : list) {
            System.out.println(it.getNom() + " " + it.getPrenom());
        }
    }
}
```

Exemple 4 :

```
package com.ensah;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MainTest {

    public static void main(String[] args) {

        // Construire la liste
        List<Etudiant> list = new ArrayList<>();

        // Construction des objets
        Etudiant etd1 = new Etudiant("Test1",
"Test1");
        Etudiant etd2 = new Etudiant("Test2",
"Test2");
        Etudiant etd3 = new Etudiant("Test3",
"Test3");

        // Ajout des données objets dans la liste
        list.add(etd1);
    }
}
```

```
list.add(etd2);
list.add(etd3);

// Tester que la liste est non vide
if (!list.isEmpty()) {
    System.out.println("La liste n'est pas
vide");
} else {
    System.out.println("La liste est
vide");
}

// suppression tous les éléments
list.clear();

if (!list.isEmpty()) {
    System.out.println("La liste n'est pas
vide");
} else {
    System.out.println("La liste est
vide");
}

list.add(etd1);
list.add(etd2);
list.add(etd3);

// Parcourir la liste
System.out.println("**** Avec for ****");
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i).getNom()
+ " " + list.get(i).getPrenom());
}

System.out.println("**** Avec for each
****");
// Avec foreach
```

```
for (Etudiant it : list) {
    System.out.println(it.getNom() + " " +
it.getPrenom());
}

list.remove(1);
System.out.println("**** Après suppression
****");

for (Etudiant it : list) {
    System.out.println(it.getNom() + " " +
it.getPrenom());
}

System.out.println("**** Après remplacement
****");

Etudiant etd4 = new Etudiant("test4",
"test4");
list.set(1, etd4);

for (Etudiant it : list) {
    System.out.println(it.getNom() + " " +
it.getPrenom());
}

}
```

5.5. Choix de la structure de données à utiliser :

Le choix d'une structure de données sera étudier en détails dans un autre chapitre approfondie sur les collections, cependant, nous pouvons donner quelques indications ici concernant les types listes. Il est recommandé de choisir le type de classe « liste » le plus adapté en fonction de l'algorithme.

Il est souhaitable d'effectuer les choix suivants :

- i. Si la taille du tableau est fixe, on utilise un tableau simple (array).

- ii. Si la taille n'est pas fixe la classe *ArrayList* est plus adaptée. Cette dernière fournit en effet les meilleures performances d'accès aux données.
- iii. Dans le cas particulier où l'algorithme nécessite un grand nombre d'insertions et de suppressions, la classe *LinkedList* pourra être employée.

Remarques importantes :

1. Il y a des classes dans le JDK présentent que pour des raisons historiques, par exemple il faut éviter l'utilisation de la classe *Vector* qui fournit les mêmes services que la classe *ArrayList* mais avec des performances amoindries.
2. Les classes *ArrayList* et *LinkedList* ne sont pas *thread-safe*. Autrement dit leurs méthodes sont pas conçues pour qu'elles soient appelées simultanément par plusieurs threads. Pour les applications nécessitant l'accès simultané par plusieurs Thread, il existe d'autres classes que nous allons étudier dans un autre chapitre.

6. Tableau en argument ou en retour d'une méthode

En Java les tableaux sont des objets, ainsi lorsqu'on transmet un nom de tableau en argument d'une méthode, on transmet en fait la référence au tableau. La méthode agit alors directement sur le tableau concerné et non sur une copie. Les mêmes réflexions s'appliquent à un tableau fourni en valeur de retour d'une méthode. Ainsi, Java permet beaucoup de flexibilité par rapport à C++ pour lequel, un tableau n'est pas un objet, ce qui fait, la transmission d'un tableau en argument correspond à son adresse ; elle est généralement accompagnée d'un second argument pour préciser la taille.

7. Méthode avec nombre d'arguments variable, notation ellipse

Depuis la version 5, Java permet de définir une méthode dont le nombre d'arguments est variable.

Par exemple, si l'on définit une méthode *add* avec l'en-tête suivant :

int add (int... values)

On pourra l'appeler avec un ou plusieurs arguments de type *int* ou même sans argument. Voici quelques appels corrects de *add* :

```
int n, p, q ;  
....  
add () ; // aucun argument  
add (n) ; // un argument de type int  
add (n, p) ; // deux arguments de type int  
add (n, p, q) ; // trois arguments de type int
```

Les trois points de cette syntaxe se nomment « *ellipsis notation* » (en français peut être traduite par notation "ellipse").

Dans le corps de la méthode, on accédera aux différents arguments représentés par l'ellipse comme si l'on avait utilisé l'en-tête ***add (int[] values)***.

Mais il n'y a pas d'équivalence absolue entre les deux notations tableau et ellipse. En effet, avec l'en-tête : ***f (int... values)*** on peut effectivement appeler *f* avec un tableau d'entiers ou avec zéro ou plusieurs arguments de type *int*. En revanche, avec l'en-tête : ***f (int[] values)*** on ne peut appeler *f* qu'avec un tableau d'entiers.

Exemple :

Voici un exemple de méthode *add* renvoyant la valeur de la somme des différents entiers passés en argument :

```
static int add (int ... values)
{
    int sum = 0 ;
    for (int i=0 ; i< values.length ; i++) {
        sum += values [i] ;
    }
    return sum ;
}
```

Règles d'utilisation de la notation ellipse :

1. L'ellipse peut être accompagnée d'autres arguments classiques, à condition de figurer en fin de liste et d'être unique. Exemples :

<code>void f (int n, float x, Point p, double... v)</code>	Utilisation <u>correcte</u>
<code>void f (int n, double... v, float x, Point p)</code>	Utilisation <u>incorrecte</u> : ellipse n'est pas en fin
<code>void f (int... vi, double... vd)</code>	Utilisation <u>incorrecte</u> : ellipse utilisée plusieurs fois

2. même si les notations tableau et ellipse ne sont pas totalement équivalentes, il n'est pas possible de définir deux méthodes utilisant l'une l'ellipse, l'autre un tableau comme dans :

```
myMethod (int... pValues)
myMethod (int[] pValues)
```

Par contre il est tout à fait possible d'effectuer la surdéfinition suivante :

```
myMethod (int... pValues)
myMethod (int n, int p)
```

3. Une méthode utilisant l'ellipse « *double...* », on peut l'appeler avec des arguments de type *double*, *double[]* et même *int* puisqu'alors Java mettra en œuvre une conversion implicite de *int* en *double*. Par contre, pour une méthode avec un argument de type *double[]*, on ne pourra plus utiliser d'arguments de type *int*, ni même de type *int[]*.
4. **Règle de recherche d'une méthode lors de la surdéfinition** : Java effectue tout d'abord une recherche, sans tenir compte des méthodes à ellipse. Si la recherche précédente n'a pas abouti (et seulement dans ce cas-là), on effectue une nouvelle recherche en faisant intervenir les méthodes à ellipse.

Exemple1 :

```
static void f (int n, int p) { ..... }
static void f (int... vi) { ..... }

.....
int i1, i2, i3 ;
f(i1, i2) ; ➔ appel de f(int, int)
f(i1) ; ➔ appel de f(int...)
f(i1, i2, i3) ; ➔ appel de f(int...)
```

Exemple2 :

```
static void f (int... vi) { ..... }
static void f (double v1, double v2) { ..... }

.....
int i1, i2 ;
f(i1, i2) ; ➔ appel de f(double, double)
```

La recherche sans ellipse conduit au choix de *f(double, double)* donc c'est elle qui sera appelée.

8. Les énumérations

8.1. Définition d'un type énuméré

Java permet de définir des types énumérés. Une énumération est un type de données particulier, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs. Ces valeurs sont des constantes nommées, par exemple MADAME, MADEMOISELLE, MONSIEUR.

Exemple 1 :

```
enum Civilite {MADAME, MADEMOISELLE, MONSIEUR} ;
```

Les valeurs de ce type énuméré sont les 3 constantes notées : MADAME, MADEMOISELLE, MONSIEUR.

On peut ainsi déclarer des objets de type Civilite et leur affecter des valeurs constantes de ce même type :

```
Civilite civilite = Civilite.MADAME ;
```

Exemple 2 :

```
enum Jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE}
```

Les valeurs de ce type énuméré sont les 7 constantes notées : LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE.

On peut ainsi déclarer des objets de type Jour

```
Jour courant, debut ;
```

et leur affecter des valeurs constantes de ce même type :

```
courant = Jour.mercredi ;
debut = Jour.lundi ;
```

Propriétés des énumérations :

- Une énumération est en fait une classe (classe énumération). Cette classe étend la classe Enum.
- Les valeurs d'une énumération sont les seules instances possibles de cette classe. Dans notre exemple, Civilite comporte trois instances et trois seulement : MADAME, MADEMOISELLE, MONSIEUR. Ces instances (et non des champs) sont finales, donc non modifiables.

- On peut comparer les instances d'une énumération à l'aide de l'opérateur == de façon sûre, et la comparaison à l'aide de la méthode *equals()* reste possible aussi.
- Il faut toujours préfixer les constantes du type énuméré par le nom de la classe correspondante comme dans: `Civilite.MADAME`, `Civilite.MADEMOISELLE`, `Civilite.MONSIEUR`
- Il est possible de définir des méthodes spécifiques à l'intérieur de la classe constituée par un type énuméré et des constructeurs privés.
- Il est possible d'utiliser un objet d'un type énuméré dans une instruction *switch*, en employant comme étiquettes les valeurs des constantes du type correspondant.

8.2. Méthodes des énumérations

8.2.1. Méthode *toString()*

Puisque il s'agit d'un type classe, alors les énumérations dispose déjà d'une méthode *toString()*. Cette méthode retourne une chaîne de caractères qui porte le nom de la constante considérée.

Exemple :

```
package com.ensah;

enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}

public class Main {

    public static void main(String[] args) {
        String mme = Civilite.MADAME.toString();
        System.out.println(mme);

        String mlle = Civilite.MADEMOISELLE.toString();
        System.out.println(mlle);

        String mr = Civilite.MONSIEUR.toString();
        System.out.println(mr);
    }
}
```

A l'exécution on obtient

```
Problems @ Javadoc Declaration Console
<terminated> Main (2) [Java Application] C:\Program Files\Java\jd
MADAME
MADEMOISELLE
MONSIEUR
```

8.2.2. Méthode *valueOf()*

Le type énuméré dispose de la méthode `valueOf()` qui retourne la valeur énumérée à partir de sa chaîne de caractères. Il existe deux versions de cette méthode, toutes les deux statiques. La première est définie dans la classe de l'énumération, la deuxième est définie dans la classe `Enum`.

Exemple 1 :

```
package com.ensah;
enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}
public class Main {
    public static void main(String[] args) {
        Civilite civilite = Civilite.valueOf("MONSIEUR");
        if (civilite == Civilite.MONSIEUR) {
            System.out.println("Oui c'est un Monsieur");
        }
    }
}
```

Exemple 2 :

Nous pouvons avoir le même résultat avec la méthode statique `valueOf` définie dans la classe `Enum`.

```
package com.ensah;
enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}
public class Main {
    public static void main(String[] args) {
        Civilite civilite = Enum.valueOf(Civilite.class, "MONSIEUR");
        if (civilite == Civilite.MONSIEUR) {
            System.out.println("Oui c'est un Monsieur");
        }
    }
}
```

8.2.3. Méthode `values()`

La méthode statique `values()` retourne un tableau de toutes les valeurs énumérées disponibles.

```
package com.ensah;
enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}
public class Main {
    public static void main(String[] args) {
        Civilite[] civilites = Civilite.values();
        System.out.println(civilites[0].toString());
        // L'appel à toString se fait automatiquement
        System.out.println(civilites[1]);
        // L'appel à toString se fait automatiquement
        System.out.println(civilites[2]);
    }
}
```

A l'exécution on obtient

```

Problems @ Javadoc Declaration Console
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\java
MADAME
MADEMOISELLE
MONSIEUR

```

8.2.4. Méthode *ordinal()*

La méthode *ordinal()* permet de retrouver le numéro d'ordre d'un élément énuméré, dans la liste de tous les éléments d'une énumération. Le premier numéro d'ordre est 0.

```

package com.ensah;
enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}
public class Main {
public static void main(String[] args) {
Civilite civilite = Civilite.MADAME;
System.out.println("L'ordre de " + civilite.toString() + " est :" +
civilite.ordinal());
civilite = Civilite.MADEMOISELLE;
System.out.println("L'ordre de " + civilite.toString() + " est :" +
civilite.ordinal());
civilite = Civilite.MONSIEUR;
System.out.println("L'ordre de " + civilite.toString() + " est :" +
civilite.ordinal());
}
}

```

A l'exécution on obtient

```

Problems @ Javadoc Declaration Console
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\java
L'ordre de MADAME est :0
L'ordre de MADEMOISELLE est :1
L'ordre de MONSIEUR est :2

```

8.2.5. Méthode *compareTo()*

La méthode *compareTo()* compare les numéros d'ordre de deux éléments énumérés. Si le premier élément est placé avant le deuxième dans la liste, alors l'entier retourné est négatif, sinon il est positif.

```

package com.ensah;

enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}

public class Main {
    public static void main(String[] args) {
        int res = Civilite.MADAME.compareTo(Civilite.MADAME);
        System.out.println("Res = " + res);
    }
}

```

```

        res = Civilite.MADAME.compareTo(Civilite.MADEMOISELLE);
        System.out.println("Res = " + res);
        res = Civilite.MADEMOISELLE.compareTo(Civilite.MADAME);
        System.out.println("Res = " + res);
        res = Civilite.MONSIEUR.compareTo(Civilite.MADEMOISELLE);
        System.out.println("Res = " + res);
    }
}

```

A l'exécution on obtient

```

Problems @ Javadoc Declaration Console
<terminated> Main [2] [Java Application] C:\Program Files\Java\jdk-11\bin\javaw.exe
Res = 0
Res = -1
Res = 1
Res = 1

```

8.2.6. Itération sur les valeurs d'un type énuméré

Il n'est pas possible d'utiliser la boucle *for* usuelle dans ce cas. En revanche, on peut utiliser à la boucle *for... each*

```

package com.ensah;
enum Civilite {
    MADAME, MADEMOISELLE, MONSIEUR
}
public class Main {
    public static void main(String[] args) {
        System.out.println("Liste des valeurs du type Civilite");
        for (Civilite it : Civilite.values()) {
            System.out.println(it);
        }
    }
}

```

A l'exécution on obtient

```

Problems @ Javadoc Declaration Console
<terminated> Main [2] [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe
Liste des valeurs du type Civilite
MADAME
MADEMOISELLE
MONSIEUR

```

Chapitre 4 : Les chaînes de caractères

1. La classe String

1.1. Déclaration et initialisation :

La bibliothèque Java standard contient une classe prédefinie nommée *String*, permettant de manipuler des chaînes de caractères. La déclaration d'une variable objet de type chaîne de caractères peut se faire par : `String lStr ;`

lStr est destinée à contenir une référence à un objet de type *String*. Pour initialiser *lStr* par la chaîne de caractères "Tarik BOUDAA" on peut écrire : `lStr = "Tarik BOUDAA" ;`

Ici *lStr* représente une référence à un objet de type *String*.



Un objet de type **String** est **immuable** c.à.d n'est pas modifiable après sa création, cependant la référence peut bien évidemment référencer une autre chaîne de caractères. Considérons l'exemple ci-dessous :

Instructions	Schématisation de la situation
<pre>String lStr1, lStr2; lStr1 = "ENSAH" ; lStr2 = "UMP" ;</pre>	<p>This diagram shows two separate string objects, "ENSAH" and "UMP", each represented by a rounded rectangle. Two rectangular boxes labeled "lStr1" and "lStr2" contain arrows pointing to their respective string objects, illustrating that each variable holds a reference to a distinct immutable object.</p>
<pre>lStr1 = "UMP" ; lStr2 = "ENSAH" ;</pre>	<p>This diagram shows the same two string objects, "ENSAH" and "UMP". However, both "lStr1" and "lStr2" now point to the same "UMP" object, which visually represents a modification of the references without modifying the original objects.</p> <p>Les chaînes de caractères « ENSAH » et « UMP » ne sont pas modifiées. Mais il y a modification des références des objets</p>

1.2. Constructeurs de la classe String

<code>String()</code>	Crée une chaîne de caractères vide.
<code>String(char[] value)</code>	Crée un objet <i>String</i> de sorte qu'il représente la séquence de caractères qui figurent dans le tableau de caractères passé en argument.
<code>String(String original)</code>	la nouvelle chaîne créée est une copie de la chaîne passée en argument.

La classe *String* possède plusieurs autres constructeurs ; cf. la documentation officielle pour la liste complète des constructeurs : <https://docs.oracle.com/javase/10/docs/api/java/lang/String.html>

Exemples :

```
// lStr1 contient la référence à une chaîne vide
String lStr1 = new String () ;

// lStr2 contient la référence à une chaîne contenant "ENSAH"
String lStr2 = new String("ENSAH") ;

// lStr3 contient la référence à une chaîne copie de lStr2, donc contenant
// "ENSAH"
String lStr3 = new String(lStr2) ;

// Constructeur prenant un tableau de char comme paramètres
char[] tabChar = {'e','n','s','a','h'} ;
String lStr4 = new String(tabChar) ;
```

1.3. longueur d'une chaîne de caractères

La méthode `length()` renvoie la taille d'une chaîne de caractères.

Exemples :

```
public static void main(String[] args) {
    String lStr = "Bonjour Tarik";
    System.out.println("Le nombre de caractères est : " + lStr.length());
}
```

Résultat de l'exécution :

Le nombre de caractères est : 13

1.4. Concaténation de chaînes

L'opérateur `+` est défini lorsque ses deux opérandes sont des chaînes. Il fournit en résultat une nouvelle chaîne formée de la concaténation des deux autres. On peut dire la même chose pour l'opérateur `+=`.

Exemple :

Instructions	Schématisation de la situation
<pre>String lStr1, lStr2, lStr3 ; lStr1 = "ENSAH" ; lStr2 = "UMP" ;</pre>	
<pre>lStr3 = lStr1+lStr2;</pre>	

Java autorise à mélanger des chaînes et expressions d'un type primitif ou objet. Le type primitif sera convertit en chaîne de caractères, et pour le type objet la conversion est réalisée par l'appel de sa méthode `toString`.

Exemples :

```

public class Personne {
    private String cni;
    private String nom;
    public Personne(String cni, String nom) {
        this.cni = cni;
        this.nom = nom;
    }
    public String toString() {
        return "cni : " + cni + " et nom : " + nom + "";
    }
    public static void main(String[] args) {
        Personne lPrsn = new Personne("A12222", "Boudaa Tarik");
        System.out.println("Infos: " + lPrsn);
    }
}

```

Résultat d'exécution :

```
Infos: cni : A12222 et nom : Boudaa Tarik
```

Remarque :

La concaténation de deux String crée un nouveau objet, ainsi en cas de plusieurs concaténations plusieurs créations d'objet vont avoir lieu, la perte de temps qui en résulte peut devenir gênante ainsi il est déconseillé d'utiliser String pour concaténer plusieurs chaînes de caractères. Pour ce faire, dans la suite de ce cours nous allons voir une autre façon plus efficace pour ces cas.

1.5. Comparaison des chaînes de caractères

Nous avons déjà vu que pour comparer les valeurs effectives des objets il faut utiliser la méthode *equals* au lieu des opérateurs == et != qui comparent les références. La classe String dispose d'une méthode *equals* qui permet de comparer les objets de la classe String.

La classe String dispose aussi deux autres méthodes de comparaison :

- La méthode *equalsIgnoreCase* qui effectue la même comparaison, mais sans distinguer les majuscules des minuscules,
- La méthode *compareTo* qui effectue des comparaisons lexicographiques de chaînes pour savoir laquelle de deux chaînes apparaît avant une autre, en se fondant sur l'ordre des caractères.

Exemples :

```

public static void main(String[] args) {
    String lStr1 = "BoudaAtarik";
    String lStr2 = "Boudaatarik";
    if (lStr1.equalsIgnoreCase(lStr2)) {
        System.out.println("OK en ignorant la casse");
    } else {
        System.out.println("KO");
    }
    if (lStr1.equals(lStr2)) {
        System.out.println("OK en respectant la casse");
    }
    else {
        System.out.println("KO si on respecte la casse");
    }
    if (lStr1.compareTo(lStr2) > 0) {
        System.out.println("la chaîne " + lStr1

```

```

        + " vient avant la chaîne :" + lStr2);
    } else {
        System.out.println("la chaîne : " + lStr2 + " vient avant
        la chaîne : "+ lStr1);
    }
}

```

Résultat d'exécution :

```

OK en ignorant la casse
KO si on respecte la casse
la chaîne : Boudaatarik vient avant la chaîne : BoudaAtarik

```

1.6. Accéder aux caractères d'une chaîne avec la méthode *charAt*

La méthode *charAt(int index)* de la classe String permet d'accéder à un caractère de rang index dans la chaîne de caractères.

Exemple 1

```

public static void main(String[] args) {

    String lNiveau = "NIVEAU 2";

    for (int i = 0; i < lNiveau.length(); i++) {
        if (lNiveau.charAt(i) == 'U') {
            System.out.println("Un caractère 'U' est à la
            position " + i + " Dans la chaine \"" + lNiveau +
            "\"");
            break;
        }
    }
}

```

Résultat d'exécution :

```
Un caractère 'U' est à la position 5 Dans la chaine "NIVEAU 2"
```

Exemple 2 :

Écrire un programme qui lit une chaîne au clavier et qui en affiche le premier et le dernier caractère.

```

public static void main(String[] args) {
    System.out.println("Entrer une chaîne de caractères");
    Scanner lIn = new Scanner(System.in);
    String lInStr = lIn.nextLine();
    System.out.println(lInStr);
    System.out.println("Le premier caractère est : "+lInStr.charAt(0));
    System.out.println("Le dernier caractère est : " +
    lInStr.charAt(lInStr.length()-1));
}

```

Résultat d'exécution :

```

Entrer une chaîne de caractères
ENSA d'Alhoceima
ENSA d'Alhoceima
Le premier caractère est : E
Le dernier caractère est : a

```

1.7. Recherche dans une chaîne de caractères

Les méthodes ci-dessous retournent *int* qui présente la position de la première occurrence d'un caractère ou d'une partie de la chaîne :

- `indexOf(String sousChaîne)`
- `indexOf(int car)`
- `indexOf(String sousChaîne, int décalage)`
- `indexOf(int car, int décalage)`

Exemples:

```
public class StringEx {
    public static void main(String[] args) {
        String ch = "bonjourensahalhoceimaensah";
        int index1 = ch.indexOf("ensah");
        int index2 = ch.indexOf('a');
        System.out.println(index1); // ==> Affiche 7
        System.out.println(index2); // ==> Affiche 10
        int index3 = ch.indexOf(97); // 97 est le code du caractère 'a'
        System.out.println(index3); // ==> Affiche 10

        //avec décalage

        //Recherche dans la sous chaîne "jourensaHALHOCEIMAENSah"
        int index4 = ch.indexOf("ensah", 3);
        System.out.println(index4); // ==> Affiche 7

        // Recherche dans la sous chaîne "ahalHOCEIMAENSah"
        int index5 = ch.indexOf("ensah", 10);
        System.out.println(index5); // ==> Affiche 21
    }
}
```

Résultat d'exécution :

```
7
10
10
7
21
```

1.8. Remplacement de caractères

Les méthodes ci-dessous permettent de faire des remplacements dans les chaînes de caractères, ces méthodes retournent String.

Méthodes	Explication
<code>replace(char ancien, char nouveau)</code>	Remplace toutes les occurrences d'un caractère dans une chaîne par un autre caractère.
<code>replaceAll(String regex, String remplacement)</code>	C'est plus simple sans traduction : <i>Replaces each substring of this string that matches the given regular expression with the given replacement.</i>
<code>replaceFirst(String regex, String remplacement)</code>	C'est plus simple sans traduction : <i>Replaces the first substring of this string that matches the given regular expression with the given replacement.</i>

Exemple 1

Ecrire un programme qui lit une chaîne de caractères au clavier et remplace les caractères ‘<’ et ‘>’ par leur équivalent en HTML (c.à.d par et » respectivement).

```
public static void main(String[] args) {
    System.out.println("Entrer une chaîne de caractères");
    Scanner lIn = new Scanner(System.in);
    String lInStr = lIn.nextLine();
    System.out.println("le texte lu au clavier : " + lInStr);
    lInStr= lInStr.replaceAll("<", "&nbsp;");
    lInStr= lInStr.replaceAll(">", "&#187;");
    System.out.println("Texte après traitement : "+lInStr);
}
```

Résultat d'exécution :

```
Entrer une chaîne de caractères
<HTML><Body><p>Bonjour ENSAH</p></HTML></Body>
le texte lu au clavier : <HTML><Body><p>Bonjour ENSAH</p></HTML></Body>
Texte après traitement :
&nbsp;HTML&#187;&nbsp;Body&#187;&nbsp;p&#187;Bonjour
ENSAH&nbsp;/p&#187;&nbsp;/HTML&#187;&nbsp;/Body&#187;
```

Exemple 2

Ecrire un programme qui remplacer toutes les occurrences de "ENSAH" dans un texte par "ENSA Al-Hoceima" :

```
public static void main(String[] args) {
    String texte = "ENSAH est une école d'ingénieur publique, La filière
    informatique de l'ENSAH est orientée vers le domaine de développement
    logiciel";
    String ch2 = texte.replaceAll("ENSAH", "ENSA Al-Hoceima");
    System.out.println(ch2);
}
```

Résultat d'exécution :

```
ENSA Al-Hoceima est une école d'ingénieur publique, La filière
informatique de l'ENSA Al-Hoceima est orientée vers le domaine de développement
logiciel
```

1.9. Extraction de sous-chaîne

Les méthodes ci-dessous permettent d'extraire une partie de la chaîne de caractères sur laquelle elles sont appelées, le caractère indicé par *début* est pris en compte, alors que le caractère correspondant à *fin* ne l'est pas :

- *subString(int début)*
- *subString(int début, int fin)*

Exemples :

```
String ch = " bonjourensahalhoceima";
String ch1 = ch.substring (5) ;
//ch1 contient "rensahalhoceima". Bien entendu ici ch n'est pas modifiée
```

```
String ch2 = String ch1 = ch.substring(2,6) ;
//ch2 contient "njou". Bien entendu ch n'est pas modifiée
```

1.10. Conversion entre type primitif et une chaîne

Dans la classe String, la méthode statique `valueOf` est surdéfinie avec un argument des différents types primitifs, cette méthode peut être utilisée pour convertir une valeur numérique en une chaîne.

Méthodes	Explication
<code>String valueOf(Type valeur)</code>	Renvoie une chaîne de caractères correspondant à la valeur numérique passée en paramètre de la méthode.

Pour réaliser les conversions inverses on peut utiliser les méthodes des classes enveloppes :

- `Byte.parseByte`,
- `Short.parseShort`,
- `Integer.parseInt`,
- `Long.parseLong`,
- `Float.parseFloat`,
- `Double.parseDouble`.

Exemples :

Écrire un programme qui récupère deux nombres réels sur la ligne de commande et qui en affiche la somme à la console :

```
public class ExempleString {

    public static void main(String[] args) {
        if (args.length == 2) {
            double d = Double.parseDouble(args[0]) + Double.parseDouble(args[1]);
            System.out.println(d);
        } else System.out.println("Arg. Error");
    }
}
```

1.11. Autres méthodes utiles

Méthodes	Explication
<code>String toLowerCase()</code>	Convertit la chaîne en minuscule.
<code>String toUpperCase()</code>	Convertit la chaîne en majuscule.
<code>String trim()</code>	Supprime les espaces blancs de début et de fin de chaîne
<code>String[] split(String motif)</code>	Décompose la chaîne en un tableau de chaînes en utilisant comme séparateur un élément d'expression régulière.
<code>char[] toCharArray()</code>	Permet de convertir une chaîne en tableau de caractères

Remarque : Il y a en Java une classe nomée `StringTokenizer`, qui permet également de découper une chaîne en différents tokens (sous-chaînes), en se basant sur un séparateur qu'on choisit librement. Pour plus d'informations vous pouvez consulter sa documentation :

<https://docs.oracle.com/javase/10/docs/api/java/util/StringTokenizer.html>

Exemple avec la méthode `split` :

On considère que les informations concernant des collaborateurs d'une entreprise sont enregistrées dans un fichier csv (c.à.d. un fichier dont les champs sont séparés par des « ; »). Par exemple le fichier contient des lignes telles que :

```
Chami;Majd;A1211;1980/12/12
Nali;Younes;A1212;1980/12/12
Belfida;Noura;A1281;1980/12/12
```

Ecrire une méthode qui prend en paramètre une chaîne de caractères qui représente une ligne de ce fichier et retourne un tableau contenant les informations de chaque personne. Tester cette méthode dans main. On suppose que les données sont déjà lues et stockées dans une liste.

```
import java.util.ArrayList;
import java.util.List;
public class StringEx {
    public String[] getCollaborateurInfos(String ligne){
        return ligne.split(";");
    }
    public static void main(String[] args) {

        List<String> lignes = new ArrayList<String>();
        lignes.add("Chami;Majd;A1211;1980/12/12");
        lignes.add("Nali;Younes;A1212;1980/12/12");
        lignes.add("Belfida;Noura;A1281;1980/12/12");
        String[] chatTab;
        for(String it : lignes){
            chatTab = new StringEx().getCollaborateurInfos(it);
            System.out.println("Nom :" + chatTab[0]);
            System.out.println("Prenom :" + chatTab[1]);
            System.out.println("Matricule :" + chatTab[2]);
            System.out.println("-----");
        }
    }
}
```

Résultat d'exécution :

```
Nom :Chami
Prenom :Majd
Matricule :A1211
-----
Nom :Nali
Prenom :Younes
Matricule :A1212
-----
Nom :Belfida
Prenom :Noura
Matricule :A1281
-----
```

2. Autres classes pour la gestion des chaînes de caractères

Nous avons vu que les objets de type String sont immuables ainsi, la moindre modification d'une chaîne ne peut se faire qu'en créant une nouvelle instance d'une chaîne (Le cas de la concaténation par exemple). La création d'objets est en effet une opération qui peut prendre du temps ; ainsi dans quelques cas où les programmes manipulent intensivement les chaînes de caractères ; l'utilisation de *String* pourra présenter des problèmes de performance. C'est pourquoi Java dispose de deux autres classes

StringBuffer et *StringBuilder* destinées elles aussi à la manipulation des chaînes, mais dans lesquelles les objets sont modifiables.

Toutes les méthodes de la classe *StringBuffer* sont synchronisées, on peut donc utiliser des objets de cette classe sans risque dans des programmes *multi-thread*. Dans le cas des programmes avec un seul thread, le fait que les méthodes de *StringBuffer* sont synchronisées peut ralentir le programme. C'est pourquoi la librairie standard Java propose la classe *StringBuilder* qui possède exactement les mêmes fonctionnalités que la classe *StringBuffer* avec des méthodes non synchronisées.

Recommendation extraite de la documentation officielle :

« *The StringBuilder class should generally be used in preference to StringBuffer, as it supports all of the same operations but it is faster, as it performs no synchronization.* »

Conclusion :

D'une manière générale si l'algorithme doit manipuler intensivement des chaînes il est souhaitable d'effectuer les choix suivants :

1. Si on n'est pas en présence de plusieurs *threads* utiliser la classe *StringBuilder* qui a les méthodes les plus efficaces pour la manipulation des chaînes.
2. Lorsque l'on n'est pas en présence de plusieurs *threads* utiliser *StringBuffer*

Lien vers la documentation officielle :

<https://docs.oracle.com/javase/10/docs/api/java/lang/StringBuffer.html>

<https://docs.oracle.com/javase/10/docs/api/java/lang/StringBuilder.html>

Exemples :

L'exemple ci-dessous vous permet de découvrir quelques méthodes principales des deux classes *StringBuffer* et *StringBuilder*

```
public class ExempleStrings {  
  
    public static void main(String[] args) {  
  
        // Création d'un objet StringBuilder  
        StringBuilder sb = new StringBuilder();  
  
        // ou StringBuffer sb = new StringBuffer();  
  
        // Modification de l'objet on y ajoutant des chaînes de caractères  
        sb.append("Hello");  
        sb.append(" ENSA ");  
        sb.append('!');  
        System.out.println(sb.toString());  
  
        // Ajouter le caractère 'H' à la fin de ENSA  
        sb.insert(10, 'H');  
        System.out.println(sb);  
  
        // On supprime la chaîne Hello  
        sb.delete(0, 5);  
        System.out.println(sb);  
  
        // On supprime également 'H'  
    }  
}
```

```
        sb.deleteCharAt(5);
        System.out.println(sb);

        // On remplace "NS" par "NM"
        sb.replace(2, 4, "NM");
        System.out.println(sb);

        // On inverse la chaîne
        sb.reverse();
        System.out.println(sb);

    }

}
```

Résultat d'exécution :

```
Hello ENSA !
Hello ENSAH !
ENSAH !
ENSA !
ENMA !
! AMNE
```

3. Simplification de l'écriture des blocs de texte

Java 14 introduit une nouvelle notation facilitant l'écriture des blocs de texte. La première ligne d'un bloc de texte est précédée par une ligne terminée par la suite de caractères """" et la dernière est suivie de cette même suite de caractères.

Exemple :

```
public class Main {

    public static void main(String[] args) {
        String pageHTML = """
                        <HTML>
                        <BODY>
                        <APPLET
                            CODE = "App2Bout.class"
                            WIDTH   = 250
                            HEIGHT  = 100
                        >
                        </APPLET>
                        </BODY>
                        </HTML>
                    """; // fin d'un bloc de texte
        System.out.println(pageHTML);

    }
}
```

A l'exécution on obtient :

```

Problems @ Javadoc Declaration Console 
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk-17.0.2
<HTML>
<BODY>
<APPLET
    CODE = "App2Bout.class"
    WIDTH   = 250
    HEIGHT  = 100
>
</APPLET>
</BODY>
</HTML>

```

4. Résumé chaînes de caractères

La classe String

Les principaux constructeurs de la classe String

<code>String()</code>	Crée une chaîne de caractères vide.
<code>String(char[] value)</code>	Crée un objet String avec la séquence de caractères passée en argument.
<code>String(String original)</code>	la nouvelle chaîne créée est une copie de la chaîne passée en argument.

Longueur d'une chaîne de caractères

<code>int length()</code>	renvoie la taille d'une chaîne de caractères
---------------------------	--

Concaténation de chaînes

L'opérateur + permet de concaténer deux chaînes de caractères. (+= aussi)

La concaténation peut également se faire avec la méthode : `String concat(String str)`

Comparaison des chaînes de caractères

<code>boolean equals(Object anObject)</code>	Compare la chaîne de caractères avec l'objet passé en paramètre
<code>boolean equalsIgnoreCase(String anotherString)</code>	Compare la chaîne de caractères avec celle passée en paramètre, mais sans distinguer les majuscules des minuscules
<code>int compareTo(String anotherString)</code>	effectue des comparaisons lexicographiques de chaînes pour savoir laquelle de deux chaînes apparaît avant une autre, en se fondant sur l'ordre des caractères.
<code>int compareToIgnoreCase(String str)</code>	Même chose que la méthode précédente mais en ignorant la case.

Accéder aux caractères d'une chaîne avec la méthode charAt

<code>char charAt(int index)</code>	permet d'accéder à un caractère de rang <code>index</code> dans la chaîne de caractères.
-------------------------------------	--

Recherche dans une chaîne de caractères

<code>int indexOf(int ch)</code>	renvoie la position de la première occurrence d'un caractère donné. En commençant la recherche à partir du début de la chaîne jusqu'à sa fin. Si le caractère n'existe pas dans la chaîne, la valeur renournée est -1.
<code>int indexOf(int ch, int fromIndex)</code>	Comme la méthode précédente mais ici, la recherche commence à partir de <code>fromIndex</code>
<code>int indexOf(String str)</code>	Renvoie la position de la première occurrence de <code>str</code> si l'objet cible contient la chaîne <code>str</code> , sinon renvoie -1.
<code>int indexOf(String str, int fromIndex)</code>	Comme la méthode précédente mais ici la recherche commence à partir de <code>fromIndex</code>

Remplacement de caractères

<code>String replace(char ancien, char nouveau)</code>	Remplace toutes les occurrences d'un caractère (<i>ancien</i>) dans une chaîne par un autre caractère (<i>nouveau</i>).
<code>String replaceAll(String regex, String replacement)</code>	permet de remplacer toutes les occurrences d'une expression régulière <code>regex</code> par une expression de <code>replacement</code>
<code>String replaceFirst(String regex, String replacement)</code>	Retourne la chaîne dans laquelle on a remplacé la première occurrence d'une expression régulière <code>regex</code> par <code>replacement</code> .

Extraction de sous-chaîne

<code>String subString(int d)</code>	Sous-chaîne depuis l'indice <i>d</i> jusqu'à la fin
<code>String subString(int d, int f)</code>	Sous-chaîne depuis l'indice <i>d</i> jusqu'au caractère d'indice <i>f</i> non inclus (càd on s'arrête à <i>f</i> -1)

Conversion entre type primitif et une chaîne

Dans la classe String, la méthode statique `valueOf` est surdéfinie avec un argument des différents types primitifs, cette méthode peut être utilisée pour convertir une valeur numérique en une chaîne.

<code>static String valueOf(boolean b)</code>	Retourne la représentation en chaîne du booléen
<code>static String valueOf(char c)</code>	Retourne la représentation en chaîne du caractère
<code>static String valueOf(char[] data)</code>	Retourne la représentation en chaîne du tableau de caractères
<code>static String valueOf(char[] data, int offset, int count)</code>	Retourne la représentation en chaîne du sous-tableau de caractères défini à partir de l'indice <i>offset</i> et de nombre d'élément égale à <i>count</i>
<code>static String valueOf(double d)</code>	Retourne la représentation en chaîne du double
<code>static String valueOf(float f)</code>	Retourne la représentation en chaîne du float
<code>static String valueOf(int i)</code>	Retourne la représentation en chaîne du int
<code>static String valueOf(long l)</code>	Retourne la représentation en chaîne du long
<code>static String valueOf(Object obj)</code>	Retourne la représentation en chaîne de l'objet Si <i>obj</i> = null, alors elle retourne la chaîne "null"; sinon elle retourne <i>obj.toString()</i>

Pour réaliser les conversions inverses on peut utiliser les méthodes statiques des classes enveloppes :

- `Byte.parseByte(String str),`
- `Short.parseShort(String str),`
- `Integer.parseInt(String str),`
- `Long.parseLong(String str),`
- `Float.parseFloat(String str),`
- `Double.parseDouble(String str),`

Autres méthodes utiles

Méthodes	Explication
<code>String toLowerCase()</code>	Convertit la chaîne en minuscule.
<code>String toUpperCase()</code>	Convertit la chaîne en majuscule.
<code>String trim()</code>	Supprime les espaces blancs de début et de fin de chaîne
<code>String[] split(String motif)</code>	Décompose la chaîne en un tableau de chaînes en utilisant comme séparateur un élément d'expression régulière.
<code>char[] toCharArray()</code>	Permet de convertir une chaîne en tableau de caractères
<code>boolean startsWith(String prefix)</code>	Renvoie true si <i>prefix</i> est un préfixe de la chaîne
<code>boolean startsWith(String prefix, int i)</code>	Renvoie true si <i>prefix</i> est un préfixe de la chaîne à partir de <i>i</i> .
<code>boolean endsWith(String suffix)</code>	Retourne true si <i>suffix</i> est un suffixe de la chaîne

Classe StringBuffer

Construction

<code>StringBuffer()</code>	Construit une chaîne vide.
<code>StringBuffer (int l)</code>	Construit une chaîne vide de capacité initiale de <i>l</i> caractères.
<code>StringBuffer (String s)</code>	Construit une chaîne de caractères à partir de la chaîne <i>s</i>

Longueur d'une chaîne

<code>int length()</code>	la longueur de la chaîne de caractères
---------------------------	--

Concaténation

<code>StringBuffer append(boolean b)</code>	Concatène la représentation en chaîne du booléen.
<code>StringBuffer append(char c)</code>	Concatène la représentation en chaîne du caractère.
<code>StringBuffer append(char[] str)</code>	Concatène la représentation en chaîne du tableau de caractères.
<code>StringBuffer append(char[] str, int offset, int len)</code>	Concatène la représentation en chaîne du tableau de caractères.
<code>StringBuffer append(double d)</code>	Concatène la représentation en chaîne du double.
<code>StringBuffer append(float f)</code>	Concatène la représentation en chaîne du float.
<code>StringBuffer append(int i)</code>	Concatène la représentation en chaîne du int.

<code>StringBuffer append(long l)</code>	Concatène la représentation en chaîne du long.
<code>StringBuffer append(Object obj)</code>	Concatène la représentation en chaîne de l'objet
<code>StringBuffer append(String str)</code>	Concatène la représentation en chaîne de la chaîne.

Caractère et sous-chaîne.

<code>char charAt(int i)</code>	Retourne le caractère à l'indice spécifié en paramètre.
<code>String substring(int i)</code>	Sous-chaîne depuis i jusqu'à la fin
<code>String substring(int i, int l)</code>	Sous-chaîne depuis i et de longueur l.

Conversions

<code>String toString()</code>	Retourne une chaîne équivalente de type String
--------------------------------	--

Modifications

<code>StringBuffer delete(int start, int end)</code>	Enlève tous les caractères compris entre start et end.
<code>StringBuffer deleteCharAt(int index)</code>	Enlève le caractère à l'indice index
<code>StringBuffer reverse()</code>	Permet de renverser la chaîne

StringBuilder

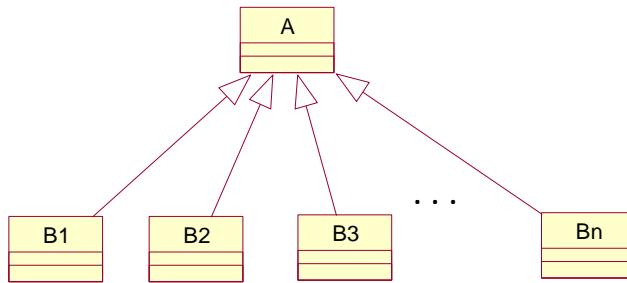
Toutes les méthodes de la classe `StringBuffer` sont synchronisées, vous pouvez donc utiliser des objets de cette classe sans risque dans des programmes *multi-thread*. Si, par contre, vous vous trouvez dans un programme avec un seul thread, le fait que les méthodes sont synchronisées peut ralentir le programme. C'est pourquoi la librairie standard Java propose la classe `StringBuilder` qui possède exactement les mêmes fonctionnalités que la classe `StringBuffer` mais avec des méthodes non thread-safe.

Chapitre 5 : Héritage et Polymorphisme

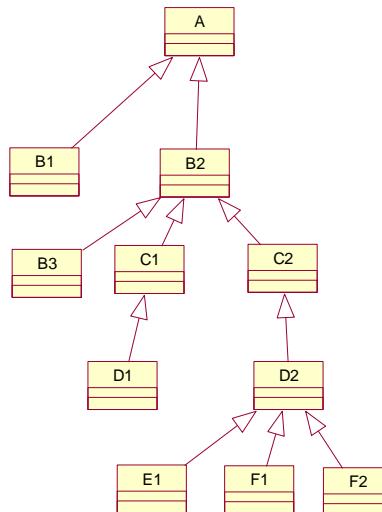
1. Généralités sur le concept de l'héritage

Le concept d'héritage constitue l'un des fondements de la programmation orientée objet. Il est notamment à l'origine des possibilités de réutilisation des composants logiciels. En effet, il permet de définir une nouvelle classe, dite classe dérivée, à partir d'une classe existante dite classe de base. Cette nouvelle classe hérite des fonctionnalités de la classe de base (champs et méthodes) qu'elle pourra modifier ou compléter à volonté, sans qu'il soit nécessaire de remettre en question la classe de base. Cette technique permet donc de développer de nouveaux outils en se fondant sur un certain acquis.

Il est possible de développer à partir d'une classe de base, autant de classes dérivées qu'on le désire (voir figure ci-dessous) :



Une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée, dans le diagramme ci-dessous B2 est une classe dérivée et en même temps est une classe de base pour B3, C1, C2 :

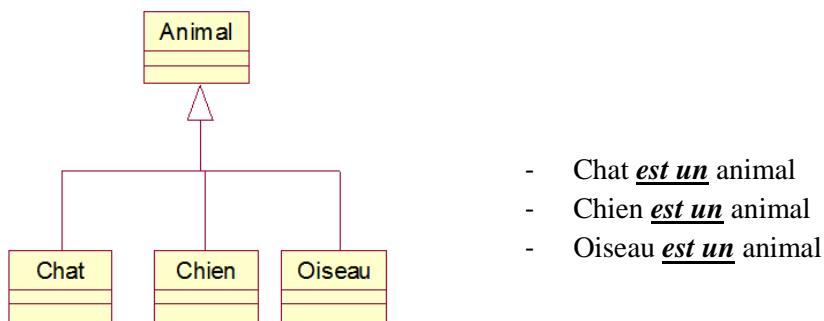


Réutiliser des classes existantes, qu'elles fassent partie de la bibliothèque Java ou qu'elles aient été créées par le développeur, est indispensable pour pouvoir développer de manière productive. La réutilisation s'exprime de plusieurs manières : une classe peut en référencer une autre grâce à la **composition** (ou **agrégation**), reprendre les caractéristiques d'une autre grâce à l'**héritage**, ou modifier l'implémentation des méthodes d'une autre grâce au **polymorphisme**.

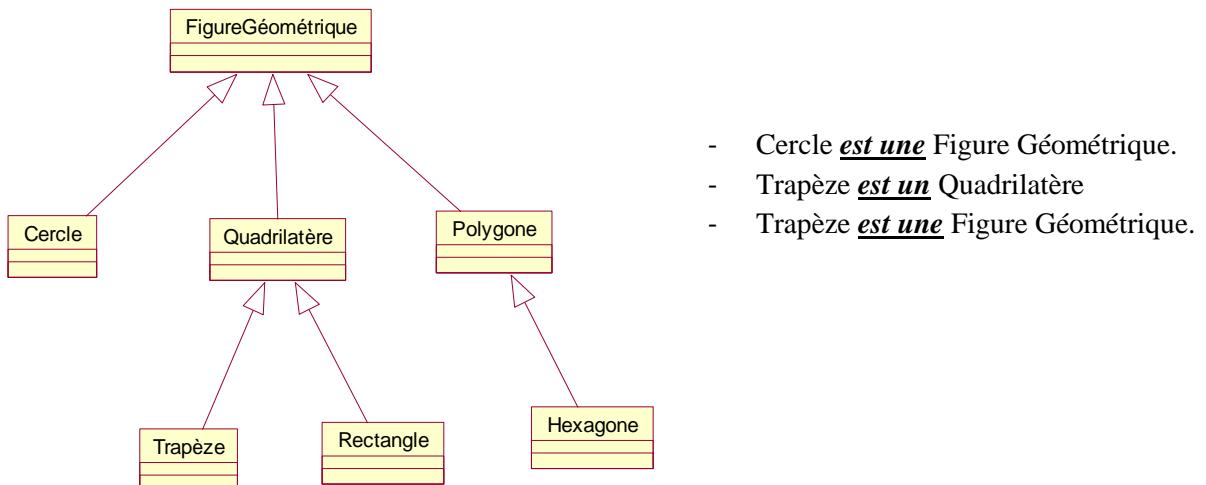
La composition associe un objet à un autre objet, définissant une relation « *a un* » entre ces deux objets : un objet de la classe *Personne* peut par exemple « *avoir* » un objet de la classe *Adresse*. Ceci se traduit en Java par un attribut de type *Adresse* dans la classe *Personne*. On dit que la classe *Adresse* est réutilisée par la classe *Personne* avec une relation de composition pour mémoriser l'adresse d'une personne (*relation a une adresse*).

L'héritage et le polymorphisme sont les deux autres concepts introduits par la programmation objet. Les classes de la programmation objet formalisent des catégories d'objets. Une catégorie donne parfois lieu à des sous-catégories, qui comptent une ou plusieurs caractéristiques supplémentaires. C'est cette hiérarchie, obtenue entre catégories et sous-catégories, que l'on tente de reproduire en programmation objet grâce à l'héritage. L'héritage définit une relation « **est un** » entre deux classes.

Exemple 1 :



Exemple 2 :



On peut voir la relation d'héritage de deux visions différentes :

- Vision descendante : la possibilité de reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir. (*Spécialisation*)
- Vision ascendante : la possibilité de regrouper en un seul endroit ce qui est commun à plusieurs classes. (*Généralisation*)

2. Mise en œuvre de l'héritage en JAVA

Commençons par un exemple simple et intuitif. Supposons que nous disposions de la classe *Personne* définie ci-dessous, et que nous ayons besoin d'une classe pour modéliser les étudiants (qui sont aussi des personnes) :

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;
    public void showDetails() {
        System.out.println("Nom : " + nom + " prenom : " + prenom + " age :" +
                           age);
    }
}
```

Une telle classe peut donc disposer des mêmes fonctionnalités que la classe *Personne*, auxquelles on pourrait ajouter, par exemple, un attribut supplémentaire pour stocker le *CNE*. Dans ce contexte, nous pouvons définir une classe *Student* comme dérivée de la classe *Personne* :

```
public class Student extends Personne {
    private String cne;

    public void setCne(String pCne) {
        cne = pCne;
    }
}
```

La syntaxe Java *A extends B*, avec *A* et *B* deux classes sert à traduire la phrase « *A hérite de B* ».

2.1. Règles d'accès d'une classe dérivée aux membres de sa classe de base

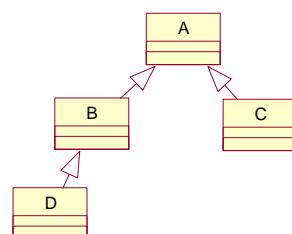
- Une classe dérivée n'accède pas aux membres privés de la classe de base.
- Une classe dérivée accède aux membres publics et aux membres protégés de la classe de base.
- Si une classe A hérite d'une autre classe B alors la partie public de A est la réunion de sa partie public et la partie public de B.
- Ainsi à la différence de C++ un membre déclaré *protected* est accessible à des classes dérivées, ainsi qu'à des classes du même paquetage.

Remarque :

Supposons que la classe A possède un attribut protégé att :

```
public class A {
    protected int att;
}
```

- B accède à l'attribut att de A
- D accède à l'attribut att de B et de A
- C n'accède à l'attribut att de B que si C et B sont dans le même package



2.2. Construction et initialisation des objets dérivés

Lors de la création d'un objet, la partie qui dépend de sa classe de base est initialisée en premier, avant celle qui dépend de sa propre classe. Si l'initialisation de la classe de base requiert l'appel d'un constructeur avec paramètres, le passage des valeurs à ce constructeur s'effectue dans le constructeur de la classe dérivée avec l'instruction *super(params)*. En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet. Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur.

2.2.1. La classe de base ne possède aucun constructeur

Dans ce cas il y aura un appel du constructeur par défaut de la classe de base

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;
}

public class Student extends Personne {
    private String cne;

    public Student(String pCne) {
        super(); <----- Appel du constructeur par défaut de A. cet
        cne = pCne;   appel sera effectué même si on supprime
    }           l'instruction super()
}
}
```

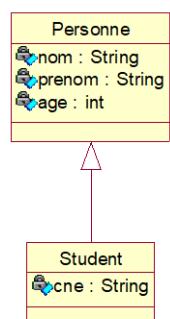
2.2.2. La classe de base et la classe dérivée disposent d'au moins un constructeur public

Le constructeur de la classe dérivée appelle le constructeur de la classe de base avec l'instruction *super(params)*

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;
    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
}

public class Student extends Personne {
    private String cne;

    public Student(String nom, String prenom, int age, String pCne) {
        super(nom, prenom, age); <----- Appel du constructeur de la classe de base
        cne = pCne;
    }
}
```



2.2.3. La classe dérivée ne possède aucun constructeur

Le constructeur par défaut de la classe dérivée appelle implicitement le constructeur par défaut de la classe de base. Donc la classe de base devra :

- soit posséder un constructeur sans arguments.
- Soit ne posséder aucune constructeur et donc il y aura appel du constructeur par défaut

Exemple 1 :

```
public class Personne {
    /Aucun constructeur
}

public class Student extends Personne {
    //Aucun constructeur
}
```

La création d'un objet Student par :
new Student(); → appel du constructeur par défaut de *Student* → appel du constructeur par défaut de *Personne*.

Exemple 2:

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;

    /** Constructeur sans args */
    public Personne() {
    }

    /** Constructeur 2 */
    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }

    /** Eventuellement d'autres constructeurs ... */
}

    
```

Dans ce cas, la création d'un objet *Student* par *new Student();* implique l'appel du constructeur par défaut de *Student* ce qui conduit à l'appel du constructeur sans arguments de *Personne*:

```
public class Student extends Personne {

    //Aucun constructeur
}
```

Exemple 3 :

```
public class Personne {

    private String nom;
    private String prenom;
    private int age;
```

```

/** Constructeur 1 */
public Personne(String nom, String prenom, int age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
}

/** Constructeur 2 */
public Personne(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
}

/** Eventuellement d'autres constructeurs*/
}

public class Student extends Personne {
    //Aucun constructeur
}

```

Le constructeur par défaut de la classe *Student* va essayer de chercher un constructeur par défaut (ou sans arguments) dans la classe *Personne* et puisque ce dernier n'existe pas alors c'est une erreur de compilation

2.3. Initialisation d'un objet dérivé

Soit le code suivant :

```
class B extends A { ..... }
```

La création d'un objet de type B se déroule en 6 étapes.

1. Allocation mémoire pour un objet de type B ; il s'agit bien de l'intégralité de la mémoire nécessaire pour un tel objet, et pas seulement pour les champs propres à B.
2. Initialisation par défaut de tous les champs de B (aussi bien ceux hérités de A, que ceux propres à B) aux valeurs par défaut.
3. Initialisation explicite, s'il y a lieu, des champs hérités de A
4. Exécution du corps du constructeur de A.
5. Initialisation explicite, s'il y a lieu, des champs propres à B ; éventuellement,
6. Exécution du corps du constructeur de B.

3. La surdéfinition et l'héritage :

La surdéfinition peut être généralisée dans le contexte de l'héritage, ainsi une classe dérivée peut surdéfinir une méthode d'une classe de base (ou, plus généralement, d'une classe ascendante). La recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage, jamais en la descendant.

Pour résoudre un appel de la forme *o.f(...)* (*o* étant un objet), **on recherche toutes les méthodes acceptables, à la fois dans la classe de l'objet *o* et dans toutes ses ascendantes. On utilise ensuite les règles habituelles de recherche de la meilleure et unique méthode.**

Exemple 1 :

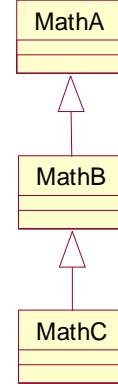
On considère la hiérarchie des classes décrite sur la figure et son code :

```
public class MathA {
    public int somme(int x, int y) {
        System.out.println("Je suis MathA");
        return x + y;
    }

    public int produit(int x, int y) {
        System.out.println("Je suis MathA");
        return x * y;
    }
}

public class MathB extends MathA {
    public float somme(float x, float y) {
        System.out.println("Je suis MathB");
        return x + y;
    }
}

public class MathC extends MathB {
    public float produit(float x, float y) {
        System.out.println("Je suis MathC");
        return x * y;
    }
}
```



Qu'affiche le programme ? et Pourquoi ?

```
public static void main(String[] args) {
    MathC c = new MathC();
    int i = 1;
    int j = 3;

    float f = 1.1f;
    float g = 3.3f;

    c.produit(i, j);
    c.produit(f, g);
    c.somme(f, g);
    c.somme(i, j);
    byte b = 1;
    c.somme(i, g);
    c.somme(b, i);
    c.somme(b, f);
}
```

Solution :

Instruction	Résultat	Explication
c.produit(i, j);	Je suis MathA	Il n'y a aucune méthode acceptable dans la classe <i>MathC</i> . La recherche d'une méthode acceptable se fait en remontant la hiérarchie d'héritage. Il n'y a aucune méthode acceptable dans la classe <i>MathB</i> . La méthode : int <i>produit</i> (int x, int y) dans <i>MathA</i> est acceptable. D'où le résultat.
c.produit(f, g);	Je suis MathC	f et g sont des float donc la méthode <i>produit</i> (float x, float y) de <i>MathC</i> est appelée
c.somme(f, g);	Je suis MathB	Il n'y a aucune méthode acceptable dans la classe <i>MathC</i> donc la méthode <i>somme</i> (float x, float y) de <i>MathB</i> sera appelée
c.somme(i, j);	Je suis MathA	Il n'y a aucune méthode acceptable dans la classe <i>MathC</i> . Les méthodes <i>somme</i> de <i>MathB</i> et <i>MathA</i> sont acceptables, la meilleure est celle de <i>MathA</i> , d'où le résultat
c.somme(i, g);	Je suis MathB	La méthode <i>somme</i> (float x, float y) dans <i>MathB</i> est la seule acceptable (après conversion de i en float)
c.somme(b, i);	Je suis MathA	Même explication que ligne 4
c.somme(b, f);	Je suis MathB	Il n'y a aucune méthode acceptable dans la classe <i>MathC</i> donc la méthode <i>somme</i> (float x, float y) de <i>MathB</i> sera appelée

Exemple 2 :

On considère la hiérarchie des classes ci-dessous :

```
public class MathA {
    public double somme(long x, double y) {
        System.out.println("Je suis MathA");
        return x + y;
    }
}
public class MathB extends MathA {
    public float somme(float x, float y) {
        System.out.println("Je suis MathB");
        return x + y;
    }
}
public class MathC extends MathB {
    public double somme(double x, double y) {
        System.out.println("Je suis MathA");
        return x + y;
    }
}
```

Qu'affiche le programme ?

```
public static void main(String[] args) {
    MathB b = new MathB();
    MathC c = new MathC();
    c.somme(1.2, 1.8);
    b.somme(1.2, 1.8);
}
```

Solution :

Etant donné que la recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage et qu'il n'existe pas de méthode *somme(double, double)* dans *mathA* ni *mathB*, le programme signalera donc une erreur de compilation dans l'instruction *b.somme(1.2, 1.8)*

4. La redéfinition des membres :

La redéfinition de méthodes permet à une classe dérivée de redéfinir une méthode de sa classe de base, en proposant une nouvelle implémentation. A la différence de la surdéfinition pour redéfinir une méthode il faut respecter la **même signature et même type de retour de la méthode à redéfinir**. C'est alors cette nouvelle méthode qui sera appelée sur tout objet de la classe dérivée. Si la signature n'est pas respectée, on a affaire à une surdéfinition.

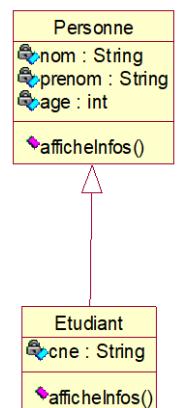
Exemple : redéfinition de la méthode *afficheInfo* dans la classe dérivée Etudiant

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;
    /** Constructeur*/
    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
    /** Affiche Infos*/
    public void afficheInfo() {
        System.out.println("Nom : "+nom+" Prénom : "+prenom+" Age : "+age);
    }
}
```

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

    public Etudiant(String nom, String prenom, int age, String cne) {
        super(nom, prenom, age);
        this.cne = cne;
    }

    /** Affiche Infos*/
    public void afficheInfo(){
        super.afficheInfo(); ←
        System.out.println("CNE : "+cne);
    }
}
```



Redéfinition de la méthode *afficheInfo*

Pour appeler la méthode *afficheInfo* de la classe de base on a utilisé le mot clé *super* qui fournit une référence sur l'objet de la classe de base.

Considérons le programme ci-dessous :

```
public static void main(String[] args) {
    Etudiant e = new Etudiant("Salimi", "Ali", 20, "A1212");
    e.afficheInfo();
}
```

A l'exécution il affiche :

Nom : Salimi Prénom :Ali Age:20
CNE : A1212

Résultat d'exécution de l'appel de la méthode *afficheInfo* de la classe Personne

Règles importantes :

1. Cas particulier des valeurs de retour covariantes : La JDK5 a introduit une exception pour la règle de contrainte sur le type de retour dans le cas d'une redéfinition : la nouvelle méthode peut depuis JDK5 renvoyer une valeur d'un type dérivé de celui de la méthode qu'elle redéfinit.

Exemple :

```
public class Personne {
    private String nom;
    private String prenom;
    private int age;

    public Personne getPersonne() {
        return new Personne();
    }

}

public class Etudiant extends Personne {
    private String cne;

    public Etudiant getPersonne() {
        return new Etudiant();
    }
}
```

La méthode *getPersonne* de la classe Etudiant redéfinit bien La méthode *getPersonne* de la classe Personne par ce qu'il s'agit d'un retour de type dérivé.

2. Si on appelle la méthode *afficheInfo* sans utiliser le mot clé super le compilateur ne signalera aucune erreur mais dans ce cas un appel récursif de la méthode *afficheInfo* va être provoqué.

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

    /** Affiche Infos*/
    public void afficheInfo(){
        afficheInfo(); ←
        System.out.println("CNE : "+cne);
    }
}
```

Provoque un appel récursif de la méthode *afficheInfo* de la classe Etudiant

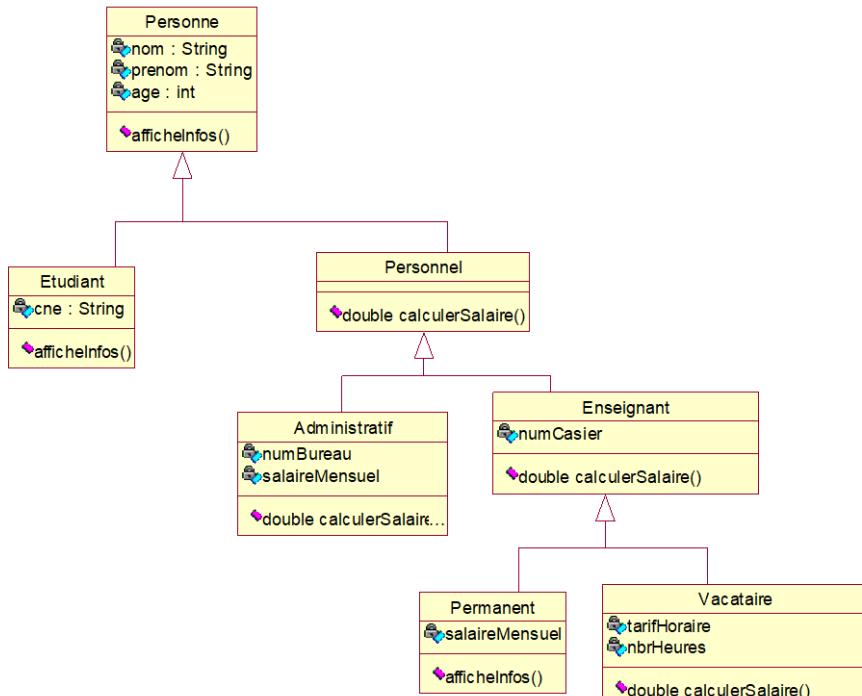
3. Rien n'empêche de redéfinir entièrement une méthode sans chercher à exploiter celle de la classe ascendante
4. L'appel par super de la méthode correspondante de la classe de base peut ne pas être la première instruction de la méthode de la classe dérivée.
5. Une classe dérivée peut définir un champ portant le même nom qu'un champ d'une classe de base ou d'une classe ascendante. Il ne s'agit pas d'une redéfinition du champ comme dans les cas des

méthodes, mais il s'agit de la création d'un nouveau champ qui s'ajoute à l'ancien. Cette possibilité est rarement utilisée dans la conception des classes.

6. Dans le cas de dérivations successives, comme on peut s'y attendre, la redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce qu'éventuellement l'une d'entre elles redéfinisse à nouveau la méthode

Exemple 1

Considérons par exemple l'arborescence ci-dessous



- i. Soit o un objet d'une classe descendante de Personne

*L'appel
o.calculerSalair()*



- Appel de *calculerSalair* de la classe Vacataire si o est une instance de Vacataire
- Appel de *calculerSalair* de la classe Enseignant si o est une instance de Enseignant ou Permanent
- Appel de *calculerSalair* de la classe Administratif si o est une instance de Administratif
- Appel de *calculerSalair* de la classe Personnel si o est une instance de Personnel

- ii. Soit v est une instance de la classe Vacataire v.afficheInfo va appeler la méthode afficheInfo de la classe Personne. De même si v est une instance de Administratif, Enseignant ou Personnel.

Exemple 2

On considère la classe Point suivante :

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void affiche() {
        System.out.println("Coordonnees : " + x + " " + y);
    }
}
```

Réaliser une classe PointNom, dérivée de Point permettant de manipuler des points définis par leurs coordonnées et un nom (caractère). On y prévoira les méthodes suivantes : Un constructeur pour définir les coordonnées et le nom d'un objet de type PointNom. Méthode affiche pour afficher les coordonnées et le nom d'un objet de type PointNom.

Solution

```
public class PointNom extends Point {
    private char nom;
    public PointNom(int x, int y, char pNom) {
        super(x, y);
        nom = pNom;
    }
    public void affiche() {
        System.out.print("Point de nom " + nom + " ");
        super.affiche();
    }
}
```

Récapitulatif des règles de redéfinition :

1. Si une méthode d'une classe dérivée a la même signature qu'une méthode d'une classe ascendante :
 - les valeurs de retour des deux méthodes doivent être exactement de même type (ou, depuis le JDK 5.0, être covariantes),
 - le droit d'accès de la méthode de la classe dérivée ne doit pas être moins élevé que celui de la classe ascendante
 - la clause throws de la méthode de la classe dérivée ne doit pas mentionner des exceptions non mentionnées dans la clause throws de la méthode de la classe ascendante.

Si ces 3 conditions sont remplies, on a à affaire à une redéfinition. Sinon, il s'agit d'une erreur de compilation.

2. Une méthode de classe (méthode *static*) ne peut pas être redéfinie dans une classe dérivée.

5. La classe Class et la classe Object

5.1. La classe Class

Les instances de la classe Class sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classe utilisée. Ces instances sont créées automatiquement par la machine virtuelle lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe Class. La classe Class permet entre autre de d'obtenir le nom d'une classe, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...

5.2. La classe Object

En Java il existe une classe nommée *Object* dont dérive implicitement toute classe simple.

Ainsi, une définition d'une classe *Personne* de cette manière :

class Personne { } Est équivalente à **class Personne extends Object{ }**

Compte tenu des possibilités de compatibilité, une variable de type *Object* peut être utilisée pour référencer un objet de type quelconque, cette particularité peut être utilisée pour manipuler des objets dont on ne connaît pas le type exact.

Bien entendu, dès qu'on souhaitera appliquer une méthode précise à un objet référencé par une variable de type *Object*, il faudra obligatoirement effectuer une conversion appropriée.

Exemple:

```
public static void main(String[] args) {
    Student s = new Student("Salimi", "Alam", 12);
    Object o = s;
    s.displayStudentInfos();
    ((Student) o).displayStudentInfos();
}
```

5.3. Les méthodes de la classe Object

La classe Object dispose de quelques méthodes qu'on peut soit utiliser telles quelles, soit redéfinir.

Méthode getClass

La méthode getClass() définit dans la classe Object renvoie une instance de la classe Class. Par héritage, tout objet java dispose de cette méthode.

```
public class ExempleGetClass {
    public static void main(java.lang.String[] args) {
        String lChaine = "test";
        Double d = new Double(1);
        Class classeString = lChaine.getClass();
        Class classeDouble = d.getClass();
        System.out.println("classe de l'objet chaine "+classeString.getName());
        System.out.println("classe de l'objet chaine = "+classeDouble.getName());
    }
}
```

Résultat :

```
classe de l'objet chaine = java.lang.String
classe de l'objet chaine = java.lang.Double
```

Ramarque : différence entre instanceof et getClass

- X instanceof Y retourne true Si Y est de même type que X ou Y est d'un type descendant de X
- X.getClass() == Y.getClass() ne retourne true que si la classe de X et la classe de Y sont identique.

La méthode `toString`

La méthode `toString` de la classe `Object` fournit une chaîne contenant :

- le nom de la classe concernée,
- la référence de l'objet en hexadécimal (précédée de @).

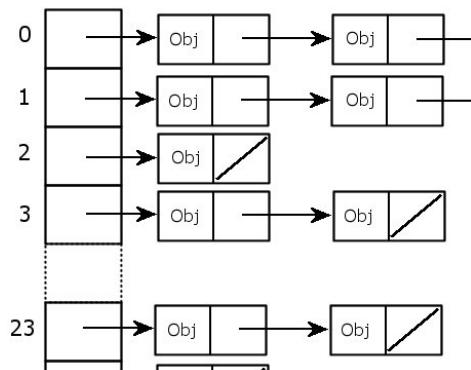
La méthode `equals`

La méthode `equals` définie dans la classe `Object` se contente de comparer les références des deux objets concernés. Bien évidemment on peut redéfinir cette méthode dans n'importe quelle classe

La méthode `hashCode`

La méthode `hashCode()` a pour objectif de fournir un code de hachage, afin d'optimiser les traitements dans certaines collections. Il est ainsi nécessaire de redéfinir la méthode `hashCode()` dès que l'on souhaite utiliser une collection qui utilise le hachage pour classer les données.

Plusieurs structures de données dans Java comme `HashSet`, `HashMap` utilisent la notion de tableau de hachage, la figure ci-dessous illustre la structure d'un tableau de hachage (que vous avez vu dans le module structure de données en GI1) :



La méthode `hashCode` est utilisée pour calculer le code de hachage d'un objet, il joue le rôle d'une fonction de hachage.

Règle à respecter : Contrat de la méthode `hashCode` :

1. Il est impératif de redéfinir `hashCode` dans toute classe où la méthode `equals` a été elle-même redéfinie
2. Si la méthode `hashCode` est invoquée plusieurs fois sur un même objet pendant l'exécution d'une application, celle-ci doit toujours renvoyer le même entier.
3. Si deux objets sont considérés égaux selon la méthode `equals` alors l'invocation de la méthode `hashCode` sur ces deux objets doit produire le même résultat
4. Il n'est pas imposé que la méthode `hashCode` de deux objets considérés comme différents par la méthode `equals` renvoie nécessairement deux résultats entiers distincts. Il est cependant important que le développeur ait conscience que la fourniture de résultats entiers différents améliore les performances des tables de hachage.

Eclipse permet de générer correctement dans la majorité des cas une méthode *hashCode*. Exemple :

```
public class Etudiant extends Personne {  
    private String cne;  
    private String filiere;  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result + ((cne == null) ? 0 : cne.hashCode());  
        result = prime * result + ((filiere == null) ? 0 : filiere.hashCode());  
        return result;  
    }  
}
```

Pour plus de détails sur cette méthode voir Chapitre méthodes communes à tous les objets, livre « Java Efficace » de Joshua Bloch



6. Principe général du polymorphisme

Le terme polymorphisme décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

En informatique, le polymorphisme désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence. Ceci veut dire que les classes issues d'une même hiérarchie sont compatibles. Ainsi, un objet de la classe *Personne* peut faire référence à un objet de la classe *Elève* (puisque un élève est aussi une personne).

Exemple 1 :

En exploitant le polymorphisme on pourra construire une liste d'objets, les uns étant de type *Personne*, les autres étant de type *Student* (dérivé de *Personne*) et appeler la méthode *afficheInfo* pour chacun des objets de la liste. Chaque objet réagira en fonction de son propre type.

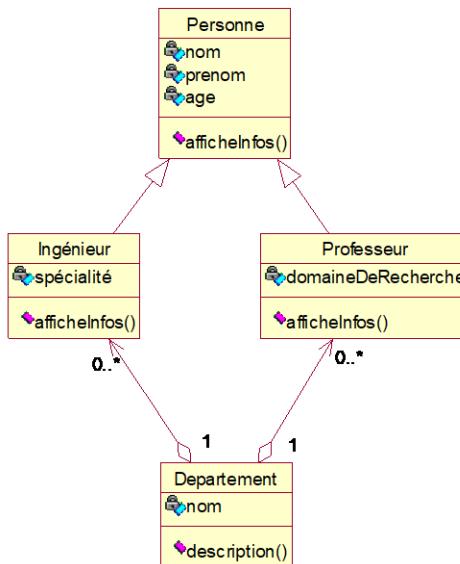
Exemple 2 :

Prenons l'exemple de modélisation d'une gestion d'un département d'une école d'ingénieur. Nous créons donc une classe nommée *Departement*. Cette classe dispose d'une méthode *description()* qui doit permettre de visualiser les informations de chacune des personnes du département.

Etudions les deux approches ci-dessous pour la conception des classes.

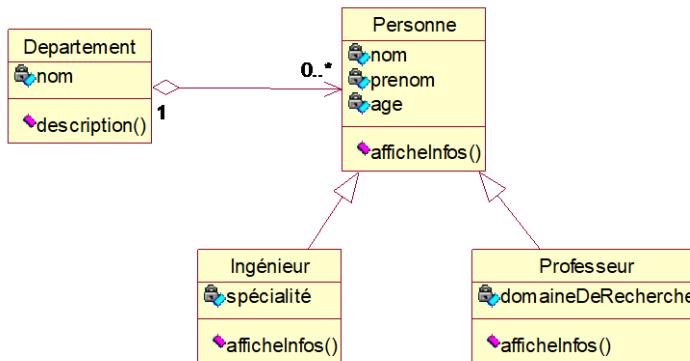
Avec la première approche, nous sommes alors obligés de proposer une composition sur chacune des classes dérivées, cette solution peut poser des problèmes dans le cas où nous devons proposer de

nouvelles classes dérivées dans cette hiérarchie. En effet, dans ce cas de figure, il serait alors nécessaire de modifier la classe *Departement* pour permettre l'intégration de la nouvelle classe dérivée.



Première approche sans utiliser le polymorphisme

Nous pouvons penser à une approche totalement différente en proposant une conception basée sur le polymorphisme. Cette fois-ci, la composition est proposée directement sur la classe de base. En fait, la classe *Departement* s'occupe de toutes les personnes sans faire de distinctions. Le mécanisme du polymorphisme permet de retrouver automatiquement de quelle personne il s'agit, afin d'afficher la bonne description.

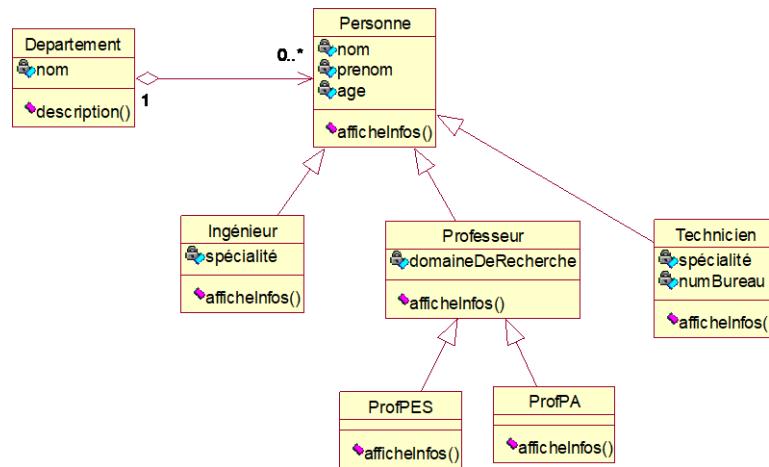


Deuxième approche avec le polymorphisme

le polymorphisme consistera donc à redéfinir correctement les méthodes *afficheInfos()* de chacune des classes faisant parties de la hiérarchie. Lorsqu'une nouvelle classe est créée, si elle veut s'intégrer dans le polymorphisme, c'est-à-dire, pouvoir participer à la modélisation de la gestion du département, elle aura pour contrat, de redéfinir sa propre méthode *afficheInfos()*. Par exemple on pourra ajouter les classes suivantes :

- Une classe *Technicien* pour présenter un technicien
- Une classe *ProfPES* pour présenter un professeur d'enseignement supérieur
- Une classe *ProfPA* pour présenter un professeur assistant

Pour participer dans la même gestion ces classes doivent respecter le contrat de redéfinition pour leurs propre méthode *afficheInfos()*.



7. Polymorphisme dans le langage JAVA

Par défaut les classes créées dans une hiérarchie dans le langage C++ (ou C#) n'intègrent pas le polymorphisme. Pour qu'une méthode soit désignée comme polymorphe, elle devra impérativement être virtuelle. En Java toutes les classes Java utilisent par défaut le polymorphisme et l'appel aux méthodes s'effectue systématiquement avec la ligature dynamique. Les mots-clés *virtual* (du C++) et *override* (de C#) n'existent pas en Java.

Le polymorphisme en Java se traduit par :

- la compatibilité par affectation entre un type classe et un type ascendant
- la ligature dynamique des méthodes.

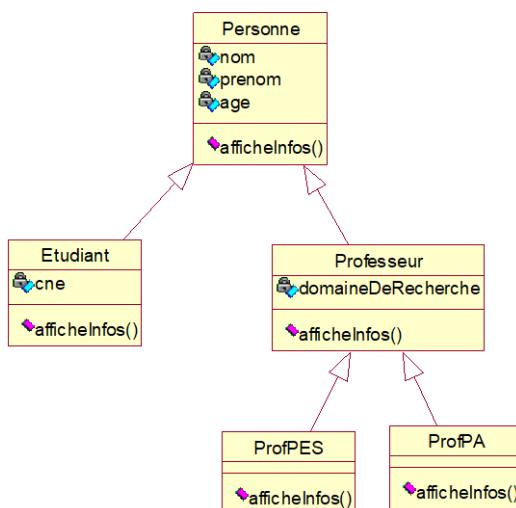
Le polymorphisme est rendu possible grâce à **l'héritage (relation est un)** et à la redéfinition des méthodes.

7.1. La compatibilité par affectation entre un type classe et un type ascendant

D'une manière générale, Java permet d'affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé.

Exemple 1 :

Considérons cette situation dans laquelle la Personne et ses dérivées sont censées disposer chacune d'une méthode *afficheInfo* :



La classe Personne :

```
public class Personne {  
    private String nom;  
    private String prenom;  
    public int age;  
    public Personne(String nom, String prenom, int age) {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.age = age;  
    }  
    public void afficheInfos(){  
        System.out.println("Nom : "+nom+" Prénom : "+prenom+" Age : "+age );  
    }  
}
```

La classe Etudiant :

```
public class Etudiant extends Personne {  
    private String cne;  
    public Etudiant(String nom, String prenom, int age, String pCne) {  
        super(nom, prenom, age);  
        cne = pCne;  
    }  
    public void afficheInfos(){  
        super.afficheInfos();  
        System.out.println(" CNE : "+cne );  
    }  
}
```

La classe Professeur :

```
public class Professeur extends Personne {  
    private String domaineDeRecherche;  
  
    public Professeur(String nom, String prenom, int age, String pDomaineDeRech)  
    {  
        super(nom, prenom, age);  
        domaineDeRecherche = pDomaineDeRech;  
    }  
    public void afficheInfos(){  
        super.afficheInfos();  
        System.out.println(" Domaine de recherche : "+domaineDeRecherche );  
    }  
}
```

La classe ProfPA:

```
public class ProfPA extends Professeur {  
  
    public ProfPA(String nom, String prenom, int age, String pDomaineDeRech) {  
        super(nom, prenom, age, pDomaineDeRech);  
    }  
  
    public void afficheInfos(){  
        System.out.println("Prof. Assistant");  
        super.afficheInfos();  
    }  
}
```

L'affectation suivante est autorisée en Java :

```
Personne p = new Etudiant("Kamali", "Mohamed", 12, "23333222");
```

La variable p est de type Personne, alors que l'objet référencé par p est de type Etudiant.

De même les affectations suivantes sont autorisées :

```
Personne p2 = new Professeur("Baha", "Fahima", 34, "bioinfo");
Personne p3 = new ProfPA("Chami", "Majd", 30, " bioinfo ");
```

La variable p2 est de type Personne, alors que l'objet référencé par p2 est de type Professeur.

La variable p3 est de type Personne, alors que l'objet référencé par p3 est de type ProfPA.

7.2. la ligature dynamique des méthodes

Considérons le même exemple précédent et examinons le résultat d'exécution de la suite d'instructions suivante :

```
Personne p = new Etudiant("Kamali", "Mohamed", 12, "23333222");
p.afficheInfos();
```

Dans la dernière instruction, la variable p est de type Personne, alors que l'objet référencé par p est de type Etudiant. L'instruction *p.afficheInfos()* appelle alors la méthode *afficheInfos* de la classe Etudiant. Le choix de la méthode à utiliser se fait dynamiquement au cours de l'exécution et non pas à la compilation. Autrement dit, au le choix de la méthode à appeler ne se base pas sur le type de la variable p, mais sur le type effectif de l'objet référencé par p. Ce mécanisme s'appelle ligature dynamique.

Exemple complet :

```
public static void main(String[] args) {
    Personne p = new Etudiant("Kamali", "Mohamed", 12, "23333222");
    Personne p2 = new Professeur("Baha", "Fahima", 34, "bioinformatique");
    Personne p3 = new ProfPA("Lamrani", "Khalid", 30, "bioinformatique");
    p.afficheInfos();
    System.out.println("-----");
    p2.afficheInfos();
    System.out.println("-----");
    p3.afficheInfos();
}
```

Résultat d'exécution :

```
Nom : Kamali Prénom : Mohamed Age : 12
CNE : 23333222
-----
Nom : Baha Prénom : Fahima Age : 34
Domaine de recherche : bioinformatique
-----
Prof. Assistant
Nom : Lamrani Prénom : Khalid Age : 30
Domaine de recherche : bioinformatique
```

8. Les classes abstraites

Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation. Elle se déclare ainsi :

```
abstract class A  
{ .....  
}
```

Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée. Mais on peut aussi trouver des méthodes dites abstraites, c'est à-dire dont on ne fournit que la signature et le type de la valeur de retour. Par exemple :

```
abstract class A {  
  
    public void f() { ..... } // f est définie dans A  
  
    // g n'est pas définie dans A ; on n'en a fourni que l'en-tête  
    public abstract void g(int n);  
  
}
```

Bien entendu, on pourra déclarer une variable de type A :

A a ; // OK : a n'est qu'une référence sur un objet de type A ou dérivé

En revanche, toute instanciation d'un objet de type A sera rejetée par le compilateur :

a = new A(...); // erreur : pas d'instanciation d'objets d'une classe abstraite

En revanche, si on dérive de A une classe B qui définit la méthode abstraite g :

```
class B extends A  
{  
    public void g(int n) { ..... } // ici, on définit g  
    ....  
}
```

on pourra alors instancier un objet de type B par new B(...) et même affecter sa référence à une variable de type A :

A a = new B(...); // OK

Quelques règles :

- Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et on doit indiquer le mot-clé *abstract* devant sa déclaration.
- Une méthode abstraite doit obligatoirement être déclarée *public*
- Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite, et il est nécessaire dans ce cas de mentionner *abstract* dans sa déclaration.

9. Les interfaces

Une interface Java est une entité distincte d'une classe qui contient une liste de méthodes abstraites que doit implémenter une classe pour rendre un service.

Si l'on considère une classe abstraite n'implantant aucune méthode et aucun champ (hormis des constantes), on aboutit à la notion d'interface (une interface est une classe abstraite pure). En effet, une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes.

Cependant la notion d'interface se révèle plus riche qu'un simple cas particulier de classe abstraite. En effet :

- une classe pourra implémenter plusieurs interfaces (*alors qu'une classe ne pouvait dériver que d'une seule classe abstraite*)
- la notion d'interface va se superposer à celle de dérivation, et non s'y substituer
- les interfaces pourront se dériver
- on pourra utiliser des variables de type interface.

9.1. Définition d'une interface

La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot-clé interface à la place de class :

```
import java.util.List;

public interface StudentManager {

    public List<Student> getAllStudents();
}
```

Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes ou des constantes.

Exemple :

```
import java.util.List;

public interface StudentManager {
    public static final String DEFAULT_CIN ="ABC";
    public List<Student> getAllStudents();
}
```

Bien entendu les méthodes d'une interface sont abstraites et publiques (puisque elles devront être redéfinies plus tard). Néanmoins, il n'est pas nécessaire de mentionner les mots-clés public et abstract (on peut quand même le faire).

9.2. Implémenter une interface

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé *implements*, comme dans :

```
import java.util.List;

public class StudentManagerImpl implements StudentManager {

    public List<Student> getAllStudents() {
```

```
    ...
}
```

Ici, on indique que StudentManagerImpl doit définir les méthodes prévues dans l'interface StudentManager.

Une même classe peut implémenter plusieurs interfaces, dans ce cas il doit obligatoirement définir les méthodes prévues dans toutes les interfaces qu'elle implémente.

Exemple :

L'interface *UserManagerService* :

```
public interface UserManagerService {
    public boolean authentification(String login, String password);
}
```

La classe *StudentManagerImpl*

```
import java.util.List;
public class StudentManagerImpl implements StudentManager,
UserManagerService {
    public List<Student> getAllStudents() {
        return null;
    }
    public boolean authentification(String login, String password) {
        return false;
    }
}
```

9.3. Héritage, polymorphisme et interface :

La clause implements est une garantie qu'offre une classe d'implémenter les fonctionnalités proposées dans une interface. Elle est totalement indépendante de l'héritage ; autrement dit, une classe dérivée peut implémenter une interface (ou plusieurs)

Il est possible de définir des variables de type interface :

```
StudentManager stMgr;
```

On pourra affecter à *stMgr* n'importe quelle référence à un objet d'une classe implémentant l'interface *StudentManager* :

```
stMgr = new StudentManagerImpl();
```

De plus, à travers *stMgr*, on pourra manipuler des objets de classes quelconques, non nécessairement liées par héritage, pour peu que ces classes implémentent l'interface *stMgr*.

Exemple :

```
public interface UserManagerService {
    public boolean authentification(String login, String password);
}

public class PersonneManagerImpl implements UserManagerService{
```

```

public boolean authentification(String login, String password) {
    System.out.println("Je suis PersonneManagerImpl");
    return true;
}

public class StudentManagerImpl extends PersonneManagerImpl implements
UserManagerService {
}

```

La notion d'interface se superpose à celle de dérivation

Il n'est pas obligatoire d'implémenter la méthode authentification, ceci est déjà fait dans la classe de base.

Si on considère le programme :

```

UserManagerService u1 = new PersonneManagerImpl();
UserManagerService u2 = new StudentManagerImpl();
u1.authentification("sds", "dd");
u2.authentification("sds", "dd");

```

Il affichera :

```

Je suis PersonneManagerImpl
Je suis PersonneManagerImpl

```

Si on redéfini la méthode authentification, les règles du polymorphisme seront appliquées :

```

public class StudentManagerImpl extends PersonneManagerImpl implements
UserManagerService {
    public boolean authentification(String login, String password) {
        System.out.println("Je suis StudentManagerImpl");
        return true;
    }
}

```

Le programme affichera dans ce cas :

```

Je suis PersonneManagerImpl
Je suis StudentManagerImpl

```

Appel de la méthode authentification redéfinie dans StudentManagerImpl

9.4. Classe interne anonymes

Il est possible de définir une classe interne, sans lui donner de nom par dérivation d'un super classe, ou par implémentation d'une interface.

La syntaxe de définition d'une classe anonyme ne s'applique que dans deux cas :

- classe anonyme dérivée d'une autre
- classe anonyme implémentant une interface

Syntaxe pour le cas Dérivation de super classe

```

SuperClasse c = new SuperClasse( ){
    //définition de la classe
    // éventuellement redéfinition des méthodes de la classe de base
}

```

};

Syntaxe pour le cas Implémentation d'interface

```
Interface c = new Interface(){  
//définition de la classe  
// implémentation obligatoire des méthodes de Interface  
};
```

Chapitre 6 : Gestion des exceptions

1. Généralités sur le traitement des erreurs

Lorsqu'un programme est en exécution, certaines circonstances exceptionnelles peuvent compromettre la poursuite normale de son exécution. Le programme peut s'arrêter d'une façon prématurée et inattendue si on ne traite pas correctement ces circonstances qui peuvent être liées à plusieurs types d'erreurs, comme par exemple :

- **Erreur matérielle** (imprimante débranchée, serveur en panne, câbles défectueux)
- **Contrainte physique** (disque plein, ...)
- **Erreurs de programmation** (cas non prévu par le programme, division par zéro, ouverture d'un fichier qui n'existe pas, ...)
- **Utilisateur récalcitrant** (type de données rentrées par l'utilisateur incorrect)
- **Erreur dans la JVM** (dépassement de la capacité de la mémoire ...)
- Et bien d'autres ...

Le traitement d'un problème dépend du contexte, parfois le traitement des problèmes possibles est décrit dans les spécifications fonctionnelles du logiciel. Ci-dessous quelques exemples de traitements qui peuvent être appliqués dans certaines situations :

- Envoyer un message d'erreur dans le fichier des logs et poursuivre l'exécution.
- Sauvegarder l'ensemble du travail en cours, et arrêter le programme (Exemple : *Google Chrome* si il se plante il enregistre les sites visités et il s'arrête afin de permettre à l'utilisateur de les récupérer après son redémarrage).
- Permettre une sortie correcte du programme, libération des ressources
- Revenir en arrière et effectuer un autre traitement.
- Répéter l'opération ayant provoqué l'erreur récursivement
-

Bien entendu, on peut envisager d'examiner toutes les situations possibles au sein du programme et faire les traitements qui s'imposent. Dans ce cas il faut tout prévoir :

- Tester l'égalité à zéro du dénominateur avant une division,
- Vérifier la syntaxe de la requête avant de l'exécuter
- Vérifier l'existence d'une table avant de l'interroger
- Vérifier le format des nombres avant les utiliser
- etc.

Cependant cette approche n'est pas efficace, en effet :

- On ne peut pas tout contrôler et on risque toujours d'oublier certaines situations
- Que faire dans le cas d'un mauvais type en entrée, ou pour les cas non prévus par le programme ?
- Le code deviendra vite illisible car sa fonction principale peut être masquée par le nombre important de liges de traitements des erreurs.

La gestion des exceptions est un mécanisme qui permet d'avoir un moyen souple et efficace pour la gestion des erreurs dans les programmes. Les exceptions permettent d'écrire le flux principal du code et de traiter les cas exceptionnels ailleurs.

Cette approche possède plusieurs avantages :

- Dissocier la détection d'une erreur de son traitement.
- Séparer la gestion des erreurs du reste du code, donc contribuer à la lisibilité des programmes.
- Séparation du code de gestion des erreurs du code de logique métier
- Meilleure lisibilité des programmes
-

D'une manière générale, une exception est une rupture de séquence déclenchée par une instruction *throw* comportant une expression de type classe. Il y a alors branchement à un ensemble d'instructions nommé "gestionnaire d'exception". Le choix du bon gestionnaire est fait en fonction du type de l'objet mentionné à *throw*.

2. Déclencher une exception avec *throw*

Considérons une classe *Etudiant* ayant deux attributs *nom* et *age* et munie d'un constructeur qui permet d'initialiser ses attributs. Supposons que nous ne voulons pas accepter des valeurs négatives pour l'âge. Pour ce faire, nous pouvons, au sein du constructeur, vérifier la validité des paramètres fournis. Ainsi, lorsqu'une valeur négative pour l'âge est passée en paramètre au constructeur nous déclençons une exception à l'aide de l'instruction *throw*. À celle-ci, nous devons fournir un objet dont le type servira ultérieurement à identifier l'exception concernée. Dans l'exemple suivant, l'objet qui représente l'erreur est de type *IllegalArgumentException*, il s'agit d'une classe prédéfinie dans Java.

```
public class Etudiant {
    private String nom;
    private int age;
    public Etudiant(String pNom, int pAge) {
        if (pAge < 0) {
            throw new IllegalArgumentException();
        }
        age = pAge;
        nom = pNom;
    }
}
```

Prenons un deuxième exemple où une méthode *debiter* d'un compte bancaire déclenche une exception si le solde est insuffisant :

```
public class Compte {
    private int idCompte;
    private int idTitulaire;
    private double solde;
    public void debiter(double montant) throws Exception {
        if (montant > solde) {
            throw new Exception(); <----- On déclenche une exception de type
        }                                -----> Exception si solde < montant
        solde -= montant;
    }
}
```

Remarquez que, à la différence du premier exemple, dans le deuxième exemple, nous avons en plus placé l'instruction *throws Exception* dans la signature de la méthode. Ceci par ce que ; comme nous allons le voir ; l'exception *IllegalArgumentException* est une exception de type implicite et *Exception* est de type explicite.

3. Utilisation d'un gestionnaire d'exception : l'instruction try/catch

Le principe de l'exception est de capturer une erreur lorsqu'on suppose qu'elle peut intervenir dans une partie de code. Ainsi, on inclut dans un bloc particulier dit "bloc *try*" les instructions pour lesquelles on risque de voir déclenchée une exception. Ce bloc *try* est suivi par un bloc de la définition de gestionnaire d'exception introduit par le mot-clé *catch*.

```
try {
    instruction 1 ;
    instruction 2 ;
    ....
    instruction i ;
    ....
    instruction n ;
}

} catch (NomException e) {
    //traitement de l'exception
    ....
}
```

Si une erreur de type *NomException* survient dans l'instruction *instruction i* alors le programme va passer directement à l'exécution du code du bloc *catch*. Bien entendu les instructions *instruction i+1,.., instruction n* ne seront pas exécutées.

Lorsqu'une méthode déclenche une exception, on cherche tout d'abord un gestionnaire dans l'éventuel bloc *try* contenant l'instruction *throw* correspondante. Si l'on n'en trouve pas ou si aucun bloc *try* n'est prévu à ce niveau, on poursuit la recherche dans un éventuel bloc *try* associé à l'instruction d'appel dans une méthode appelante, et ainsi de suite. Le gestionnaire est rarement trouvé dans la méthode qui a déclenché l'exception puisque l'un des objectifs fondamentaux du traitement d'exception est précisément de séparer déclenchement et traitement.

Exemple :

```
public class Compte {
    private int idCompte;
    private int idTitulaire;
    private double solde;

    public Compte(int idCompte, int idTitulaire, double solde) {
        this.idCompte = idCompte;
        this.idTitulaire = idTitulaire;
        this.solde = solde;
    }

    public void debiter(double m) throws Exception {
        if (m > solde) {
            throw new Exception(); <----->
        }
        solde -= m;
    }

    public static void main(String[] args) {
        Compte c = new Compte(1, 1, 100);
        try {
            System.out.println("Début de l'opération");
            c.debiter(900); <----->
            System.out.println("Fin de l'opération");
        }
    }
}
```

On déclenche une exception de type *Exception*

Une exception de type *Exception* sera déclenchée et donc il y aura une interruption et la méthode passe directement à l'exécution des instructions du bloc *catch*

```

} catch (Exception e) {
    System.out.println("L'opération ne peut pas s'effectuer car le solde est insuffisant");
}

try {
    System.out.println("Début de l'opération");
    c.debiter(50); <-----  
Il n'y aura pas d'exception puisque 50 est un
    System.out.println("Fin de l'opération");
} catch (Exception e) {
    System.out.println("L'opération ne peut pas s'effectuer car le solde est insuffisant");
}

}

```

try sans interruption

Résultat de l'exécution :

```

Début de l'opération
L'opération ne peut pas s'effectuer car le solde est insuffisant
Début de l'opération
Fin de l'opération

```

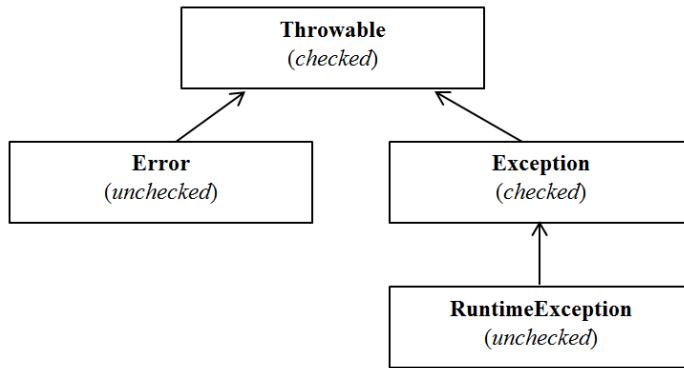
4. Exceptions standards et exception définies par l'utilisateur

Java fournit de nombreuses classes prédéfinies dérivées de la classe *Exception*, qui sont utilisées par certaines méthodes standards ; par exemple, la classe *IOException* et ses dérivées sont utilisées par les méthodes d'entrées-sorties. Certaines classes d'exceptions sont même utilisées par la machine virtuelle à la rencontre de situations anormales telles qu'un indice de tableau hors limites, une taille de tableau négative, etc. Ces exceptions standards se classent en deux catégories.

- **Les exceptions explicites** (on dit aussi vérifiées, ou *checked* en anglais): Ces exceptions doivent être traitées par une méthode, ou bien être mentionnées dans la signature de la méthode par la clause *throws*.
- **Les exceptions implicites** (on dit aussi non vérifiées, ou *unchecked* en anglais): peuvent ne pas être mentionnées dans une clause *throws* et on n'est pas obligé de les traiter (mais on peut quand même le faire).

En fait, cette classification sépare les exceptions susceptibles de se produire n'importe où dans le code de celles dont on peut distinguer les sources potentielles. Par exemple, un débordement d'indice ou une division entière par zéro peuvent se produire presque partout dans un programme ; ce n'est pas le cas d'une erreur de lecture, qui ne peut survenir que si l'on fait appel à des méthodes bien précises

La figure ci-dessous montre la hiérarchie des exceptions dans Java :



- **java.lang.Exception** (ou ses dérivées sauf *RuntimeException*) Ces erreurs doivent être traitées par l'appelant de la méthode qui lève l'exception. Ce sont des exceptions de type explicites ou *Checked Exceptions*. Elles doivent être déclarées dans la signature de la méthode après le mot clé *throws*.
- **Java.lang.RuntimeException** (ou ses dérivées) Les erreurs qui peuvent ne pas être traitées. Ce sont de type implicite *Unchecked Exceptions*. Ces d'exceptions peuvent être levées même sans déclaration préalable dans le *throws*. Exemple : *ArrayIndexOutOfBoundsException*
- **java.lang.Error** (ou ses dérivées) Les erreurs graves qui causent généralement l'arrêt du programme. Ces erreurs empêchent le bon fonctionnement de la JVM (exemple : *OutOfMemoryError*, *VirtualMachineError*). On ne doit pas rattraper ce type d'erreurs. Ces erreurs sont de type *Unchecked Exceptions*

Puisque toutes les exceptions définies par l'utilisateur sont des classes descendantes de la classe *Exception*, nous pouvons réutiliser les constructeurs et les méthodes de cette classe dans ses sous-classes. Ci-dessous les principaux constructeurs de cette classe et quelques-unes de ces méthodes :

Constructeurs :

- **Exception()** : construit une exception avec un message vide.
- **Exception(String message)** : construit une exception avec un message passé en paramètre
- **Exception(Throwable cause)** : construit une exception avec la cause passée sous forme d'un objet de type *Throwable*. Le message prend la valeur renournée par cette expression : *(cause==null ? null : cause.toString())*
- **Exception(String message, Throwable cause)** : construit une exception avec un message et une cause passés en paramètre.

Méthodes : ces méthodes sont héritées de la classe *Throwable*

- **String getMessage** : permet d'accéder au message de l'exception
- **Throwable getCause()** : permet d'accéder à la cause emballée dans l'objet *Exception*
- **void printStackTrace()** : permet d'afficher la pile de l'exception dans la sortie standard d'erreur (*System.err*)

Pour plus de détails vous pouvez voir la documentation officielle dans le lien suivant :

<https://docs.oracle.com/javase/10/docs/api/java/lang/Exception.html>

Pour créer une nouvelle exception, il suffit d'écrire une classe et la dériver d'une des classes d'exceptions standards de Java ou d'une des classes d'exceptions définies par l'utilisateur. Généralement on hérite de la classe *Exception* (et parfois de la classe *RuntimeException*).

Exemple :

```
public class CompteNotFoundException extends Exception {

    public CompteNotFoundException() {
    }

    public CompteNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }

    public CompteNotFoundException(String message) {
        super(message);
    }
}
```

5. Transmission d'information au gestionnaire d'exception

On peut transmettre une information au gestionnaire d'exception :

- Par le biais de l'objet fourni dans l'instruction *throw*,
- Par l'intermédiaire du constructeur de l'objet *Exception*.

En pratique, on utilise surtout cette seconde méthode de transmission d'information.

Exemple :

```
public class Compte {

    private int idCompte;
    private int idTitulaire;
    private double solde;

    public Compte(int idCompte, int idTitulaire, double solde) {
        this.idCompte = idCompte;
        this.idTitulaire = idTitulaire;
        this.solde = solde;
    }

    public void debiter(double m) throws Exception {
        if (m > solde) {
            throw new Exception("Le montant " + m + " est supérieur au solde du compte " + solde);
        }
        solde -= m;
    }

    public static void main(String[] args) {
        Compte c = new Compte(1, 1, 100);

        try {
            System.out.println("Début de l'opération");
            c.debiter(900);
            System.out.println("Fin de l'opération");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Passage d'informations au gestionnaire d'exception sous forme d'un message avec le constructeur de l'objet de type *Exception*

On peut récupérer les informations dans le gestionnaire des exceptions, ici nous avons utilisé la méthode *getMessage* pour récupérer le message

6. Re-déclenchement d'une exception

Dans un gestionnaire d'exception, il est possible de demander que, malgré son traitement, l'exception soit retransmise à un niveau supérieur (un autre appelant), comme si elle n'avait pas été traitée. Il suffit pour cela de la relancer en appelant à nouveau l'instruction *throw*, comme dans les exemples ci-dessous :

```
try
{
.....
}
catch (XException e) // gestionnaire des exceptions de type XException
{
.....
throw e ; // on relance l'exception e de type XException
}
```

ou

```
try
{
.....
}
catch (XException e) // gestionnaire des exceptions de type XException
{
.....
// on relance une autre exception de type AutreException
throw new AutreException(...) ;
}
```

Dans le deuxième exemple, il est recommandé de passer l'exception originale *e* de type *XException* au constructeur de *AutreException* pour remonter la stack et ne pas cacher l'origine de l'erreur aux appelants suivants.

7. Gestion de plusieurs exceptions

Pour gérer plusieurs exceptions on place plusieurs gestionnaires l'un après l'autre. Lorsqu'une exception est déclenchée dans un bloc *try*, on recherche parmi les différents gestionnaires associés celui qui correspond à l'objet mentionné à *throw*. L'examen a lieu dans l'ordre où les gestionnaires apparaissent. On sélectionne le premier qui est soit du type exact de l'objet, soit d'un type de base (*polymorphisme*). Il est donc nécessaire de placer les exceptions dans les blocs *catch* dans un ordre allant de la plus spécifique à la plus générale.

8. Le bloc finally

Java permet d'introduire, à la suite d'un bloc *try*, un bloc particulier d'instructions qui seront toujours exécutées :

- soit après la fin "naturelle" du bloc *try*, si aucune exception n'a été déclenchée
- soit après le gestionnaire d'exception (à condition, que ce dernier n'ait pas provoqué d'arrêt de l'exécution).

Ce bloc est introduit par le mot-clé *finally* et doit obligatoirement être placé après le dernier gestionnaire.

Exemple 1 :

```
public class Exemple {  
    public void f() {  
        try {  
            int i = 1 / 0;  
        } catch (Exception e) {  
            System.out.println("Ok");  
        } finally {  
            System.out.println("fin");  
        }  
    }  
    public static void main(String[] args) {  
        new Exemple().f();  
    }  
}
```

Le bloc *finally* s'exécute et donc on aura l'affichage suivant :

Ok
fin

Exemple 2

```
public class Exemple {  
    public void f() {  
        try {  
            int i = 1 / 0;  
        } catch (Exception e) {  
            System.out.println("Ok");  
            int i = 1 / 0;  
        } finally {  
            System.out.println("fin");  
        }  
    }  
    public static void main(String[] args) {  
        new Exemple().f();  
    }  
}
```

Le bloc *finally* s'exécute bien que le bloc *catch* a déclenché une erreur lui aussi, ainsi le programme affiche :

Ok
Exception in thread "main" java.lang.ArithmaticException: / by zero
fin

Exemple 3 :

Cet exemple montre que nous pouvons omettre le bloc *catch* si nous avons un bloc *finally*

```
public class Exemple {  
    public void f() {  
        try {  
            System.out.println("normal");  
        } finally {  
            System.out.println("fin");  
        }  
    }  
}
```

```
public static void main(String[] args) {
    new Exemple().f();
}
```

Le programme affiche :

normal
fin

Exemple 4

Les exemples suivants montrent que nous pouvons imbriquer les blocs *try* :

```
public class Exemple {
    public void f() {
        try {
            try {
                int i = 1 / 0;
            }
            catch (Exception e) {
                System.out.println("Ok1");
            } finally {
                System.out.println("fin");
            }
        } catch (Exception e) {
            System.out.println("Ok2");
        }
    }
    public static void main(String[] args) {
        new Exemple().f();
    }
}
```

Le programme affiche :

Ok1
Fin

Le code ci-dessous présente un autre exemple dans lequel deux bloc *try* sont imbriqués :

```
public class Exemple {
    public void f() {
        try {
            try {
                int i = 1 / 0;
            } finally {
                System.out.println("fin");
            }
        } catch (Exception e) {
            System.out.println("Ok");
        }
    }
    public static void main(String[] args) {
        new Exemple().f();
    }
}
```

Le programme affichera :

```
fin  
Ok
```

Un dernier exemple d'imbrication des blocs *try* :

```
public class Exemple {  
    public void f() {  
        try {  
            try {  
                int i = 1 / 0;  
            } catch (Exception e) {  
                System.out.println("Ok1");  
                int i = 1 / 0;  
            } finally {  
                System.out.println("fin");  
            }  
        } catch (Exception e) {  
            System.out.println("Ok2");  
        }  
    }  
  
    public static void main(String[] args) {  
        new Exemple().f();  
    }  
}
```

Le programme affichera :

```
Ok1  
fin  
Ok2
```

Exemple 5

Bien entendu nous pouvons utiliser un bloc *try/catch* à l'intérieur d'un bloc *catch*, comme dans l'exemple suivant:

```
public class Exemple {  
    public void f() {  
        try {  
            int i = 1 / 0;  
        } catch (Exception e) {  
  
            try {  
                System.out.println("Ok1");  
                int i = 1 / 0;  
            } catch (Exception ex) {  
                System.out.println("Ok2");  
            }  
        } finally {  
            System.out.println("fin");  
        }  
    }  
}
```

```
public static void main(String[] args) {
    new Exemple().f();
}
```

Le programme affichera :

```
Ok1  
Ok2  
fin
```

Exemple 06

Cet exemple montre que le bloc *finally* ne s'exécute pas si on arrête le programme explicitement dans le bloc *catch* avec un appel à *System.exit*

```
public class Exemple {
    public void f() {
        try {
            int i = 1 / 0;
        } catch (Exception e) {
            System.out.println("Ok");
            System.exit(-1);
        } finally {
            System.out.println("fin");
        }
    }

    public static void main(String[] args) {
        new Exemple().f();
    }
}
```

Le résultat de l'exécution est :

```
Ok
```

Exemple 07

Cet exemple montre que l'instruction *return* dans un bloc *catch* n'influence pas l'exécution du bloc *finally*

```
public class Exemple {
    public void f() {
        try {
            int i = 1 / 0;
        } catch (Exception e) {
            System.out.println("Ok");
            return;
        } finally {
            System.out.println("fin");
        }
    }

    public static void main(String[] args) {
        new Exemple().f();
    }
}
```

Le résultat de l'exécution est :

Ok
fin

Exemple 08 : Gestion des transactions

Le code ci-dessous montre une utilisation classique de gestion des exceptions pour la gestion des transactions

```
Connection connexion = null;
connexion.setAutoCommit(false);
try {
    // Instructions SQL de la transaction
    instruction 1;
    ...
    instruction n ;

    // Confirmation de la transaction
    connexion.commit();
} catch (SQLException e) {
    // annulation de la transaction
    connexion.rollback();
} finally {
    //Dans tous les cas il faut rendre les choses à leur état initial
    connexion.setAutoCommit(true);
}
```

Exemple 09 : I/O Pattern

Ce pattern est utilisé généralement lors d'un accès à des ressources comme les fichiers.

```
try{

    //déclaration de la ressource
    File file = new File("monfichier.txt")
    try{

        //utilisation de la ressource
        file.write("un truc");

    } finally {

        //fermeture de la ressource dans tous les cas
        file.close();
    }
} catch(IOException ex) {

    //traitement de l'exception
    traitementException(ex);
}
```

On peut aussi l'écrire de la manière suivante :

```
try {
    // déclaration de la ressource
    File file = new File("monfichier.txt");
    // utilisation de la ressource
    ...
} catch (IOException ex) {
    // traitement de l'exception
    ....
} finally {
    // fermeture de la ressource
    file.close();
}
```

Remarque : Depuis Java 7, il est possible de mentionner dans une instruction *catch* plusieurs types d'expressions, à condition qu'elles soient soumises au même traitement :

```
catch (ExceptionType1 | ExceptionType2 ex)      {
    // traitement commun aux exceptions ExceptionType1 et ExceptionType2
}
```

9. Les bonnes pratiques

- **Ne jamais ignorer une exception.**

Exemple de ce qu'il ne faut pas faire :

```
try {
    maMethodeQuiPlante();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Dans le cas où l'on ne sait pas quoi faire dans le bloc *catch* : laisser remonter l'exception à un composant technique qui saura quoi faire, ou au pire il affichera une page d'erreur convenable à l'utilisateur.

- **Utiliser throws de manière exhaustive.**

Si on a une exception *AException* héritée par *BException* et *CException* dans la signature de la méthode qui peut déclencher ces exceptions il faut utiliser *throws AException, BException, CException* et pas seulement *throws AException* afin de savoir ce qui s'est vraiment passé. (Noter que l'ordre n'est pas important dans l'instruction *throws*)

- **Les exceptions ne sont pas faites pour le contrôle de flux.**

Au lieu de ce code par exemple :

```
try {
    monattribut.maMethod();
} catch (NullPointerException npe) {
    context.reAskAttribute();
}
```

Il faudrait mieux utiliser une condition :

```
if (monattribut != null) {  
    monattribut.maMethod();  
} else {  
    context.reAskAttribute();  
}
```

- Utiliser le Pattern d'entrées/sorties lorsqu'il est nécessaire.
- Ne pas utiliser return dans un bloc finally.
- Utiliser les exceptions standards pour éviter de réinventer la roue.
- Remonter toujours la stack : une exception peut en cacher une autre : Il faut utiliser le constructeur de *Exception* qui peut prendre une exception en paramètre afin d'éviter de perdre les informations associées à la première. Il est important de lui adjoindre un message explicite.

Dans l'exemple ci-dessous le code n'est pas bon puisque on perd la stack :

```
try {  
    fct1();  
} catch (XXXException e) {  
    throw new YYYException ("la cause"); //On perd la stack  
}
```

Voici un exemple de ce qu'il faut faire : ici fct1 appelle fct2 et fct2 appelle fct3 sans perdre la stack

```
public void fct1() {  
    try {  
        fct2();  
    } catch (ZZZException e) {  
        TRACER.error("Je trace la STACK", e);  
    }  
}  
  
public void fct2() throws ZZZException {  
    try {  
        fct3();  
    } catch (ZZZException e) {  
        throw new ZZZException ("la cause 2",e);  
    }  
}  
  
public void fct3() throws ZZZException {  
    throw new ZZZException("C la cause 3");  
}
```

- Ne pas déclarer/attraper des erreurs plus large que celle qui peuvent survenir.

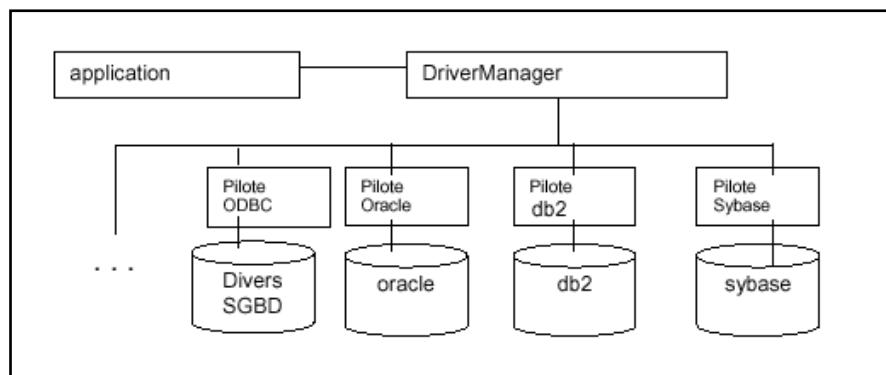
Chapitre 7 : Accès aux bases de données

1. Accès aux bases de données avec JDBC

JDBC est l'acronyme de Java DataBase Connectivity et désigne l'API Java standard qui permet d'accéder, à partir de programmes Java, à un SGBD relationnel.

1.1. Driver JDBC

Chaque éditeur de SGBDR fournit son driver JDBC sous forme d'un ensemble de classes rassemblées dans un fichier d'archive .jar, qu'il faut ajouter dans le *classpath* du projet.



En général, pour traiter une instruction SQL avec JDBC, on procède comme suit :

1. Etablissement d'une connexion.
2. Créer une instruction (Statement).
3. Exécuter la requête.
4. Traitement de l'objet ResultSet.
5. Fermer la connexion.

1.2. Etablir une connexion à une base de données

La première chose que vous devez faire, est d'établir une connexion avec votre SGBD. Cela implique deux étapes :

a. Charger les pilotes

Le chargement du pilote (ou des pilotes) à utiliser est très simple. La documentation du pilote devrait fournir le nom de la classe à utiliser. Si par exemple le nom du pilote est « *NomPilote* », vous devrez charger le pilote avec cette ligne de code :*Class.forName("NomPilote");*

- Par exemple pour MySQL : *Class.forName("com.mysql.jdbc.Driver");*
- Pour postgreSQL : *Class.forName("jdbc.postgresql.Driver");*

Une fois le pilote chargé, le programme devient êtes prêt pour créer une connexion avec un SGDB.

b. Se connecter

La deuxième étape pour établir une connexion est de faire appel à la méthode *get Connection* de la classe *DriverManager* :

```
Connection con = DriverManager.getConnection(url, "Login ", "password de la base");
```

url : est une chaîne de caractère de la forme *jdbc:accesSGBDR* avec *accesSGBDR* décrit la machine où tourne le *SGBDR* et le nom de la base de données, auxquels on veut se connecter. Exemple : Pour un serveur MySQL sur une machine ayant comme adresse ip : « *x.y.z.w* » l'url sera :

jdbc : mysql:// x.y.z.w /nomBase

Exemple :

```
try {  
    //charger le pilote de la base de données  
    Class.forName("com.mysql.jdbc.Driver");  
  
    connection = DriverManager.getConnection("jdbc:mysql://localhost/db","tba","121");  
  
} catch (SQLException e) {  
  
    // Cas d'une erreur de connexion à la base  
    //Traiter l'erreur ici  
  
} catch (ClassNotFoundException e) {  
  
    // Cas d'une erreur dans le Driver  
    //Traiter l'erreur ici  
}
```

Remarque :

- Généralement les paramètres de la base de données ne doivent pas être écrits en dure dans le code source, il faut les renseigner dans un fichier de configuration. (généralement un fichier .properties , ou xml).
- Pour libérer les ressources il faut fermer la connexion avec la base avec *connection.close()*. (généralement cette instruction se fait dans un bloc *finally*).

1.3. Traitement des commandes SQL avec JDBC

a. Créer une instruction JDBC

Un objet *Statement* est une interface qui représente une instruction SQL. On peut créer un objet *Statement* puis l'exécuter pour obtenir des résultats sous forme d'un objet *ResultSet*.

Pour exécuter un objet *Statement* on utilise la méthode *executeQuery* pour une instruction *SELECT* et la méthode *executeUpdate* pour les instructions visant à créer ou modifier des tables.

Vous devez avoir l'instance d'une connexion active pour créer un objet *Statement*.

Pour créer un objet *Statement* on utilise la méthode *createStatement* d'une connexion active (objet de type *Connection*).

Exemple 1 :

```
Statement stmt = conn.createStatement();  
stmt.executeUpdate("insert into personne values ('salimi','kamali',12)");
```

Exemple 2 :

```
Statement stmt = conn.createStatement();  
stat.executeUpdate( "delete from entreprises where codepost like '77%'");
```

Exemple 3 :

```
Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("select name from personne");
```

b. Exploiter les résultats d'une sélection SQL

Le retour d'un ordre `executeQuery(...)` est un objet de type `ResultSet`, une collection de lignes constituées de 1 à n colonnes.

Pour accéder à la première ligne du résultat, il est nécessaire d'appeler la méthode `next()`, pour passer à la ligne suivante, il suffit d'appeler de nouveau cette méthode, etc.

Exemple :

```
ResultSet rs = stat.executeQuery("select * from personne");

// Pour accéder à chacun des tuples du résultat de la requête :
while (rs.next()) {
    String nom = rs.getString("nom");
    String prenom = rs.getString("prenom");
    int age = rs.getInt("age");
    ...
}
```

L'appel à la méthode `next()` de l'objet `Statement` est obligatoire avant tout appel aux méthodes permettant d'accéder à une valeur d'une colonne de la ligne courante.

Il y a deux façons d'accéder à une valeur d'une colonne :

- Par le nom de la colonne, comme dans l'exemple précédent
- Par position, qui commence à la position 1 (et non 0 comme avec les collections et les tableaux Java)

Par exemple si on considère que la table PERSONNE à la structure suivante dans la base de données :

nom	prenom	Age
salam	morad	12

On aura les équivalences :

```
int age = rs.getInt("age"); ⇔ int age = rs.getInt(3);
String nom = rs.getString("nom"); ⇔ String nom = rs.getString(1);
```

La méthode `executeUpdate()` de Statement, ne retourne pas un objet `java.sql.ResultSet` mais retourne le nombre de lignes impactées par l'instruction.

Cet exemple supprime de la table PERSONNE toutes les personnes ayant le prénom commence par « Am »

```
int nbr = stat.executeUpdate("delete from personne where prenom like 'Am%'");
System.out.println("Il y a eu " + nbr + " lignes supprimées.");
```

c. Utilisation des requêtes paramétrées (précompilées)

Plutôt que de générer une requête SQL à chaque besoin, nous pouvons préparer une seule requête paramétrée, et nous en servir à chaque fois, mais avec des valeurs de paramètre différentes.

Chaque paramètre d'une requête préparée est spécifié par un « ? ». En reprenant l'exemple précédent, nous pourrions prévoir une requête dont le « nom » servirait de paramètre.

Exemple :

`SELECT * FROM personne WHERE nom = ?`

Cette technique apporte une certaine amélioration au niveau des performances. Chaque fois que la base de données exécute une requête, elle commence par déterminer une stratégie lui permettant d'exécuter la requête de manière efficace. En préparant une requête et en l'utilisant par la suite, vous vous assurez que la tâche de préparation n'est faite qu'une seule fois.

Pour mettre en œuvre les requêtes paramétrées, vous devez cette fois-ci utiliser la classe qui implémente l'interface *PreparedStatement* en place de la classe *Statement*, et par la même occasion, utiliser la méthode *prepareStatement* de la classe *Connection* qui attend la requête paramétrée en argument.

Exemple :

```
PreparedStatement instruction = connexion.prepareStatement("SELECT * FROM personne WHERE nom = ?");
```

Avant d'exécuter une instruction préparée, vous devez affecter les valeurs aux paramètres avec la méthode *setXXX* (numéroParamètre, valeurParamètre) associée. Comme pour les méthodes *getXXX* de *ResultSet*, il existe une méthode *setXXX* pour plusieurs types de données (*setString*, *setInt*....).

Le premier argument correspond au numéro du paramètre que nous devons définir suivant son emplacement dans la requête paramétrée. La position 1 représente le premier « ? » le second argument est la valeur que nous désirons affecter au paramètre.

Si vous utilisez une requête préparée que vous avez déjà exécutée, et que cette requête possède plusieurs paramètres, tous ces paramètres restent inchangés tant qu'ils ne sont pas modifiés par une méthode *setXXX*. Cela signifie que vous avez uniquement besoin d'appeler la méthode *setXXX* pour les paramètres qui doivent être modifiés d'une requête à l'autre.

Si vous voulez effacer toutes les valeurs données aux paramètres vous pouvez le faire avec la méthode *clearParameters()*.

Exemple 1

```
Class.forName("com.mysql.jdbc.Driver");
Connection connexion = DriverManager.getConnection("jdbc:mysql://localhost/gscompte", "root", "kirato");
PreparedStatement lPstat = connexion.prepareStatement("SELECT * FROM personne WHERE nom = ?");
lPstat.setString(1, "Salami");
ResultSet res = lPstat.executeQuery();
```

Exemple 2

```
String strDeleteClient = "DELETE FROM CLIENT WHERE ID = ?";
try {
    PreparedStatement stmtDeleteClient = connect.prepareStatement(strDeleteClient);
    stmtDeleteClient.clearParameters();
    stmtDeleteClient.setInt(1, id);
    stmtDeleteClient.executeUpdate();

} catch (SQLException e) {

    // Traitement de l'exception
}
```

Après exécution d'une requête paramétrée on obtient un objet de type *ResultSet* ainsi l'exploitation des résultats d'une requête se fait de la même manière que les requêtes normales.

1.4. Les transactions

Une transaction est un groupe d'instructions SQL qui doivent être effectuées ensemble pour assurer la cohérence d'une base de données.

Exemple parfois, vous ne voulez pas qu'une instruction prenne effet sans qu'une autre lui succède.

Par exemple lors d'une opération informatique de transfert d'argent d'un compte bancaire sur un autre compte bancaire, il y a une tâche de retrait d'argent sur le compte source et une de dépôt sur le compte

cible. Le programme informatique qui effectue cette transaction va s'assurer que les deux opérations peuvent être effectuées sans erreur, et dans ce cas, la modification deviendra alors effective sur les deux comptes. Si ce n'est pas le cas l'opération est annulée. Les deux comptes gardent leurs valeurs initiales. On garantit ainsi la cohérence des données entre les deux comptes.

Cette cohérence est assurée par l'utilisation des transactions informatiques.

Avec *Jdbc* les trois méthodes *setAutoCommit*, *commit*, *rollback* gèrent les transactions sur une base de données.

- *Commit* : valide une transaction
- *rollback* : annule une transaction et remet les valeurs qui ont été modifiées à leurs valeurs originales

Par défaut, *Connection* est en mode auto-commit, c'est-à-dire que chaque changement dans la base est effectivement enregistré après l'exécution de chaque requête.

Pour autoriser deux ou plusieurs instructions à être groupées dans une transaction il faut mettre hors service le mode auto-commit. C'est ce que nous faisons par *setAutoCommit(false)* (passage au mode transactionnel)

Une fois que le mode auto-commit est désactivé, aucune instruction SQL ne prendra effet jusqu'à ce que la méthode *commit* soit employée. Toutes les instructions exécutées après l'appel de la méthode *commit* seront incluent dans la transaction et donc, pourront prendre effet en tant qu'unité.

Généralement le pattern ci-dessous est employé :

```
Connection connexion = null;
connexion.setAutoCommit(false);

try {
    Instruction1 ;
    Instruction2 ;
    Instruction3 ;
    ... ;
    instructionN ;
} } Instructions SQL de la transaction
connexion.commit(); //Confirmation de la transaction

} catch (SQLException e) {
    // annulation de la transaction
    connexion.rollback();
} finally {
    connexion.setAutoCommit(true);
}
```

Pour plus de détails sur les transactions voir le cours de J2EE de M. BOUDAA Tarik sur la plateforme e-services.

2. Les injections SQL

Considérons une page d'authentification avec deux champs, login et password. Le code serveur permettant de vérifier l'identité et le mot de passe est une requête SQL qui ressemble à

```
request = "SELECT * FROM members WHERE ident=''' + login + ''' AND pwd=''' + password + '''";
```

Si l'utilisateur saisit ensah comme login et admin123 comme mot de passe, la requête serveur devient :

```
request = "SELECT * FROM members WHERE ident='ensah' AND pwd='admin123';
```

Or si l'utilisateur est un pirate, il pourra entrer sur le site sans connaitre d'identifiant valide, en saisissant dans le champ login : " OR 1=1 --

La requête devient alors :

```
requete = "SELECT * FROM members WHERE ident=''' OR 1=1 --' AND pwd=''';
```

La requête est toujours vraie et cela permet au pirate d'entrer sur le site. Mais ce n'est tout, le compte retourné par la requête est le premier enregistrement dans la base de données, ce qui correspond, dans la grande majorité des cas à un compte administrateur.

Remarque : le texte de la requête après les -- est ignoré et vu comme du commentaire par les SGBD. Pour faire face aux injections SQL il faut respecter plusieurs bonnes pratiques dans le développement par exemple :

- a. Vérifier de manière précise et exhaustive l'ensemble des données venant de l'utilisateur
- b. Utiliser des comptes utilisateurs SQL à accès limité (en lecture-seule) quand cela est possible.
- c. Utiliser des requêtes SQL paramétrées
- d. Comparer le nom et mot de passe de l'usager avec ceux retournés par la requête SQL
- e.

3. Utilisation d'un ORM pour la gestion de la persistance

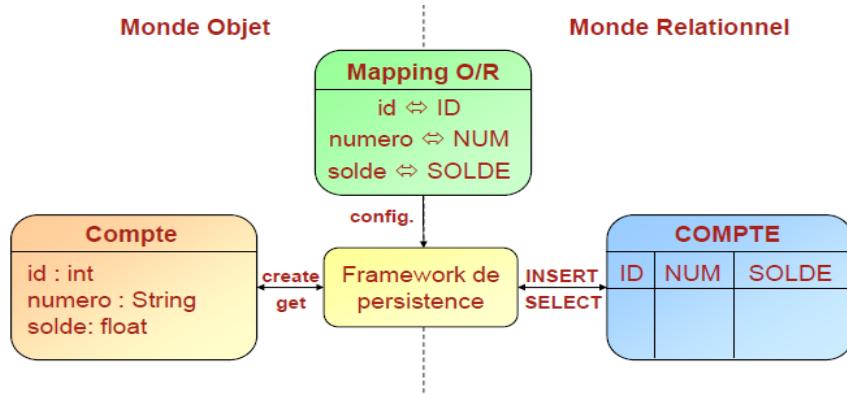
« Cette partie présente d'une façon générale les fonctionnalités d'un ORM, une description plus approfondie sera donnée en parallèle dans les séances de TP ».

La plupart des données manipulées par les applications doivent être stockées dans des bases de données relationnelles. Dans un système de gestion de base de données relationnelle, les données sont organisées en tables formées de lignes et de colonnes ; elles sont identifiées par des clés primaires et, parfois, par des index. Les relations entre tables utilisent les clés étrangères et joignent les tables en respectant des contraintes d'intégrité. Ce vocabulaire est totalement étranger à un langage orienté objet. En Java par exemple, nous manipulons des objets qui sont des instances de classes ; les objets héritent les uns des autres. Cependant, bien que les objets encapsulent soigneusement leur état et leur comportement, cet état n'est accessible que lorsque la machine virtuelle (JVM) s'exécute, lorsqu'elle s'arrête ou que le ramasse-miettes nettoie la mémoire, tout disparait. Pour pouvoir stocker l'état d'un objet d'une façon permanente, il doit être persisté sur un système de stockage comme la base de données. Dans ce cas on dit que l'objet est persistant, c'dà, il peut stocker son état afin de pouvoir le réutiliser plus tard

Il existe plusieurs moyens pour persister l'état des objets :

- Sérialisation (insuffisante pour les applications complexes)
- JDBC (plusieurs inconvénients pour les grands projets)
- ORM (Mapping Objet Relationnel)

Le mapping objet-relationnel (en anglais object-relational mapping ou ORM) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé. On pourrait le désigner par « correspondance entre monde objet et monde relationnel »



Un outil de mapping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le *mapping* des tables avec les classes, des champs avec les attributs, des relations et des cardinalités.
- Proposer une interface qui facilite la mise en œuvre des actions de type *CRUD*
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base de données utilisée.
- Proposer un support pour les transactions
- Assurer une gestion des accès concurrents (*verrou, deadlock, ...*)
- Fournir des fonctionnalités pour améliorer les performances (*cache, lazyloading, ...*)
- ...

3.1. Différence en Framework et API

Un framework, désigne un ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel (*architecture*). Un framework se distingue d'une simple bibliothèque logicielle principalement par :

- **son caractère générique, faiblement spécialisé**, contrairement à certaines bibliothèques ; un framework peut à ce titre être constitué de plusieurs bibliothèques chacune spécialisée dans un domaine. Un framework peut néanmoins être spécialisé, sur un langage particulier, une plateforme spécifique, un domaine particulier : communication de données, data mapping, etc.
- le cadre de travail (traduction littérale de l'anglais : framework) qu'il impose de par sa construction même, **guidant l'architecture logicielle voire conduisant le développeur à respecter certains patrons de conception** ; les bibliothèques le constituant sont alors organisées selon le même paradigme.

Une **interface de programmation** applicative (API pour Application Programming Interface) est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. Elle est offerte par une bibliothèque logicielle ou un service web, le plus souvent accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur.

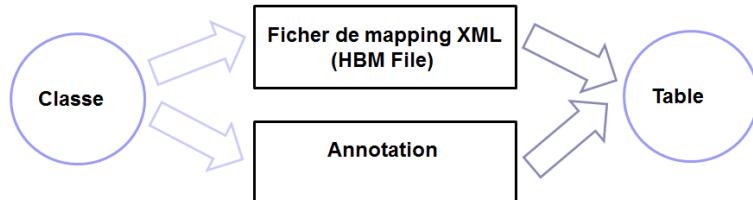
3.2. Framework Hibernate

Dans le monde Java il existe plusieurs Frameworks ORM : Hibernate, iBatis, EclipseLink, TopLink ...
Hibernate est l'un des plus utilisé actuellement. Cette section décrit d'une façon générale ce Framework, son étude plus détaillée sera effectuée sous forme de travaux pratiques.

Hibernate est un outil de *mapping O/R* qui permet la persistance transparente pour des objets Java dans des bases de données relationnelles. Il regroupe un ensemble de librairies assurant la tâche de persistance.

Ci-dessous quelques fonctionnalités et caractéristiques du Framework Hibernate :

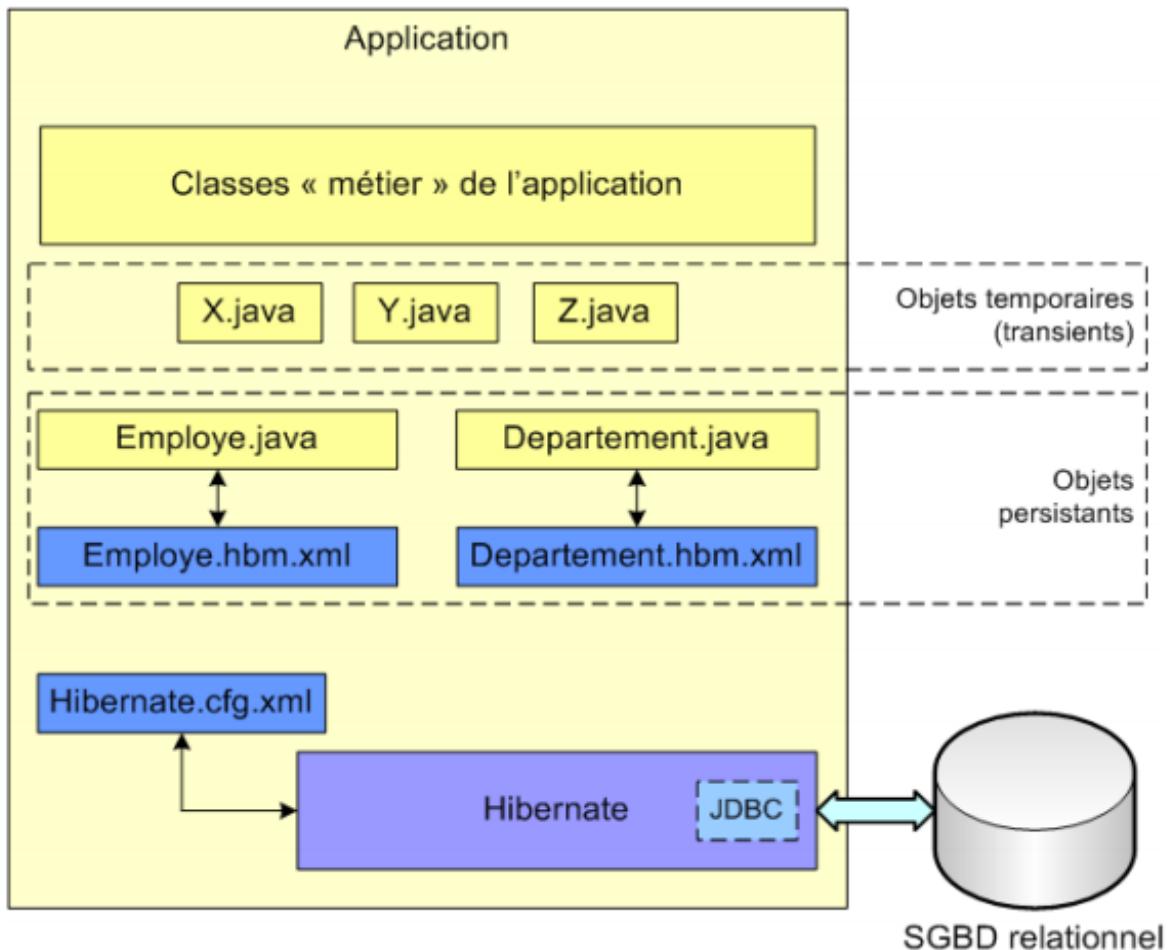
- Hibernate génère le code SQL, ainsi :
 - ✓ Pas de requête SQL à écrire
 - ✓ Pas d'Objet *ResultSet* à gérer : Cycle de récupération manuelle des *ResultSet* et le « *Casting* » de chaque ligne du *ResultSet* (type *Object*) vers un type d'objet métier.
- Permet la persistance transparente : Le développeur peut faire de ses classes métiers des classes persistantes sans ajout de code tiers.
- Les objets métiers sont plus faciles à manipuler.
- Théoriquement il n'y a pas dépendance envers une base de données précise.
- Il permet de gérer les problématiques courantes de persistance d'une façon efficace : le chargement tardif, système de cache, gestion de la concurrence, pool de connexion...
- Il existe deux moyens pour faire le *mapping* avec des tables et des classes :
 - ✓ En utilisant les annotations
 - ✓ En utilisant des configurations XML (*hbm files*)



3.2.1. Le fichier hibernate.cfg

Parce que Hibernate est conçu pour fonctionner dans différents environnements, il existe beaucoup de paramètres de configuration à fixer (paramètres d'accès à la base de données : login, mot de passe, le dialecte SQL à utiliser, taille du pool de connexion, configuration des transactions, ...). Ces configurations s'effectuent dans le fichier *hibernate.cfg.xml* (ou *hibernate.properties*)

3.2.2. Contenu d'une (simple) application Hibernate

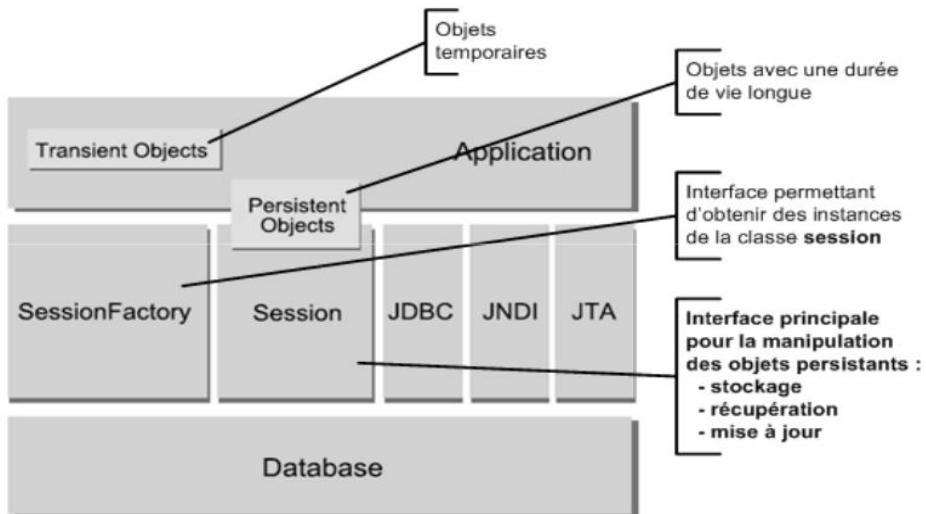


3.2.3. Session Hibernate et SessionFactory

Session Hibernate : est un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Fabrique des objets Transaction. La Session contient un cache (de premier niveau) des objets persistants, qui sont utilisés lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

SessionFactory :

- Un cache threadsafe de mappages compilés pour une base de données. Elle permet de fabriquer les Sessions
- Elle est Client du *ConnectionProvider* (une fabrique de connexions JDBC)
- SessionFactory peut contenir un cache optionnel de données (de second niveau)

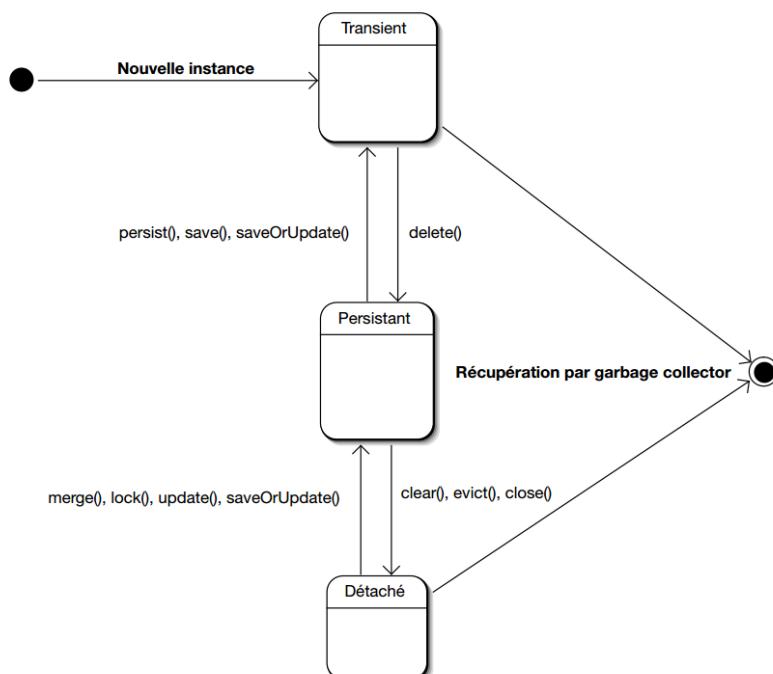


3.2.4. Le cycle de vie d'un objet manipulé avec Hibernate

Il y a trois états possibles pour les instances d'objets :

- **Transient (temporaire, éphémère)** : Objet n'ayant pas d'image dans la base de donnée (ne survivent pas à l'arrêt de l'application).
- **Persistant** : Objet stocké dans la base de données, liés à un contexte de persistance (objet session). Il y a garantie par Hibernate de l'équivalence entre les données stockées en base et l'objet Java persistant
- **Détaché** : ancien objet persistant en dehors d'un contexte de persistance. Cet objet n'est pas surveillé par la session.

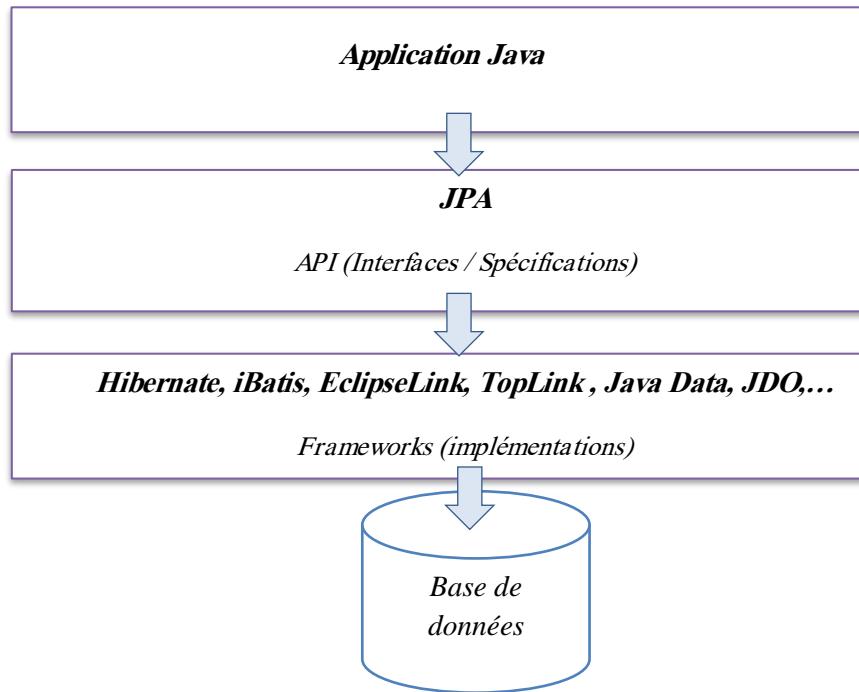
La figure suivante résume le cycle de vie d'un objet manipulé par Hibernate et les méthodes permettant de passer d'un état à un autre :



4. L'API JPA

La technologie JPA (*Java Persistence API*) a pour objectif d'offrir un modèle d'ORM (*Object Relational Mapping*) indépendant d'un produit particulier ou une implémentation particulière (comme Hibernate, TopLink, etc.). Cette technologie est basée sur un ensemble d'interfaces et de classes permettant de séparer l'utilisateur d'un service de persistance (votre application) et le fournisseur d'un service de persistance (l'implémentation comme Hibernate).

La technologie est utilisable dans les différents types d'applications Java (dans un conteneur Web, dans un conteneur EJB ou une application Java standard)



« Pour les exemples de codes sources et la mise en pratique du Framework Hibernante voir le TP ».

Chapitre 9 : La générnicité dans Java

En cours de rédaction

Chapitre 10 : Les collections

En cours de rédaction

Chapitre 11 : Les Threads

En cours de rédaction

Chapitre 12 : Initiation à la programmation graphique avec Java

En cours de rédaction