

CHAPITRE 1. BASES DU LANGUAGE JAVA

ELYAHYAOU.S - JAVA

1. Historique & versions de Java

Les débuts de Java (Source : Wikipedia)

▪ Origine du projet Java

Le langage Java est issu d'un projet de Sun Microsystems datant de 1990 : l'ingénieur Patrick Naughton n'était pas satisfait par le langage C++ utilisé chez Sun, ses interfaces de programmation en langage C, ainsi que les outils associés. Alors qu'il envisageait une migration vers NeXT, on lui proposa de travailler sur une nouvelle technologie et c'est ainsi que le Projet Stealth (furtif) vit le jour. Le projet sera rebaptisé Green Project.

▪ Choix du nom

Le nom « Java » n'est pas une abréviation, il a été choisi lors d'un brainstorming en remplacement du nom d'origine « Oak », à cause d'un conflit avec une marque existante, parce que le café (« java » en argot américain) est la boisson favorite de nombreux programmeurs. Le logo choisi par Sun est d'ailleurs une tasse de café fumant.

▪ 1994 – Lancement public

En octobre 1994, HotJava et la plate-forme Java furent présentés pour Sun Executives. Java 1.0a fut disponible en téléchargement en 1994 mais la première version publique du navigateur HotJava arriva le 23 mai 1995 à la conférence SunWorld.

L'annonce fut effectuée par John Gage, le directeur scientifique de Sun Microsystems. Son annonce fut accompagnée de l'annonce surprise de Marc Andressen, vice-président de l'exécutif de Netscape que Netscape allait inclure le support de Java dans ses navigateurs. Le 9 janvier 1996, le groupe Javasoft fut constitué par Sun Microsystems pour développer cette technologie. Deux semaines plus tard la première version de Java était disponible.

▪ 2000 – La version 2 de Java

L'apparition de la version 1.2 du langage marque un tournant significatif : c'est en 2000 qu'apparaît simultanément la déclinaison en deux plateformes Java :

- Java 2 Standard Edition (J2SE), plateforme avec les API et bibliothèques de bases, devenue depuis Java SE ;
- Java 2 Enterprise Edition (J2EE), extension avec des technologies pour le développement d'applications d'entreprise, devenue Java EE.

Sun les qualifie alors de plateforme Java 2 par opposition aux premières générations 1.0 et 1.1. Toutes les versions ultérieures, de J2EE 1.2 à Java SE ou Java EE 7 restent désignées sous le qualificatif de plateformes Java 2, bien que le '2' ait été depuis officiellement abandonné.

▪ 2006 – Passage sous licence open-source

Le 11 novembre 2006, le code source du compilateur javac et de la machine virtuelle HotSpot ont été publiés en Open Source sous la Licence publique générale GNU.

Le 13 novembre 2006, Sun Microsystems annonce le passage de Java, c'est-à-dire le JDK (JRE et outils de développement) et les environnements Java EE (déjà sous licence CDDL) et Java ME sous licence GPL d'ici mars 2007, sous le nom de projet OpenJDK.

▪ Acquisition par Oracle

La société Oracle a acquis en 2009 l'entreprise Sun Microsystems. On peut désormais voir apparaître le logo Oracle dans les documentations de l'api Java.

Le 12 avril 2010, James Gosling, le créateur du langage de programmation Java, démissionne d'Oracle pour des motifs qu'il ne souhaite pas divulguer. Il était devenu le directeur technologique de la division logicielle client pour Oracle.

▪ Historique des versions – actuelle : JavaSE 10

Version	Last update	Dénomination JSE/JRE	Spécifications	JDK	Période de maintenance	Support étendu
1.0	1.0.2	Java 1.0	JSR 52	JDK 1.0.2	1996-2000	
1.1	8_16	Java 1.1	JSR 52	1.1.8_16	1997-2000	
1.2	2_017	J2SE 1.2	JSR 52	1.2.2_11	2000-2006	
1.3	1_29	J2SE 1.3	JSR 58	1.3.1_29	2000-2001	
1.4	2_30	J2SE 1.4	JSR 59	1.4.2_30	2000-2008	
1.5	0_22 à 0_85	J2SE 5.0	JSR 176	1.5.0_22	2002-2009	Mai 2015
1.6	0_45 à 0_111	Java SE 6	JSR 270	6u113	2005-2013	Décembre 2018
1.7	0_79 à 0_80	Java SE 7	JSR 336	1.7.0_79	2011- 2015	Juillet 2022
1.8	0_171	Java SE 8	JSR 337	1.8.0_171	2014-sept2018	Juillet 2019
9	9.0.4	Java SE 9	JSR379	9.0.4	2018-?	
10	10.0.1	Java SE 10	JSR383	10.0.1	2018-?	

2. L'environnement de développement Java 8

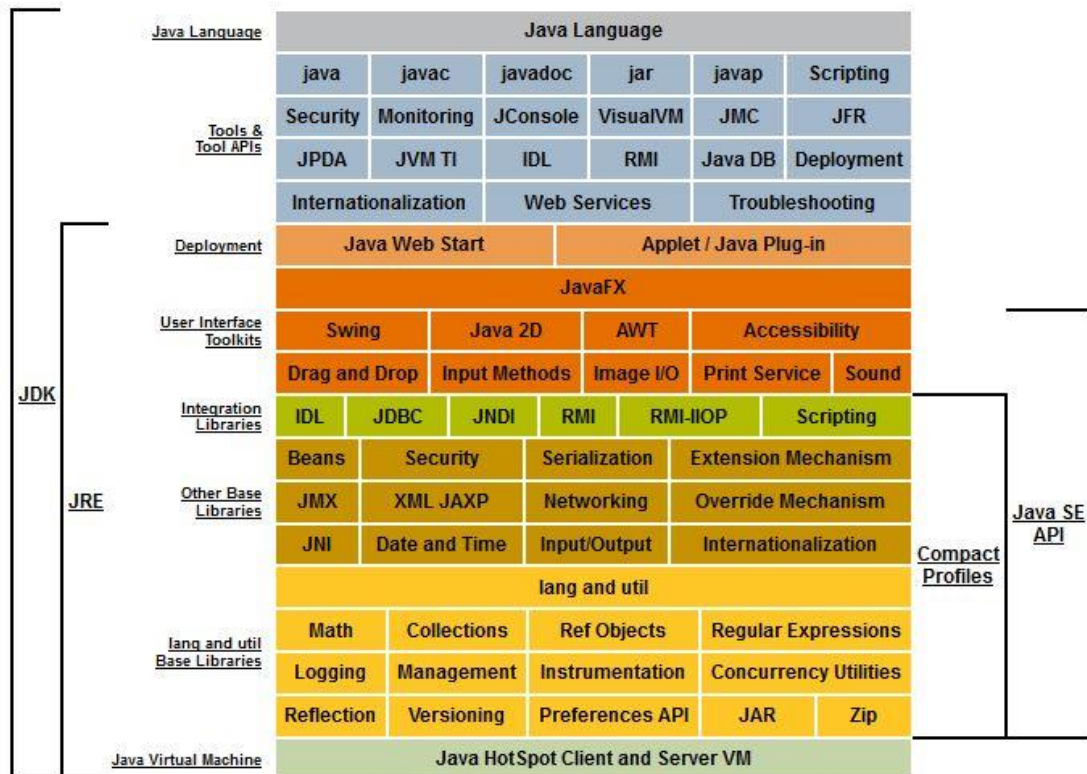
Installer Java

Ce tutoriel repose sur la version 8 de Java.

La version 8 du JDK (Java Development Kit) de la version JavaSE (Standard Edition) de java est téléchargeable sur cette page :

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

Liste des APIs contenues dans JavaSE :



L'ensemble des classes Java que toute application Java doit utiliser pour accéder à une BD dans un SGBD, est contenu dans l'API JDBC.

Installer l'IDE

L'interface de développement utilisée dans ce cours sera l'IDE (Integrated Development Environment) NetBeans 8.2 téléchargeable sur cette page :

<https://netbeans.org/downloads/>

3. Un 1^{er} programme

Exemple

```
/**
 * @author s_elyahyaoui
 */
public class JAVAII {
    public static void main(String[] args) {
        System.out.print("Bonjour tout le monde!");
    }
}
```

Output - JAVA-II (run) × Javadoc

```
run:
Bonjour tout le monde!
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Les types de variables primitifs & constantes

Définition : Types de base

Les types de base en Java, ou types primitifs, ou types simples, sont pratiquement les mêmes qu'en langage **C** :

- **char** : 16 bits (un seul caractère : '...')
- **short** : 16 bits (entier signé)
- **int** : 32 bits (entier signé)
- **long** : 64 bits (entier signé)
- **float** : 32 bits (réel)
- **double** : 64 bits (réel)
- **byte** : 8 bits (entier signé)
- **boolean** : 1 bit (true / false)



REMARQUES

- En Java le type **String** (chaîne de caractères : "...") est un type référence et non un type simple.
- Par convention, les noms de tous les types référence doivent commencer par une lettre majuscule.

Définition : Noms de variables

Le nom d'une variable

- ne doit pas contenir d'espaces,
- ne doit pas commencer par un symbole (sauf le symbole `_` ou `$`),
- ne doit pas être un mot réservé de Java.

- | | |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> • abstract • boolean • break • byte • case • catch • char • class • continue • const • default • do • double • else • enum (nouveau Java 5.0) • extends • false • final • finally • float • for • goto • if • implements • import • instanceof | <ul style="list-style-type: none"> • int • interface • long • native • new • null • package • private • protected • public • return • short • static • super • switch • synchronized • this • throw • throws • transient • true • try • void • volatile • while |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- La déclaration et l'initialisation se font de la même manière qu'en langage **C**,

Ex.

```
int c = 25000;
```

sauf pour le cas où une valeur numérique est affectée à un type dont la taille est inférieure, dans ce cas il s'agit d'une erreur de compilation

Ex.

```
byte c = 25000;
```

- Les réels sont par défaut des « **double** », sauf s'ils sont suivis d'un '**f**' ou '**F**' pour indiquer qu'il s'agit d'un « **float** »
- L'utilisation de variables non initialisées est une erreur de compilation.

Définition : Constante

- Les constantes sont déclarées avec le mot clef **final**,
- le reste de la syntaxe est le même que pour les autres variables,
- elles doivent être initialisées à la déclaration,
- leur contenu ne peut pas être modifié.

Exemple

```
int a = 5;
int x = a - 15;

int n1, n2;

int n3 = 100, n4 = -2;
```

```
short b = (short) a; /* ici un « cast » est nécessaire pour éviter une erreur de compilation.
```

Le casting (ou trans-typage) est l'action de changer le type d'une valeur vers le type de la variable d'affectation (b). le langage java est un langage très typé, à l'inverse d'autres langages comme php ou javascript.

```
*/
```

```
float y = 5.06F;
```

```
double z = 0.23;
```

```
boolean test = (y >= z);
```

```
char s = 'A';
```

```
char p = '*';
```

```
final int k = 15;
```

```
k = 16; // erreur de compilation
```

5. Les types références


Définition

Les types références sont :

- les types références prédéfinis de Java (*String*, *Integer*, *List*, ...),
- les types créés par le programmeur,
- les tableaux.

Pour les types simples, la zone mémoire allouée contient la valeur associée au nom de la variable, tandis que pour les types références ou *objets* (voir prochain chapitre), cette zone mémoire contient l'adresse mémoire (en hexadécimal) où sont stockés les champs de cet objet.

6. Les opérateurs arithmétiques & logiques

Définitions : opérateurs arithmétiques 1			
=	opérateur d'affectation		
%	modulo : reste de division euclidienne		
==	égale : test d'égalité entre deux chiffres		
!=	différent de : test d'inégalité entre deux chiffres		
<	inférieur		
>	supérieur		
<=	inférieur ou égale		
>=	supérieur ou égale		
+	addition de chiffres / concaténation de chaines de caractères		
*	multiplication		
-	soustraction		
/	division		
Définitions : opérateurs arithmétiques 2			
+= (incrémententation)	x += a;	x = x + a;	<code>int x = 1;</code> <code>x += 5; // la valeur de x est 6</code>
-= (décrémententation)	x -= a;	x = x - a;	<code>int x = 9;</code> <code>x -= 5; // la valeur de x est 4</code>
++ (incrémententation par 1)	x++; ou bien ++x;	x = x + 1;	<code>int x = 9;</code> <code>x++; // la valeur de x est 10</code> <code>++x; // la valeur de x est 11</code>
-- (décrémententation par 1)	x--; ou bien --x;	x = x - 1;	<code>int x = 9;</code> <code>x--; // la valeur de x est donc 8</code> <code>--x; // la valeur de x est donc 7</code>
*=	x *= a;	x = x * a;	<code>int x = 10;</code> <code>x *= 5; // la valeur de x est 50</code>
/=	x /= a;	x = x / a;	<code>int x = 12;</code> <code>x /= 4; // la valeur de x est 3</code>
%=	x %= a;	x = x % a;	<code>int x = 27;</code> <code>x %= 5; // la valeur de x est 2</code>
<div> IMPORTANT</div> <p>1- Les opérateurs ++ et -- peuvent être utilisés dans des opérations d'affectation et de calcul. Exemple :</p> <pre>int x = 10; int n = ++x;</pre> <p>Le résultat de l'opération dépend de l'emplacement de l'opérateur ++ C.à.d. ++x (dans ce cas on parle de Pré-incrémententation) ou bien x++ (dans ce cas on parle de Post-incrémententation)</p>			

Exemple :

Décrémentement (opérateur --)													
Pré	Post												
Cas d'utilisation : Opération d'affectation													
<pre>int x = 10, n = 0; n = --x;</pre> <table border="1"> <thead> <tr><th>x</th><th>n</th></tr> </thead> <tbody> <tr><td>10</td><td>0</td></tr> <tr><td>9</td><td>9</td></tr> </tbody> </table> <p>L'opération d'affectation de n (n = ...), utilise ici la nouvelle valeur de x, c.à.d. la valeur de x après la décrémentement</p>	x	n	10	0	9	9	<pre>int x = 10, n = 0; n = x--;</pre> <table border="1"> <thead> <tr><th>x</th><th>n</th></tr> </thead> <tbody> <tr><td>10</td><td>0</td></tr> <tr><td>9</td><td>10</td></tr> </tbody> </table> <p>L'opération d'affectation de n, utilise ici <u>l'ancienne</u> valeur de x, c.à.d. la valeur de x avant la décrémentement</p>	x	n	10	0	9	10
x	n												
10	0												
9	9												
x	n												
10	0												
9	10												
Cas d'utilisation : Opérations arithmétiques													
<pre>int x = 10, n = 0; n = 2 * --x;</pre> <table border="1"> <thead> <tr><th>x</th><th>n</th></tr> </thead> <tbody> <tr><td>10</td><td>0</td></tr> <tr><td>9</td><td>18</td></tr> </tbody> </table> <p>L'opérateur de multiplication (... * ...), utilise ici la nouvelle valeur de x, c.à.d. la valeur de x après la décrémentement</p>	x	n	10	0	9	18	<pre>int x = 10, n = 0; n = 2 * x--;</pre> <table border="1"> <thead> <tr><th>x</th><th>n</th></tr> </thead> <tbody> <tr><td>10</td><td>0</td></tr> <tr><td>9</td><td>20</td></tr> </tbody> </table> <p>L'opérateur de multiplication (... * ...), utilise ici <u>l'ancienne</u> valeur de x, c.à.d. la valeur de x avant la décrémentement</p>	x	n	10	0	9	20
x	n												
10	0												
9	18												
x	n												
10	0												
9	20												
Cas d'utilisation : Affichage à l'écran													
<pre>int x = 0; System.out.print(--x);</pre> <p>Valeur de x : -1 Valeur affichée à l'écran : -1</p> <p>La fonction d'affichage utilise ici la nouvelle valeur de x, c.à.d. la valeur de x après la décrémentement.</p>	<pre>int x = 0; System.out.print(x--);</pre> <p>Valeur de x : -1 Valeur affichée à l'écran : 0</p> <p>L'opération d'affichage, utilise ici <u>l'ancienne</u> valeur de x, c.à.d. la valeur de x avant la décrémentement.</p>												

Résumé : (Les cas étudiés sont valables pour l'opérateur ++ également)

Les opérateurs ++ et -- ne sont prioritaires par rapport aux autres opérations que quand il s'agit de pré-incrémentement ou de pré-décrémentement.

2- Les opérateurs += , -= , *= , /= et %= sont **toujours prioritaires** par rapport à l'opérateur d'affectation =

Exercice 1

Déterminer les valeurs de la variable **x** après chaque instruction :

double x = 0;	0
x *= 10;	...
x /= 5;	...
x %= 2;	...
x += 10;	...
x *= 1.25;	...

Exercice 2

Déterminer les valeurs des variables à chaque instruction du programme :

```
{
  int i = 1;
  int n = i++;
  i = 10;  n = ++i;
  i = 20;  int j = 5;  n = i++ * ++j;
  i = 15;  n = i += 3;
  i = 3;  j = 5;  n = i *= --j;
}
```

i	j	n
1		
2		1
11		11
21	6	120
18	6	18
12	4	12

Exercice 3

```
{
  byte i = 4, j = 1, k = 31, n;
  n = ... ;
}
```

En utilisant les **3** variables **i**, **j** et **k**, et l'opérateur **++** ou **--**, compléter la 2^{ème} instruction pour que la variable **n** prenne la valeur **150**.

Définition : Expression logique

Une expression logique est une expression qui n'a comme valeur possible que l'une des valeurs **Vrai** ou **Faux**.

Exemple :

```
byte x = 5;
```

```
byte y = 2;
```

```
bool z = (x == y);
```

Expression
logique

Un opérateur logique est un opérateur qui agit sur des variables et des expressions logiques.

Définition : Opérateurs logiques

le **NON**
!

A	! A : <i>non A</i>
VRAI	FAUX
FAUX	VRAI

le **OU**
|| ou bien **|**

A	B	A B : <i>A ou B</i>
VRAI	VRAI	VRAI
VRAI	FAUX	VRAI
FAUX	VRAI	VRAI
FAUX	FAUX	FAUX

le **ET**
&& ou bien **&**

A	B	A && B : <i>A et B</i>
VRAI	VRAI	VRAI
VRAI	FAUX	FAUX
FAUX	VRAI	FAUX
FAUX	FAUX	FAUX

le **OU EXCLUSIF**
^

A	B	A ^ B : <i>A ou bien B</i>
VRAI	VRAI	FAUX
VRAI	FAUX	VRAI
FAUX	VRAI	VRAI
FAUX	FAUX	FAUX

Exercice

Donner des valeurs aux variables **i**, **j** et **k** pour que les variables **a**, **b** et **c** reçoivent toutes la valeur **true** :

```
{
  bool i = ..., j = ..., k = ...;
  bool a = (i && k) ^ (j || i);

  i = ...; j = ...; k = ...;
  bool b = (i ^ !j) && (!k && j);

  i = ...; j = ...; k = ...;
  bool c = (i || true) && ((j & k) && i);
}
```

i	j	k
...
...
...

7. Afficher / lire des données

Définition : Imprimer des données vers « le fichier de sortie standard » du programme

Comme on le remarque dans le premier exemple, l'instruction **System.out.println(...)** sert à afficher toutes sortes de données (numériques, chaînes de caractères, ...), tout en passant à la ligne suivante.

- **System** : cette classe utilitaire permet, entre autres, d'utiliser l'entrée standard (le clavier) et la sortie standard (l'écran).
- **out** : objet de type **PrintStream** (flux d'écriture), et attribut de la classe **System**, cet attribut gère la sortie standard, qui est par défaut : le périphérique d'affichage (l'écran).
- **println** : méthode de l'objet **out** qui écrit dans la console les informations passées en paramètre (chaînes de caractères, nombres, ...)
- **print** : idem que **println**, sans retour à la ligne.
- Pour faire une concaténation entre des chaînes de caractères, ou entre chaînes et variables, on utilise l'opérateur « + »



REMARQUE

Les notions de programmation orientée objets (Classes, Objets, Méthodes, ...) seront détaillées dans le prochain chapitre.

Définition : Lire des données depuis « le fichier d'entrée standard » du programme

- Créer un objet de type **Scanner**, classe utilisée pour lire des informations depuis différentes sources.
- Le constructeur de la classe **Scanner** doit prendre un objet de type **InputStream** (flux d'entrée), qui doit être **System.in**, autre attribut de la classe **System**, qui représente l'entrée standard, qui est par défaut : le clavier.

// la classe **Scanner** se trouve dans le package : **java.util**

/* toutes les classes java sont organisées dans des bibliothèques, logées dans des dossiers appelés en java « packages » */

```
java.util.Scanner sc = new Scanner(System.in);
```

- Utiliser les méthodes **nextInt()**, **nextDouble()**, **nextLine()** de l'objet **Scanner**, ... pour lire un entier, un réel, du texte, ...

Exemple

```

8  import java.util.Scanner;
9
10 public class JAVAI {
11     public static void main(String[] args) {
12         System.out.println("tapez le jour, le mois et l'annee de naissance :");
13         Scanner sc = new Scanner(System.in);
14         short jour = sc.nextShort();
15         short mois = sc.nextShort();
16         short annee = sc.nextShort();
17         sc.nextLine();
18         System.out.println("tapez votre nom :");
19         String nom = sc.nextLine();
20
21         System.out.println("\nBonjour M." +
22                             nom +
23                             ", \nVotre date de naissance est le : " +
24                             jour + "/" + mois + "/" + annee
25                             );
26     }
27 }

```

Tester sans la ligne 17. Interpréter le résultat.

```
tapez le jour, le mois et l'annee de naissance :
1
1
1998
tapez votre nom :
ELYAHYAOU

Bonjour M.ELYAHYAOU,
Votre date de naissance est le :1/1/1998
```

Exercice 1

Ecrire un programme qui demande à l'utilisateur de taper deux chiffres, et qui affiche leur somme, leur produit et leur modulo.

Exercice 2

Ecrire un programme qui calcule et affiche le volume d'une sphère, dont le rayon sera lu depuis le clavier.

Rappel : $\text{Volume} = \pi * \text{Rayon}^3 * 4/3$

Exercice 3

Ecrire un programme qui demande à l'utilisateur de taper les valeurs deux chiffres **x** et **y**, et qui échange les valeurs de **x** et **y**.

Afficher les deux valeurs après l'échange.

Exercice 4

Ecrire un programme qui demande à l'utilisateur la quantité commandée et le prix unitaire de 3 produits. (Si un chiffre tapé est incorrect, on considère la valeur **0**).

Ensuite le programme doit afficher :

Le total du produit 1,

Le total du produit 2,

Le total du produit 3 (avec une réduction de 10% pour le total du produit 3),

Le total de la commande,

Et le prix TTC (Total + TVA) à payer, sachant que la valeur de la TVA est de 20%.

L'exécution doit se dérouler exactement de la manière suivante :

```
tapez la quantite commandee du produit 1 :      1
tapez le prix unitaire du produit 1 :    100

tapez la quantite commandee du produit 2 :      2
tapez le prix unitaire du produit 2 :     25

tapez la quantite commandee du produit 3 :      1
tapez le prix unitaire du produit 3 :    100

Total commande : 240
Prix TTC : 288
```

8. Les structures conditionnelles

Définition : Structure optionnelle

Une structure optionnelle est constituée d'un bloc d'instructions (une ou plusieurs instructions), que l'ordinateur ne doit exécuter **que si** une condition est vérifiée, **sans aucune autre alternative**. Si la condition n'est pas vraie, le bloc d'instructions est ignoré.

Syntaxe de base :

```
if(condition booléenne) {
    instructions
}
```

- La condition booléenne peut être un booléen, ou un ensemble de booléens combinés avec des opérateurs logiques.
- Pour une bonne lisibilité, l'indentation du code est nécessaire.
- Si un bloc IF ne contient qu'une seule instruction, alors les accolades ne sont pas obligatoires :

```
if(condition booléenne)
    instruction
```

- Un bloc IF peut contenir un autre bloc IF :

```
if(condition booléenne) {
    instructions

    if(condition booléenne) {
        instructions
    }
    instructions
}
```

Définition : Structure alternative IF/ELSE

Une structure alternative IF / ELSE sert à exécuter un bloc d'instructions si une condition est vraie, **sinon on exécute un autre bloc d'instructions**.

Syntaxe de base :

```
if(condition booléenne) {
    bloc 1
} else {
    bloc 2
}
```

- Si un bloc ELSE ne contient qu'une seule instruction, alors les accolades ne sont pas obligatoires :

```
if(condition booléenne)
    instruction1
else
    instruction2
```

Un bloc ELSE peut lui aussi contenir un autre bloc IF, ou IF / ELSE

Définition : Structure alternative SWITCH-CASE

Supposons qu'on soit obligé de faire un test comme celui-là :


```

if(x == 0) {
    instruction1;
} else {
    if(x == 1) {
        instruction2;
    } else {
        if(x == 2) {
            instruction3;
        } else {
            if(x == 3) {
                instruction4;
            } else {
                /* instruction à exécuter par défaut, au cas où x n'égale
                aucune des valeurs 0,1,2 ou 3 */
                instruction5;
            }
        }
    }
}

```

La structure alternative SWITCH – CASE est un moyen élégant et efficace de réaliser le programme précédent d'une manière plus simple.

Syntaxe de base :

```

switch(variable) {
    case valeur_1 :
        traitement_1;
        break;
    case valeur_2 :
        traitement_2;
        break;
    case valeur_3 :
        traitement_3;
        break;
    case .....
    default :
        traitement_par_défaut;
        break;
}

```

- Tout comme les blocs **if** et **else**, un bloc **case** ou **default** peut tout contenir (des instructions, des structures **if**, **if/else**, **switch-case**, ...)
- Dans un bloc **case** ou **default** les accolades ne sont pas obligatoires, **même quand le bloc contient plusieurs instructions.**
- L'instruction **break** sert à sortir de la structure **switch** quand l'un des blocs **case** est vérifié. Mais elle n'est pas obligatoire selon la syntaxe de java.
- Le bloc **default** n'est pas obligatoire.
- Quand plusieurs blocs case contiennent le même traitement, ils peuvent être « rassemblés ». Syntaxe :

```

case valeur_1 :
case valeur_2 :

```

```
case valeur_3 :  
    traitement_3;  
    break;
```

Le test switch, peut être appliqué aux variables de types entier, caractère, ou chaîne de caractère.

Définition : Structure alternative ? :

Utilisée pour les affectations soumises à un test **if/else**.

Syntaxe de base :

```
variable = (condition booléenne) ? valeur_1 : valeur_2;
```

Traduction en utilisant **if/else** :

```
if(condition booléenne)  
    variable = valeur_1  
else  
    variable = valeur_2;
```

Exercice 1

Écrire un programme qui permet de déterminer le plus petit et le plus grand de 3 chiffres x, y et z qui seront lus depuis le clavier.

Exercice 2

Écrire un programme qui affiche le signe du produit de deux nombres A et B sans calculer le produit.

Exercice 3

Écrire un programme qui calcule le salaire brut d'un ouvrier connaissant le nombre d'heures nbr_h (lu depuis le clavier) et le tarif horaire tarif_h (lu depuis le clavier), sachant que les heures supplémentaires (au-delà de 172 heures) seront payées 50% en plus du tarif normal. Le programme doit aussi afficher le salaire net :

Si le salaire brut est sup. ou égal à 7000dh, alors le salaire net est : salaire brut – 30%
Sinon, le salaire net est : salaire brut – 20%.

Exercice 4

Écrire un programme qui lit le nombre d'enfant d'une famille, et qui affiche le montant de l'allocation familiale que doit recevoir cette famille, de cette manière :

- Si la famille ne contient pas d'enfants, aucune allocation
- Entre 1 et 3 enfants, allocation de 150DH
- Entre 4 et 6 enfants, allocation de 250DH
- Plus de 7 enfants, allocation de 350DH

Si le nombre d'enfants est incorrecte, afficher un message d'erreur.

Exercice 5

Un patron décide de calculer le pourcentage de sa participation aux prix des repas de ses employés de la façon suivante :

- Salarié célibataire participation de 20%
- Salarié marié participation de 25%
- Salarié marié ayant des enfants participation de 10% supplémentaires par enfant
- Si le salaire mensuel est inférieur à 5000 DH la participation est majorée de 10%
- La participation est plafonnée à 50%.

Écrire un programme qui lit les informations au clavier et affiche le montant de la participation à laquelle aura droit le salarié.

Exercice 6

Écrire un programme qui lit le mois (en chiffres 1 : Janvier, ..., 12 : Décembre) depuis le clavier, et qui affiche le nombre de jours de ce mois.

Exemples d'exécution :

```
tapez le mois
9
30 jours
```

```
tapez le mois
2
28 ou 29 jours
```

```
tapez le mois
12
31 jours
```

```
tapez le mois
13
valeur incorrecte !
```

9. Les structures itératives

Définition

Une structure répétitive (ou *itérative*, appelée aussi *boucle*) oblige le programme à répéter un bloc d'instructions tant que la condition de répétition est vraie.

Exemple : Somme des entiers de 1 à 10.

On se propose de calculer la somme des nombres entiers entre 1, 2, 3, ..., 10. Il nous faudra donc une variable qui sera nommée **compteur** pour faire le comptage de **1** à **10**, et une autre **som** pour faire la somme.

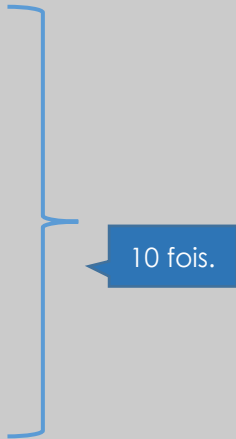
Code sans structure itérative :

```
byte compteur = 1, som = 0;

som += compteur;
compteur++; // compteur vaut 2

som += compteur;
compteur++; // compteur vaut 3

som += compteur;
compteur++; // compteur vaut 4
...
...
...
som += compteur;
compteur++; // compteur vaut 10
System.out.println("valeur de la somme : " + som);
```



La partie répétée sera exécutée tant que **compteur <= 10**

Bien sûr, il n'est pas pratique de répéter ces instructions, d'où l'utilité des **boucles**.

La boucle WHILE

Syntaxe de base :

```
while(condition booléenne) {
    instructions
    incrémentation / décrémentation
}
```

- De même que les autres structures déjà étudiées (if, else, ...), si un bloc **while** ne contient qu'une seule instruction, alors les accolades ne sont pas obligatoires.
- Un bloc peut lui aussi contenir des structures alternatives (if, if/else, switch, ...), itératives, ...
- La boucle **while** continue de s'exécuter tant que la condition booléenne est vraie. Si la condition booléenne est fausse, les instructions de la boucle ne seront pas exécutées.
- Il est essentiel que la condition booléenne devienne fausse après un certain nombre d'itérations, sinon on parle de « boucle infinie ». Une boucle infinie conduit à un plantage du programme.
- L'utilisation de l'instruction **break** permet d'arrêter une boucle, **même si la condition booléenne est encore vraie**.

Syntaxe de base :

```
while(condition booléenne)
{
    instructions
    if(autre condition booléenne)
    {
        break;
    }
    incrémentation / décrémentation
}
```

Exemple : Somme des entiers de 1 à 10.

```
short n = 10;
short i = 1;
short somme = 0;
while(i <= 10) {
    somme += i++;
}

System.out.println("La somme des entiers de 1 à "
    + n + " est : " + somme);
```

La somme des entiers de 1 à 10 est : 55

La boucle DO-WHILE

Syntaxe de base :

```
do {
    instructions
    incrémentation / décrémentation
} while(condition booléenne);
```

- La boucle **do-while** a le même effet que la boucle **while**, sauf que la boucle **do-while** vérifie la condition d'itération à la fin et non au début. Donc même si la condition est fausse, les instructions seront exécutées au moins une fois.
- La boucle **do-while** est toujours suivie d'un point-virgule.

Exemple : Somme des entiers de 1 à 10.

```
short n = 10;
short i = 1;
short somme = 0;
do {
    somme += i++;
} while(i <= 10);

System.out.println("La somme des entiers de 1 à "
    + n + " est : " + somme);
```

La boucle FOR

Syntaxe de base :

```
for(initialisation; condition booléenne; incrémentation/décrémentation) {
    instructions
}
```

- La boucle **for** a le même effet que la boucle **while**.
- La déclaration de la boucle **for** doit **obligatoirement** contenir ces **3** éléments :
 - L'initialisation du compteur (ou bien déclaration et initialisation),
 - Condition booléenne de répétition de la boucle,
 - L'instruction d'incrément ou décrémentation du compteur.

Exemple : Somme des entiers de 1 à 10.

```

short n = 10;
short i;
short somme = 0;
for(i = 1; i <= 10; i++) {
    somme += i;
}

System.out.println("La somme des entiers de 1 à "
    + n + " est : " + somme);

```

Exercice 1

Écrire un programme qui lit **2** entiers **X1** puis **X2**, et qui calcule la somme des entiers compris entre **X1** et **X2**. (La valeur de **X1** doit être inférieure à celle de **X2**, sinon on les échange).

Exercice 2

Écrire un programme qui lit un entier **N** et calcule et affiche son factoriel ($N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1$)

Exercice 3

Écrire un programme qui lit un entier **N** et qui affiche les entiers de **1** à **N**, 5 par 5, séparés par des tabulations.

Exemple d'exécution pour **N = 22** :

```

1 2 3 4 5
6 7 8 9 10
...
21 22

```

...

21 22

Indication : Utiliser l'opérateur modulo (%)

Exercice 4

Écrire un programme qui lit un entier **N** et qui produit l'affichage suivant (en escaliers) :

```

1
 2
 3
 4
.....
  N

```

Exercice 5

Écrire un programme qui lit un entier **N** et qui affiche un carré d'étoiles contenant **N** lignes, dont chacune contient **N** étoiles :

```

**** *
**** *
**** *
...
**** *

```

Ensuite on se propose de réaliser les figures suivantes :

```

*
* *
* * *
* * * *

```

Ou bien :

```

* * * *
* * *
* *
*

```

Ou bien :

```
*  
**  
***  
****  
...
```

10. Les tableaux

1. Tableaux de dimension 1

Définition

Un tableau est aussi une variable, mais en java, il a les spécificités suivantes :

- Il est considéré comme une référence et non pas une variable simple.
- Il peut contenir plusieurs valeurs selon sa taille, et ces valeurs doivent être du même type. (tableau d'entiers, de réels, de chaînes de caractères, ...)
- L'indexation des éléments d'un tableau de taille n commence par 0, puis, 1, ..., le dernier élément a l'indice $n-1$:

Indices : 0 1 2 ... $n-1$

			...	
--	--	--	-----	--

- La taille du tableau est **fixe**, on ne peut ni ajouter ni supprimer des éléments. La seule option possible est la mise à jour d'éléments.
- Chaque tableau possède l'attribut **length**, qui donne la taille du tableau.
- Syntaxe de base :

```
// 1iere syntaxe :
int[] x = new int[taille]; // déclaration & allocation d'espace mémoire
x[indice] = valeur; // initialisation

// 2ieme syntaxe :
int y[] = {10, 10, -1}; // déclaration et initialisation. doivent se faire en une seule
// instruction

int[] x;
x = {15, -24, 5}; // erreur de compilation
```

La boucle FOR pour les tableaux

```
for(double elem : tableau) {
    .....
}
```

Exemple : déclarer puis remplir un tableau d'entiers de taille 5 depuis le clavier

```
int i;
int t[] = new int[5];
Scanner sc = new Scanner(System.in);
for(i = 0 ; i <= 4 ; i += 1){
    System.out.println("tapez un entier");
    t[i] = sc.nextInt();
}
```

Exemple : Déclarer & remplir un tableau nommé B qui contiendra 5 réels, le dernier sera la moyenne des 4 autres

```
double B[] = new double[5];
B[0] = 1251.05556;
B[1] = 0.25;
B[2] = 10;
B[3] = -10;
B[4] = (B[0] + B[1] + B[2] + B[3]) / 4;
```


Exemple : La boucle FOR pour les tableaux

```
int t[] = {-1, 99, 367, 25, 0, 25};
for (int element : t) {
    System.out.print(element + "\t");
}
```

Exercice 1

Demander à l'utilisateur de saisir le nombre d'élèves **N** d'une classe.
 Ensuite demander les notes d'examen de ces **N** élèves, qui seront stockées dans le tableau **notes**. Le programme doit ensuite afficher le nombre d'élèves qui ont eu la moyenne.

Exercice 2

Ecrire un programme qui demande à l'utilisateur de saisir **5** entiers qui seront stockés dans un tableau **T_ENTIER**. Le programme doit ensuite afficher la valeur du plus grand élément de ce tableau.

Indication : stocker le **1^{er}** élément dans une variable **plusGrand**, comparer ensuite tous les autres éléments un par un avec **plusGrand**, chaque fois qu'un élément est supérieur à **plusGrand**, **plusGrand** devient cet élément-là.

Exercice 3

Lire les éléments d'un tableau de chaînes de caractères de taille **5** depuis le clavier, et afficher ensuite celle qui contient le plus de caractères.

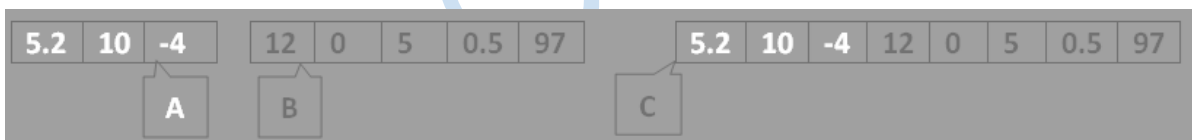
Indication : s'inspirer de l'exercice précédent et utiliser la méthode **length()** de la classe **String**.

Exercice 4

Ecrire un programme qui lit les valeurs de **2** tableaux de chiffres **A** et **B** de tailles **N** et **M** lues au clavier, et qui regroupe les éléments de **A** et **B** dans un tableau **C**.

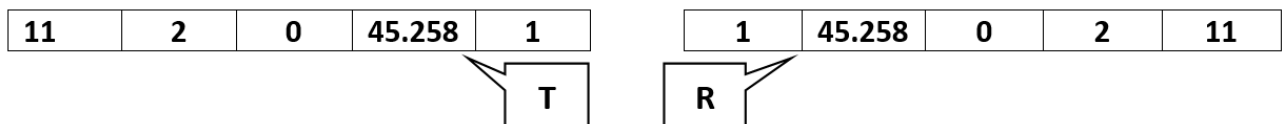
Afficher ensuite les éléments de **C** à la manière de l'exemple **18**.

Exemple :

**Exercice 5**

Ecrire un programme qui demande à l'utilisateur de taper **5** chiffres qui seront stockés dans un tableau **T**, le programme doit remplir un autre tableau **R** avec les éléments de **T** mais à l'envers. Afficher ensuite les éléments de **R**.

Exemple :

**Exercice 6**

Ecrire un programme qui lit une chaîne de caractères **S** et qui teste si oui ou non cette chaîne est un palindrome.

Indications :

- Un palindrome est une chaîne de caractères qui se lit de la même manière aussi bien de droite à gauche que de gauche à droite. Ex. "**kayak**", "**OTO**", ...
- Transférer les caractères de **S** dans un tableau de caractères **T** en utilisant la méthode **toCharArray** de la classe **String**
- Créer un autre tableau **R** et s'inspirer de l'exercice précédent pour chercher si **S** est un palindrome.
- La taille d'un tableau est indiquée par l'attribut de tableaux **length**

Exercice 7

Ecrire un programme qui demande à l'utilisateur de taper **N** entiers qui seront stockés dans un tableau **T**. Le programme doit trier le tableau par ordre croissant et ensuite les afficher.

Indication (algorithme de tri « bulle »):

On parcourt le tableau en comparant **T[0]** et **T[1]** et en échangeant ces éléments s'ils ne sont pas dans le bon ordre.

On recommence le processus en comparant **T[1]** et **T[2]**,... et ainsi de suite jusqu'à **T[N-1]** et **T[N-2]**.

Lors de chaque itération, on compte le nombre d'échanges effectués.

On fait autant d'itérations que nécessaire jusqu'à ce que le nombre d'échanges soit **0** : le tableau est alors trié.

2. Tableaux de dimension 2

Définition

Un tableau à **2** dimensions, ou *matrice*, est une grille de cellules, caractérisée par un nombre de lignes et un nombre de colonnes. Et comme pour les tableaux à dimension **1**, l'indexation des lignes et des colonnes commence par **0**.

Exemple, une matrice de réels, de **4** lignes et **6** colonnes peut être représentée comme ceci :

ligne 0						
ligne 1						
ligne 2						
ligne 3						
colonnes:	0	1	2	3	4	5

- La syntaxe de déclaration d'un tableau à dimensions **2** se fait comme pour celle d'un tableau à dimension **1**, mais avec **deux** paires de crochets, la 1^{ière} pour les lignes, et la 2^{ème} pour les colonnes.
- Syntaxe de base :

```
// 1iere syntaxe :
int[][] x = new int[nbr_lignes][nbr_colonnes]; // déclaration
x[ligne][colonne] = valeur;
// 2ieme syntaxe :
int y[][] = {{10, 10},{5, 5},{1, 1}}; // déclaration et initialisation

int y[][];
y = {{10, 10},{5, 5}}; // erreur de compilation
```

Exemple : Lire les éléments d'un tableau d'entiers de taille [3][3].

```
// lecture des éléments de la ligne 0
System.out.print("tapez un entier\t");
t_int[0][0] = scn.nextInt();
System.out.print("tapez un entier\t");
t_int[0][1] = scn.nextInt();
System.out.print("tapez un entier\t");
t_int[0][2] = scn.nextInt();
```


```
// lecture des éléments de la ligne 1
System.out.print("tapez un entier\t");
t_int[1][0] = scn.nextInt();
System.out.print("tapez un entier\t");
t_int[1][1] = scn.nextInt();
System.out.print("tapez un entier\t");
t_int[1][2] = scn.nextInt();
```


```
// lecture des éléments de la ligne 2
System.out.print("tapez un entier\t");
t_int[2][0] = scn.nextInt();
```


```
System.out.print("tapez un entier\t");
t_int[2][1] = scn.nextInt();
System.out.print("tapez un entier\t");
t_int[2][2] = scn.nextInt();

// même exemple avec les boucles
int l,c;
l = 0;
while(l < 3) {
    c = 0;
    while(c < 3) {
        System.out.print("tapez un entier\t");
        t_int[l][c] = scn.nextInt();
        c++;
    }
    l++;
}
```

Exercice 1

Remplir deux matrices **M1** et **M2** de **2** lignes et **2** colonnes depuis le clavier, et afficher la matrice **S** qui représente leur somme. Exemple :

M1	1	1	M2	4	4	S	5	5
	1	1		4	4		5	5

Exercice 2

Remplir une matrice **A** comme ceci :

1	1	1	1	1	5 1 ^{ères} puissances de 1
2	4	8	16	32	5 1 ^{ères} puissances de 2 ($2^1, 2^2, 2^3, \dots$)
3	9	27	81	243	5 1 ^{ères} puissances de 3 ($3^1, 3^2, 3^3, \dots$)
4	16	64	256	1024	5 1 ^{ères} puissances de 4 ($4^1, 4^2, 4^3, \dots$)

Afficher ensuite la moyenne de chaque ligne comme ceci :

Moyenne de la ligne 1 : 1.0
 Moyenne de la ligne 2 : 12.4
 Moyenne de la ligne 3 : 72.6
 Moyenne de la ligne 4 : 272.8

Exercice 3

Ecrire un programme qui remplit une matrice **M** de **3** lignes et **3** colonnes et qui décale les lignes de **M** d'une position vers le bas. Afficher ensuite **M**.
 Exemple :

1	1	1	
4	4	4	
3	3	0	

→

3	3	0
1	1	1
4	4	4

Exercice 4

Ecrire un programme qui lit un entier **n** et qui affiche le **triangle de Pascal** d'ordre **n** de la manière suivante : (utiliser un tableau d'entiers)

Tapez l'ordre du triangle : 5

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
  
```

11. Les chaines de caractères

1. Caractères & chaines de caractères

Caractère / Chaine de caractère

1- caractères : le type **char**
 char x = 'A', y = '\n';
 2- chaines de caractères : la classe **String**
 String x = "ABC", y = "";
 Ou bien :
 String x = new String("ABC");
 3- afficher les caractères spéciaux
 \ ' Apostrophe
 \ " Double-quotes
 \\ Antislash
 \t Tabulation
 \b Retour arrière (backspace)
 \r Retour chariot
 \f Saut de page (form feed)
 \n Saut de ligne (newline)

Exemple

```
System.out.println("\ABC efgh\" + '\n' + '\r' + "D");
System.out.println("");

System.out.println("\ABC efgh\" + '\r' + "D");
System.out.println("");

System.out.println("ABC" + '\b' + "D");
System.out.println("");
```

```
run:
"ABC efgh"
D

D

ABD
```

BUILD SUCCESSFUL (total time: 0 seconds)

Obtenir la longueur d'une chaine de caractère : fonction length()

```
String message = "bonjour tout le monde";
System.out.println(message.length()); // affiche 21
```

La concaténation

La concaténation entre chaine de caractères et autres données se fait par l'opérateur +

Exemple

```
String message = "bonjour, tapez votre nom";
System.out.println(message);
String nom = new Scanner(System.in).nextLine();
message = "Votre nom \" + nom + "\" contient " + nom.length() + " caracteres.";
System.out.println(message);
```

```
bonjour, tapez votre nom
ELYAHYAOU
Votre nom "ELYAHYAOU" contient 10 caracteres.
```

2. Extraire un nombre depuis une chaîne de caractères

Les méthodes parse

```
String ch1 = "un nombre entier";
int nbr1 = Integer.parseInt(ch1); // extraire un nombre entier d'une chaîne
String ch2 = "un nombre réel";
double nbr2 = Double.parseDouble(ch2); // extraire un nombre réel d'une chaîne
```

Exemple

```
String ch1 = "10";
int nbr1 = Integer.parseInt(ch1);
String ch2 = "5.05";
double nbr2 = Double.parseDouble(ch2);
System.out.println("1ier test : " + (ch1 + ch2));
System.out.println("2ieme test : " + (nbr1 + nbr2));

1ier test :105.05
2ieme test :15.05
```

3. Chercher / remplacer une sous-chaîne dans une chaîne

Les méthodes startsWith / endsWith / contains

Retournent **true** si une chaîne commence par / se termine par / contient une autre.

Exemple

```
boolean b1 = "ELYAHYAOU".startsWith("ELY"); // true
boolean b1 = "ELYAHYAOU".endsWith("ELY"); // false
boolean b1 = "ELYAHYAOU".contains("ELY"); // true
```

La méthode subString

```
String ch2 = ch1.substring(debut)
/* Retourne dans une autre chaîne ch2, la partie de ch1 qui commence depuis la position
debut. L'indexation commence par 0. */
String ch2 = ch1.substring(debut,fin)
/* Retourne dans une autre chaîne ch2, la partie de ch1 qui commence depuis la position
debut et qui se termine à la position fin-1. L'indexation commence par 0. */
```

Exemple

```
String nom = "ELYAHYAOU";
System.out.println("1ier test : " + nom.substring(2));
System.out.println("2ieme test : " + nom.substring(2,3));
System.out.println("3ieme test : " + nom.substring(2,4));
System.out.println("4ieme test : " + nom.substring(7,10));

1ier test :YAHYAOU
2ieme test :Y
3ieme test :YA
4ieme test :OUI
```

Les méthodes replace et replaceFirst

```
String nom = "ELYAHYAOU";
System.out.println("1ier test : " + nom.replace("YA", "ya"));
System.out.println("2ieme test : " + nom.replaceFirst("YA", "ya"));

1ier test :ELyAHyaOU
2ieme test :ELyAHYAOU
```

4. Comparer deux chaînes de caractères

Changer la casse vers majuscule / minuscule

```
// retourner une chaîne qui correspond à la même chaîne en minuscules
```

```
String ch2 = ch1.toLowerCase();
// action inverse
String ch2 = ch1.toUpperCase();
```

Exemple

```
String nom = "ELYAHYAOU";
System.out.println("test :" + nom.toLowerCase());

test :elyahyaoui
```

Comparer deux chaînes

```
// comparer sans tenir compte de la casse
boolean b1 = "ELYAHYAOU".equalsIgnoreCase("ElYahyAOui"); // true
// comparer en tenant compte de la casse
boolean b1 = "ELYAHYAOU".equals("ElYahyAOui"); // false
```

Tester si une chaîne est vide

```
if(chaine != null) {
    if(chaine.isEmpty()) {
        // la chaîne est vide
    }
}
```

Enlever les espaces des deux côtés

```
String ch1 = ch2.trim();
```

5. Les chaînes de caractères modifiables

La classe StringBuilder

Rappel : une chaîne de caractères est un **objet** et non une variable primitive. Les objets String ont une particularité de plus par rapport aux autres : lors **de chaque modification** d'une chaîne de caractères, un nouvel objet est créé et alloué à la référence de cette chaîne, tandis que l'objet courant est abandonné. Ceci génère un grand nombre d'objets non utilisés, ce qui constitue une mauvaise gestion de l'espace mémoire. Ce sont les chaînes de caractères **StringBuilder** et **StringBuffer** qui sont utilisées dans les cas où on a besoin d'effectuer plusieurs modifications (ex. : multiple concaténation).

Exemple : Composer une requête SQL

```
String nomTable = "CLIENT ";
String infos = " id,ville,age ";
String critere1 = " nom = 'ELYAHYAOU' ";
String critere2 = " profession = 'ENSEIGNANT' ";

StringBuilder requete = new StringBuilder();
requete.append("select");
requete.append(infos);
requete.append("from");
requete.append(nomTable);
requete.append("where");
requete.append(critere1);
requete.append("and");
requete.append(critere2);

System.out.println(requete.toString());

select id,ville,age from CLIENT where nom = 'ELYAHYAOU' and profession = 'ENSEIGNANT'
```


12. Les dates

1. Obtenir une date ou une heure

Obtenir la date

La date et le temps sont gérés par les classes **LocalDate**, **LocalDateTime** et **LocalTime** du package **java.time**.

Syntaxe de base de l'utilisation de ces classes :

- Obtenir la date courante

```

LocalDate d = LocalDate.now();
System.out.println(d);
// format de l'affichage : 1983-07-30

```

- Obtenir la date & heure courante

```

LocalDateTime dt = LocalDateTime.now();
System.out.println(dt);
// format de l'affichage : 1983-07-30T01:30:00.781

```

- Obtenir l'heure courante (format 24H)

```

LocalTime t = LocalTime.now();
System.out.println(t);
// format de l'affichage : 01:30:00.781

```

- Obtenir l'une des composantes d'une date (l'année, le jour, l'heure, ...)

Utilisation des méthodes `getSecond`, `getMinute`, `getHour`, `getDayOfWeek`, `getDayOfMonth`, ...

```

LocalDateTime dt = LocalDateTime.now();
System.out.println(dt);

```

```

System.out.println(dt.getDayOfMonth());
System.out.println(dt.getDayOfWeek());
System.out.println(dt.getDayOfYear());
System.out.println(dt.getMonth());
System.out.println(dt.getYear());
System.out.println(dt.getHour());
System.out.println(dt.getMinute());

```

```

2018-10-25T09:31:23.130
25
THURSDAY
298
OCTOBER
2018
9
31

```



Un objet **LocalDateTime** possède toutes les composantes, mais un objet **LocalTime** ne possède pas la méthode `getYear()` par ex., un objet **LocalDate** ne possède pas la méthode `getMinute()`, etc.

Obtenir une date spécifique

```

LocalDate d = LocalDate.of(annee, Month.mois, jour);
// annee et jour : entiers
// Month.mois : nom du mois en anglais, prédéfini dans la classe Month.
// la méthode of est valable aussi pour LocalTime et LocalDateTime.

```

Exemple

```
LocalDate d = LocalDate.of(1983, Month.JULY, 30);
System.out.println(d);
```

```
1983-07-30
```

2. Formater une date**Définition**

Le « formatage » d'une date consiste à obtenir une chaîne de caractères qui correspond à cette date sous un format spécifique (jour en lettres / chiffres, mois en lettres / chiffres, ...), et selon les paramètres linguistiques d'une langue spécifique.

Étapes à suivre :

0- créer un objet `LocalDateTime` (ou `LocalDate` ou `LocalTime`)

```
LocalDateTime dt = LocalDateTime.now();
```

1- créer un objet de type `DateTimeFormatterBuilder`.

```
DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
```

2- définir le format souhaité

```
builder.appendPattern("chaîne formatée");
```

Exemples : (voir la documentation javadoc de la méthode `appendPattern`)

M : le mois en chiffres

m : les minutes

MMMM : le mois en lettres

EEEE : le jour en lettres

Etc.

Symbol	Meaning	Presentation	Examples
-----	-----	-----	-----
G	era	text	AD; Anno Domini; A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	week-of-month	number	3
a	am-pm-of-day	text	PM
h	clock-hour-of-am-pm (1-12)	number	12
K	hour-of-am-pm (0-11)	number	0
k	clock-hour-of-am-pm (1-24)	number	0

3- créer un objet de type `DateTimeFormatter` sur la base du *builder*

```
DateTimeFormatter formatter = builder.toFormatter(Locale.une_Langue);
```

4- formater un objet `LocalDateTime` (ou `LocalDate` ou `LocalTime`) avec l'objet *formatter*

```
dt.format(formatter);
```

Exemple

```
LocalDateTime dt = LocalDateTime.now();
DateTimeFormatterBuilder builder = new DateTimeFormatterBuilder();
builder.appendPattern("EEEE, d MMMM Y - k:m");
DateTimeFormatter formatter = builder.toFormatter(Locale.FRENCH);
System.out.println(dt.format(formatter));
```

```
jeudi, 25 octobre 2018 - 21:16
```

3. Arithmétique des dates

Comparer deux dates

```
LocalDateTime dt = LocalDateTime.now();
LocalDateTime dt2 = LocalDateTime.of(1983, Month.JULY, 30, 01, 0);
System.out.println(dt.isEqual(dt2));
System.out.println(dt.isAfter(dt2));
System.out.println(dt.isBefore(dt2));
```

```
false
true
false
```

Ajouter des valeurs à une date

Utiliser les méthodes **plusDays**, **plusMonths**, **plusHours**, **plusYears**, ... et **minusDays**, **minusMonths**, ... selon le type d'objet `LocalTime`, `LocalDate` ou `LocalDateTime`.

Exemple

```
LocalDate dt = LocalDate.of(1983, Month.JULY, 30);
System.out.println("test 1 : " + dt.plusDays(2));
LocalDate dt2 = dt.plusWeeks(1);
System.out.println("test 2 : " + dt2);
System.out.println("test 3 : " + dt.isAfter(dt2));
dt2 = dt.minusMonths(2);
System.out.println("test 4 : " + dt2);
System.out.println("test 5 : " + dt.isBefore(dt2));
```

```
test 1 : 1983-08-01
test 2 : 1983-08-06
test 3 : false
test 4 : 1983-05-30
test 5 : false
```

On peut aussi utiliser la classe `java.time.temporal.ChronoUnit` :

```
public static void main() {
    LocalDate dt = LocalDate.of(1983, Month.JULY, 30);
    System.out.println(dt);
    LocalDate dt2 = dt.plus(2, ChronoUnit.DAYS);
    System.out.println(dt2);
    dt2 = dt.minusMonths(2);
    System.out.println(dt2);
    System.out.println(dt.isBefore(dt2));
    dt.plus(2, ChronoUnit.DAYS);
}
```

Unit that represents the concept of a decade. For the ISO calendar system, it is equal to 10 years.

When used with other calendar systems it must correspond to an integral number of days and is normally an integral number of years.

CENTURIES	ChronoUnit
DAYS	ChronoUnit
DECADES	ChronoUnit
ERAS	ChronoUnit
FOREVER	ChronoUnit
HALF_DAYS	ChronoUnit
HOURS	ChronoUnit
MICROS	ChronoUnit
MILLENNIA	ChronoUnit
MILLIS	ChronoUnit
MINUTES	ChronoUnit
MONTHS	ChronoUnit
NANOS	ChronoUnit
SECONDS	ChronoUnit
WEEKS	ChronoUnit
YEARS	ChronoUnit

CHAPITRE 2. LA POO EN JAVA

1. Notions de Classes et Objets

1. Définitions et syntaxe

Notion de Classe / Objet

▪ Classe

Tout comme les structures de données en langage C, une classe en POO désigne une catégorie d'individus ayant tous les mêmes propriétés (appelées « attributs ») et les mêmes comportements (appelées « méthodes » ou « opérations »).

▪ Objet

Chaque individu appartenant à une classe est appelé un « objet ». Chaque objet définit ses propres valeurs pour les attributs.

Exemple

Tous les objets de la classe **Voiture** ont les mêmes propriétés :

- Marque
- Pays
- Modèle
- Année
- Couleur
- Matricule
- Energie
- ...

Et les mêmes comportements :

- Démarrer
- Accélérer
- Freiner
- ...

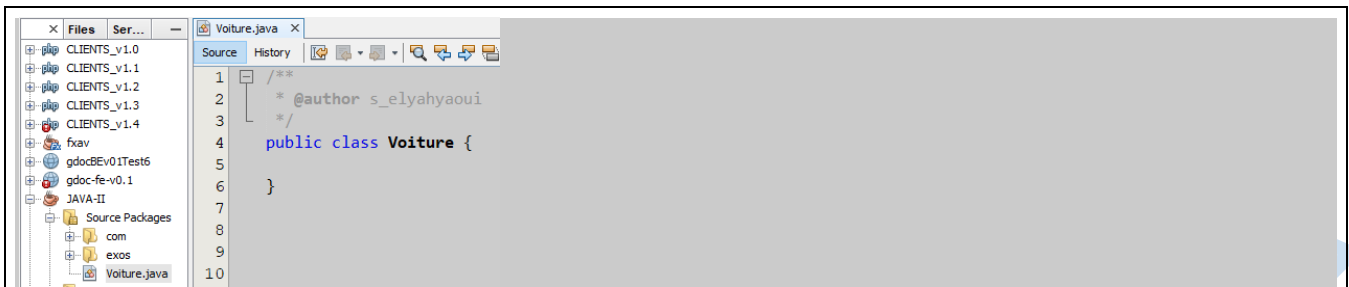
Syntaxe

▪ Classe

- Syntaxe de base de création d'une classe :

```
class nom_classe {
    // déclaration des attributs et méthodes
}
```

- On peut créer plusieurs classes dans un seul fichier Java.
- La classe qui porte le nom du fichier doit être déclarée avec le mot-clef « **public** ».



- Objets

- Création d'objet : les objets sont créés avec l'opérateur **new** qui fait l'appel du **constructeur** de la classe : une méthode qui porte le même nom de la classe.

```
Voiture v1 = new Voiture();
```

- Pour les classes qui ne contiennent pas de constructeur(s), le compilateur java appelle le « constructeur par défaut ».
- Chaque appel de l'opérateur **new** produit la création en espace mémoire d'un objet, **dont l'adresse est stockée dans la référence.**

La référence

L'objet

```
Voiture v1 = new Voiture();
```

- o Affectation d'objets : quand on effectue une affectation d'objet, c'est la référence (l'adresse) de l'objet qui est copiée dans la deuxième référence, et non l'objet lui-même.

```
Voiture v1 = new Voiture();
```

```
Voiture v2 = v1; // v1 et v2 pointent vers le même objet.
```

2. Déclaration des attributs et méthodes

Syntaxe

- Attributs

```
class nom_classe {
    type nom_attribut1 [= valeur_par_defaut];
    type nom_attribut2 [= valeur_par_defaut];
    .....
}
```

La valeur par défaut n'est pas obligatoire, mais si elle est spécifiée, l'attribut aura cette valeur dès la création de chaque objet.

- Si on n'indique pas la valeur par défaut d'un attribut, java affectera la valeur par défaut adéquate à chaque type d'attribut :

char Le caractère vide

0 nombres (int, double, ...)

références (comme String)	null
---------------------------	------

- Les attributs d'un objet sont accessibles par l'opérateur **point**.

- Méthodes :

- Syntaxe générale :

```
class nom_classe {
    // attributs
    type nom_attribut1 [= valeur_par_defaut];
    type nom_attribut2 [= valeur par defaut];
```

```
// méthodes
type_retour nom_methode1(type1 param1, type2 param2, ...);
type_retour nom_methode2(type1 param1, type2 param2, ...);
.....
}
```

- Même si une méthode ne prend aucun paramètre, les parenthèses sont obligatoires.
- Le nom + la liste des paramètres constituent ce qu'on appelle « **la signature** » de la méthode.
- Les méthodes qui ne retournent rien doivent spécifier le type **void** (vide) dans **type_retour**.
- Les méthodes d'un objet sont aussi accessibles par l'opérateur **point**.
- Dans toutes les méthodes d'un objet, ce dernier se référence lui-même par l'opérateur **this**.

3. Le constructeur

Définition

- Méthode qui sert à construire des objets d'une classe.
- Le constructeur doit porter le même nom que la classe, et **ne doit pas spécifier un type de retour**.
- Une classe peut avoir plusieurs constructeurs, ayant chacun une signature différente.
- Même si une classe ne définit pas explicitement son constructeur, elle a le constructeur par défaut.
- Si une classe définit au moins un seul constructeur, elle n'a plus droit au constructeur par défaut.
- Les types de constructeurs :

```
class Voiture {
    String marque, pays, modele;

    // constructeur sans paramètres
    Voiture() {

    }

    // constructeur avec paramètres
    Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
    }

    // constructeur par recopie
    Voiture(Voiture v) {
        this.marque = v.marque;
        this.pays = v.pays;
        this.modele = v.modele;
    }
}
```

TP : Gestion des commandes

- Créer un projet nommé « commandes ».
 - Créer un package et qui va contenir les classes suivantes :
 - Client : id, nom, adresse, tel
 - Commande : id, date, type (express / normale), payee (0 / 1), client
 - Produit : id, designation, prixUnit, qteStock
 - LigneCommande : produit, commande, qteCommandee, reduction (réel qui représente le pourcentage de la réduction)
 - GestionCommande : classe qui contiendra la fonction main().
 - La classe LigneCommande doit contenir la méthode calculTotalProduit() qui doit calculer et retourner le total du produit.
 - La classe Commande doit contenir la méthode calculTotalCommande() qui doit calculer et retourner le total de la commande.
- Dans le Main() :
- Créer un client puis une commande dont les infos : id, date, type (express / normale), payee (0 / 1) seront lues depuis le clavier. Cette commande sera liée au client créé précédemment.
 - Demander aussi les infos du client de cette commande.
 - Demander à l'utilisateur le nombre de produits de la commande.
 - Créer un tableau d'objets qui va stocker les infos de tous les produits commandés (id, designation, prixUnit, qteStock, qteCommandee et réduction). Ces infos doivent être lues depuis le clavier.
 - Calculer et afficher les détails de la commande comme ceci :

```

BON DE COMMANDE N°2605

DATE :      JEUDI 29/11/2018
CLIENT :    ELYAHYAOU

ARTICLES :
  DESIGNATION                QTE    P.U.    REDUCT    TOTAL
* LAPTOP S.VAIO SVF15N2Y    1      13000     5%      12350
* ADAPTER VGA/HDMI S.VAIO    1       100      0%       100

TOTAL :      12450
TOTAL TTC :  14940

```

2. Les membres statiques

1. Les attributs et méthodes statiques

Définition

- Dans la classe Client par ex., les attributs **nom**, **prénom**, **adresse**, ... et les méthodes **sePresenter()**, **afficherMesCommandes()**, ... sont appelés attributs et méthodes « d'objet », car ils changent d'un objet à un autre, ils n'existent donc pas sans objets.
- Par contre, un attribut dit « statique » présente les différences suivantes :
 - Il n'appartient pas à l'objet, mais à toute la classe,
 - Sa valeur est la même pour tous les objets,
 - Ne nécessite pas l'instanciation d'objets pour être utilisé.
- Syntaxe : déclaration normale + ajout du mot-clef « **static** »
- Exemple :

```
class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;
}
```

2. Le bloc STATIC

Définition

- Syntaxe :

```
class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;
    static {
        // instructions
    }
}
```

- Le bloc statique est une partie de la classe qui est semblable à une méthode, mais qui ne s'exécute **qu'une seule fois**, lors de la toute première utilisation de la classe, c.à.d. : le premier appel d'un attribut ou méthode statique, ou la création du premier objet.
- Le bloc statique est souvent utilisé pour :
 - Initialiser des attributs statiques, ou,
 - Effectuer des traitements qui doivent être faits avant de commencer à utiliser des objets de la classe.

Exemple

```
public class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
        nombreVoitures++;
        System.out.println("CREATION DE L'OBJET " + nombreVoitures);
    }

    static {
        nombreVoitures = 0;
        System.out.println("EXECUTION DU BLOC STATIC");
    }
}
```



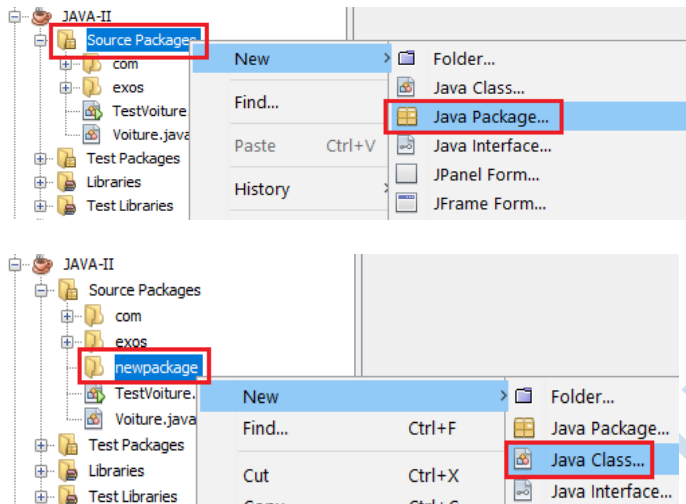
```
public class TestVoiture {  
    public static void main(String[] args) {  
        Voiture v1 = new Voiture("", "", "");  
        Voiture v2 = new Voiture("", "", "");  
        Voiture v3 = new Voiture("", "", "");  
        Voiture v4 = v3;  
    }  
}
```

```
EXECUTION DU BLOC STATIC  
CREATION DE L'OBJET 1  
CREATION DE L'OBJET 2  
CREATION DE L'OBJET 3
```

3. Les packages et l'Encapsulation

Les packages

- Un package est un dossier qui regroupe des classes, des ressources (fichiers textes, fichiers de configuration, fichiers audio, ...), ... qui sont utilisés à une fin précise (lecture / écriture sur des fichiers, connexion à une base de données, ...).
- Selon les normes de Java, toute classe doit être logée dans un package spécifique, et non dans le package par défaut, qui est le package **source** du projet.
- Créer une classe dans package :



- Chaque fichier java appartenant à un package (à part le package source), doit déclarer le nom du package avant la déclaration de la classe.

```
package newpackage;

public class NewClass {

}
```

Le principe d'encapsulation

▪ Scenario :

- Un objet de la classe Conducteur (conducteur de voitures) possède les méthodes **conduire()**, **stationner()** et **payerHorodateur()**.
- La méthode **stationner()** fait l'appel de la méthode **payerHorodateur()** pour payer le stationnement, et la méthode **payerHorodateur()** doit utiliser l'attribut **argentPoche**.
- Avant que l'objet Conducteur fasse l'appel de sa méthode **stationner()**, un autre objet X accède en écriture à l'attribut **argentPoche** et le remet à zéro.
- L'objet Conducteur fait l'appel de sa méthode **stationner()** qui fait l'appel de la méthode **payerHorodateur()**, et là, problème : l'attribut **argentPoche** n'est pas suffisant pour garantir le bon fonctionnement de la méthode.

▪ Analyse :

- Les méthodes d'un objet A utilisent les attributs de cet l'objet. Donc si un autre objet B modifie la valeur de l'un des attributs de A, alors les méthodes de ce dernier pourraient ne pas fonctionner correctement.
- Donc pour bien fonctionner, les objets doivent avoir un mécanisme qui interdit l'accès libre (surtout en écriture) à leurs attributs, et qui le remplace par un accès « personnalisé » ou « contrôlé ».

- Ce mécanisme s'appelle **l'encapsulation**. Et impose aux classes de définir des méthodes appelées « accesseurs » ou bien « *getters and setters* », qui vont contrôler l'accès aux attributs des objets en lecture et en écriture.
- Exemple :

Sans encapsulation

```
public class Conducteur {
    public double argentPoche;
}
```

Avec encapsulation

```
public class Conducteur {
    private double argentPoche;

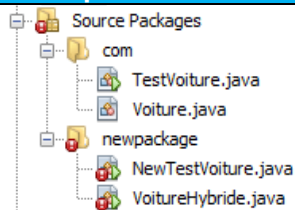
    public getArgentPoche() {
        return argentPoche;
    }

    public setArgentPoche(double argent) {
        if(this.argentPoche < argent)
            this.argentPoche = argent;
    }
}
```

« Niveaux de visibilité » ou « Modificateurs d'accès »

- Pour que l'encapsulation fonctionne, il faut « restreindre » l'accès aux attributs (et aux méthodes) à l'aide de ce qu'on appelle des « modificateurs d'accès » :
 - **public** : visible partout, ce qui veut dire que l'accès – à cet attribut ou cette méthode – est libre pour toutes les autres classes, pour la lecture et l'écriture.
 - **package** :
 - Visible pour toutes les classes qui sont **dans le même package**.
 - C'est le niveau de visibilité par défaut, **on ne l'écrit pas explicitement**.
 - **protected** : idem que **package**, sauf que les classes filles peuvent hériter d'un membre déclaré **protected**, **mais pas d'un membre déclaré package**. Les notions de classes filles et d'héritage feront l'objet du paragraphe 5.
 - **private** : invisible pour toutes les autres classes, c'est le niveau le plus restreint.

Exemple : Modificateur « protected »



```
// package "com"
package com;

public class Voiture {
    protected String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
        nombreVoitures++;
        System.out.println("CREATION DE L'OBJET " + nombreVoitures);
    }

    static {
        nombreVoitures = 0;
        System.out.println("EXECUTION DU BLOC STATIC");
    }
}
```

```

package com;

public class TestVoiture {

    public static void main(String[] args) {
        Voiture v1 = new Voiture("", "", "");
        System.out.println(v1.marque);
    }
}

// package "newpackage"
package newpackage;

import com.Voiture;

public class NewTestVoiture {
    public static void main(String[] args) {
        Voiture v1 = new Voiture("", "", "");
        System.out.println(v1.marque);
    }
}

package newpackage;

import com.Voiture;

public class VoitureHybride extends Voiture {
    public static void main(String[] args) {
        Voiture v1 = new Voiture("", "", "");
        System.out.println(v1.marque);
        VoitureHybride vh = new VoitureHybride("", "", "");
        vh.marque = "LARAKI";
    }
}

```

La méthode main() de la classe TestVoiture peut voir l'attribut « marque » d'un objet Voiture sans problème, car les deux classes sont dans le même package.

La classe NewTestVoiture n'a pas le droit d'accéder à l'attribut « marque » car il est protégé.

marque has protected access in Voiture

(Alt-Enter shows hints)

Même la classe VoitureHybride qui hérite de la classe Voiture (extends) n'a pas le droit d'accéder à l'attribut « marque » d'un objet Voiture car il est protégé, et ce même si elle hérite de cet attribut en tant que classe fille de la classe Voiture.

marque has protected access in Voiture

(Alt-Enter shows hints)

Exemple : Modificateur par défaut « package »

```

// package "com"
public class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
        nombreVoitures++;
        System.out.println("CREATION DE L'OBJET " + nombreVoitures);
    }

    static {
        nombreVoitures = 0;
        System.out.println("EXECUTION DU BLOC STATIC");
    }
}

package com;

public class TestVoiture {
    public static void main(String[] args) {
        Voiture v1 = new Voiture("", "", "");
        System.out.println(v1.marque);
    }
}

// package "newpackage"

```

Idem que pour « protected », la classe TestVoiture peut voir l'attribut « marque » d'un objet Voiture sans problème, car les deux classes sont dans le même package.

```
package newpackage;

import com.Voiture;

public class NewTestVoiture {
    public static void main() {
        Voiture v1 = new Voiture();
        System.out.println(v1.marque);
    }
}
```

marque is not public in Voiture; cannot be accessed from outside package

```
package newpackage;

import com.Voiture;

public class VoitureHybride extends Voiture {

    public static void main(String[] args) {
        Voiture v1 = new VoitureHybride();
        System.out.println(v1.marque);
        Voiture v2 = new VoitureHybride();
        v2.marque = "LARAKI";
    }
}
```

marque is not public in Voiture; cannot be accessed from outside package

La classe VoitureHybride qui hérite de la classe Voiture (*extends*) ne peut ni accéder à l'attribut « marque » d'un objet Voiture, ni hériter de cet attribut dans ses propres objets.

4. Passage de paramètres à une méthode

Paramètre de type « simple »

- Lors du passage d'un paramètre simple à une méthode, celle-ci n'a pas le droit de modifier la valeur de ce paramètre, elle utilise donc une copie de la valeur.

Paramètre de type « référence »

- Lors du passage d'un paramètre objet à une méthode, c'est la référence de l'objet qui est manipulée, la méthode pourra donc modifier les attributs de l'objet, sauf si l'objet ne le permet pas (principe d'encapsulation).
- Le bloc statique est souvent utilisé pour :
 - Initialiser des attributs statiques, ou,
 - Effectuer des traitements qui doivent être faits avant de commencer à utiliser des objets de la classe.

Exemple 1 : paramètre de type simple

```
static void mise_a_zero(double a) {
    a = 0;
    System.out.println("pendant l'appel de mise_a_zero - " + a);
}

public static void main(String[] args) {
    double d1 = 15.05;
    System.out.println("avant l'appel de mise_a_zero - " + d1);
    mise_a_zero(d1);
    System.out.println("apres l'appel de mise_a_zero - " + d1);
}
```

```
avant l'appel de mise_a_zero :      15.05
pendant l'appel de mise_a_zero :    0.0
apres l'appel de mise_a_zero :      15.05
```

Exemple 2 : paramètre de type référence

```
static void mise_a_zero(Voiture a) {
    a.setMarque("");
    a.setModel("");
    a.setPays("");
}

public static void main(String[] args) {
    Voiture v1 = new Voiture("VOLVO", "SUEDE", "C60");
    mise_a_zero(v1);
    System.out.println("apres l'appel de mise_a_zero : "
        + v1.getMarque() + " - " + v1.getPays() + " - " + v1.getModel());
}
```

```

public class Voiture {
    private String marque, pays, modele;
    static int nombreVoitures;
    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
        nombreVoitures++;
        System.out.println("CREATION DE L'OBJET " + nombreVoitures);
    }
    public String getMarque() {
        return marque;
    }
    public void setMarque(String marque) {
        this.marque = marque;
    }
    public String getPays() {
        return pays;
    }
    public void setPays(String pays) {
        this.pays = pays;
    }
    public String getModele() {
        return modele;
    }
    public void setModele(String modele) {
        this.modele = modele;
    }
    public static int getNombreVoitures() {
        return nombreVoitures;
    }
    public static void setNombreVoitures(int nombreVoitures) {
        Voiture.nombreVoitures = nombreVoitures;
    }

    static {
        nombreVoitures = 0;
        System.out.println("EXECUTION DU BLOC STATIC");
    }
}

```

```

EXECUTION DU BLOC STATIC
CREATION DE L'OBJET 1
apres l'appel de mise_a_zero : - -

```

TP : Gestion des commandes v1.1

- Dans le projet « commandes », créer deux packages **stock** et **facture**.
- Les classes Client, Commande, Produit et LigneCommande doivent être déplacées vers le package **stock**.
- La classe GestionCommande doit être déplacée vers le package **facture**.
- Tous les attributs doivent devenir privés. Gérer les changements de code en conséquence.

5. Les classes abstraites et l'Héritage

1. Les classes abstraites

Définition

▪ Scenario :

- Supposons que nous avons les classes **Vehicule**, **Voiture**, **SemiRemorque** et **Moto**. Toutes ces classes possèdent la méthode **demarrer()**.
- On **ne peut pas** définir le code de cette méthode dans la classe **Vehicule**, car son comportement dans cette classe est **abstrait**, vu qu'on ne sait pas de quel **sous-type** de véhicule il s'agit.
- On dit alors que **Vehicule** est le type **générique**, et que **Voiture**, **SemiRemorque** et **Moto** sont des types **spécifiques**, ou des **sous-types** du type **Vehicule**.
- Si on descend vers les types spécifiques, on pourra alors définir la méthode **demarrer()** car elle deviendra parfaitement **concrète**.

▪ Règles & définitions – 1 :

- La syntaxe de déclaration d'une méthode abstraite :

```
abstract type_retour nom_methode(paramètres);
```

- Une méthode abstraite **ne peut pas être déclarée privée**. Les autres modificateurs d'accès sont permis.
- Une classe qui contient au moins une méthode abstraite, est nécessairement abstraite elle aussi.

```
abstract class Vehicule {
    abstract void demarrer();
}
```

- Une classe peut être abstraite sans avoir aucune méthode abstraite.
- Une classe abstraite peut avoir un (ou plusieurs) constructeur, mais **elle ne peut pas être instanciée**.

2. Le principe d'héritage

Règles & définitions – 2

- L'héritage est une relation entre classe fille ou spécifique et classe mère ou générique.
- Cette relation est produite par le mot-clef **extends**. Syntaxe :

```
public class Voiture extends Vehicule {
    ...
}
```

classe fille.
classe mère.

- Quand une classe A1 hérite d'une autre classe A2, alors tous les attributs et méthodes publiques et protégés de A2 sont transmis vers A1, pas besoin de les redéclarer. Les autres membres déclarés **private** et **package** **ne sont pas hérités**.

```
/* Rappel : classes Voiture et VoitureHybride
```

```
Les attributs déclarés package dans la classe Voiture ne sont pas hérités dans sa classe fille VoitureHybride. */
```



```

public class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
        nombreVoitures++;
        System.out.println("CREATION DE L'OBJET " + nombreVoitures);
    }

    static {
        nombreVoitures = 0;
        System.out.println("EXECUTION DU BLOC STATIC");
    }
}

```

```
package newpackage;
```

```
import com.Voiture;
```

```
public class VoitureHybride extends Voiture {
```

```

    public static void main(String[] args) {
        Voiture
        System
        Voiture
        vh.marque = "LAKAKI";
    }

```

La classe VoitureHybride ne peut pas hériter de cet attribut `package` car il est invisible pour elle, mais il sera hérité si la classe est déplacée vers le même package que sa classe mère.

- Le langage Java permet l'héritage transitif : Quand une classe A1 hérite d'une classe A2, qui hérite elle-même d'une classe A3, alors tous les attributs et méthodes publiques et protégés de A2 et A3 sont transmis vers A1.
- Le langage Java **ne permet pas** l'héritage multiple : On ne peut pas hériter les classes – par ex. – Vehicule, Personne et Maison dans une seule classe.
- Si dans la classe fille un attribut est déclaré avec le même nom qu'un attribut hérité depuis la classe mère, alors le nouvel attribut **remplacera** ou **cachera** l'attribut d'origine.

```

public class VoitureHybride extends Voiture {
    private String marque;
    private double capaciteElectrique;
}

```

`capaciteElectrique` représente le kilométrage maximal qu'on peut faire en mode électrique. Cet attribut est **spécifique aux objets VoitureHybride** et n'existe pas dans les objets Voiture.

- Si la classe mère définit **un ou plusieurs constructeurs avec paramètres**, alors au niveau de la classe fille, **une erreur de compilation sera signalée** si ces conditions ne sont pas réunies :
 - la classe fille **est obligée de définir elle aussi au moins l'un des constructeurs** de la classe mère,
 - ce constructeur **doit faire l'appel de celui de la classe mère**, en utilisant l'opérateur **super**,
 - dans ce constructeur, l'appel du constructeur de la classe mère **doit être la première instruction**.
- La règle précédente **n'est plus valable** si la classe mère ajoute **un constructeur sans paramètres**.

```
// classe mère : Voiture
```

```

public class Voiture {
    String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
    }
}

```

```
// classe fille : VoitureHybride
```

```
public class VoitureHybride extends Voiture {
```

constructor Voiture in class Voiture cannot be applied to given types:
required: String,String,String
found: no arguments
reason: actual and formal argument lists differ in length

→ Solution :

```
public class VoitureHybride extends Voiture {
    private double capaciteElectrique;

    public double getCapaciteElectrique() {
        return capaciteElectrique;
    }

    public void setCapaciteElectrique(double capaciteElectrique) {
        this.capaciteElectrique = capaciteElectrique;
    }

    public VoitureHybride(String marque, String pays, String modele, double capaciteElectrique) {
        super(marque, pays, modele);
        this.capaciteElectrique = capaciteElectrique;
    }
}
```

Le constructeur de la classe fille doit initialiser les attributs d'origine (**marque**, **modele** et **pays**) de l'objet, et l'attribut (ou les attributs) qui décrit l'aspect spécifique de l'objet : **capaciteElectrique**.

- Si une classe T2 hérite d'une classe T1, alors tout objet de type T2 est considéré comme un objet de type T1, et donc, une affectation d'un objet de type **T2** à une référence de type **T1** est 100% correcte. **L'inverse est faux**, et produit **une erreur de compilation**, et même si on force l'affectation par un cast, on obtient **une erreur d'exécution**.

```
}
public static void main(String[] args) {
    VoitureHybride v2 = new Voiture("Renault", "France", "Fluence", 0);
    Voiture v1 = new VoitureHybride("Renault", "France", "Fluence", 0);
}
```

VoitureHybride = Voiture → **faux**.
Voiture = VoitureHybride → **correct**.

// Avec le cast :

```
24 public static void main(String[] args) {
25     VoitureHybride v2 = (VoitureHybride) new Voiture("Renault", "France", "Fluence", 0);
26     Voiture v1 = new VoitureHybride("Renault", "France", "Fluence", 0);
27 }
28
29 }
```

Output - JAVA-11 (run) x Javadoc
run:
EXECUTION DU BLOC STATIC
Exception in thread "main" java.lang.ClassCastException: com.Voiture cannot be cast to com.VoitureHybride
at com.VoitureHybride.main(VoitureHybride.java:25)

- Idem pour les paramètres de méthodes : si une méthode accepte un paramètre de type Voiture, elle acceptera **les sous-types** de Voiture.

```
public class TestVoiture {
    static void mise_a_zero(Voiture a) {
        a.setMarque("");
        a.setModele("");
        a.setPays("");
    }

    public static void main(String[] args) {
        VoitureHybride v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 1000);
        mise_a_zero(v1);
        System.out.println("apres l'appel de mise_a_zero : "
            + v1.getMarque() + " - " + v1.getPays() + " - " + v1.getModele());
    }
}
```

Toute voiture hybride est une voiture, elle a donc tous les attributs d'une voiture, il n'y a donc aucun risque de dysfonctionnement en passant un objet **VoitureHybride** à la méthode mise_a_zero.

- Si la classe mère A1 possède des méthodes abstraites, alors pour la classe fille :
 - Soit elle définit **toutes** les méthodes abstraites de la classe mère A1,
 - Soit elle décide d'en laisser quelques-unes abstraites, et dans ce cas **elle doit être abstraite elle aussi**.

```

/**
 * @author s_elyahyaoui
 */
public abstract class Vehicule {
    protected abstract void demarrer();

    public Vehicule() {
    }
}

// classe Voiture sans définir la méthode demarrer()

package com;
public class Voiture extends Vehicule {
    protected String marque, pays, modele;
    static int nombreVoitures;

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
    }

    public String getMarque() {

```

Règles & définitions – 3 : Redéfinition / Surcharge des méthodes.

■ Définition / Redéfinition des méthodes : **Override**

- Si la méthode est abstraite dans la classe mère, alors au niveau de la classe fille on parle de « **définition** » de la méthode.
- Si la méthode est déjà définie dans la classe mère, alors au niveau de la classe fille on parle de « **redéfinition** » de la méthode.
- Dans les deux cas, cette action est appelée un **Override**, et on utilise le même terme au niveau de la syntaxe Java :

```

public class Voiture extends Vehicule {
    protected String marque, pays, modele;
    static int nombreVoitures;

    @Override
    protected void demarrer() {
        System.out.println("Demarrage d'une voiture " + this.marque + " " + this.modele + " :");
        System.out.println("Insérer clef de contact, ");
        System.out.println("Allumer démarreur, ");
        System.out.println("Demarrer, ");
        System.out.println("\nDemarrage réussi !!");
    }

    public Voiture(String marque, String pays, String modele) {
        this.marque = marque;
        this.pays = pays;
        this.modele = modele;
    }
}

```

■ Les méthodes héritées depuis la classe mère ont **par défaut le même comportement dans les classe filles.**

```

public class VoitureHybride extends Voiture {
    private double capaciteElectrique;

    public double getCapaciteElectrique() {
        return capaciteElectrique;
    }

    public void setCapaciteElectrique(double capaciteElectrique) {
        this.capaciteElectrique = capaciteElectrique;
    }

    public VoitureHybride(String marque, String pays, String modele, double capaciteElectrique) {
        super(marque, pays, modele);
        this.capaciteElectrique = capaciteElectrique;
    }

    public static void main(String[] args) {
        Voiture vh = new VoitureHybride("Renault", "France", "Fluence ZE", 0);
        vh.demarrer();
    }
}

```

```
EXECUTION DU BLOC STATIC
Demarrage d'une voiture Renault Fluence ZE :
Insérer clef de contact,
Allumer démarreur,
Demarrer,

Demarrage réussi !!
```

- Il arrive que la définition d'une méthode dans la classe mère ne convienne pas à l'une des classes filles, cette dernière peut donc redéfinir la méthode selon sa logique à elle :

```
/**
 * @author s_elyahyaoui
 */
public class VoitureCarte extends Voiture {

    public VoitureCarte(String marque, String pays, String modele) {
        super(marque, pays, modele);
    }

    @Override
    protected void demarrer() {
        System.out.println("Demarrage d'une voiture " + this.marque + " " + this.modele + " :");
        System.out.println("Insérer carte de contact, ");
        System.out.println("Cliquer sur le bouton Start, ");
        System.out.println("Demarrer, ");
        System.out.println("\nDemarrage réussi !!");
    }
}

public class TestVoiture {
    public static void main(String[] args) {
        Voiture v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 1000);
        v1.demarrer();
        System.out.println("\n **** \n");
        v1 = new VoitureCarte("BMW", "ALLEMAGNE", "745I");
        v1.demarrer();
    }
}
```

```
EXECUTION DU BLOC STATIC
Demarrage d'une voiture VOLVO C60 :
Insérer clef de contact,
Allumer démarreur,
Demarrer,

Demarrage réussi !!

****

Demarrage d'une voiture BMW 745I :
Insérer carte de contact,
Cliquer sur le bouton Start,
Demarrer,

Demarrage réussi !!
```

- Les méthodes déclarées avec le mot-clef **final** ne peuvent pas être redéfinies dans les classes filles.
- Les méthodes héritées depuis la classe mère, doivent avoir **le même degré de visibilité dans la classe fille**, ou bien **un degré plus fort**.

```
public class VoitureCarte extends Voiture {

    public VoitureCarte(String marque, String pays, String modele) {
        super(marque, pays, modele);
    }

    @Override
    private void demarrer() {
        System.out.println("Demarrage d'une voiture " + this.marque + " " + this.modele + " :");
        System.out.println("Insérer carte de contact, ");
        System.out.println("Cliquer sur le bouton Start, ");
        System.out.println("Demarrer, ");
        System.out.println("\nDemarrage réussi !!");
    }
}
```

demarrer() in VoitureCarte cannot override demarrer() in Vehicule attempting to assign weaker access privileges; was protected

(Alt-Enter shows hints)

- La redéfinition nécessite de garder **la même signature de la méthode** (même nom et même liste des paramètres). Pour une méthode qui est déjà définie dans la classe mère, si on change sa signature dans la classe fille alors on parle de **surcharge** (Overload), et non de redéfinition.
- On peut surcharger une méthode dans sa classe d'origine, sans avoir besoin d'héritage. La surcharge d'une méthode consiste à déclarer cette méthode avec des signatures différentes (changer la liste des paramètres). De cette manière on peut donc changer le comportement de cette méthode selon la signature.

```
public class VoitureHybride extends Voiture {
    private double capaciteElectrique;

    public double getCapaciteElectrique() {
        return capaciteElectrique;
    }

    public void setCapaciteElectrique(double capaciteElectrique) {
        this.capaciteElectrique = capaciteElectrique;
    }

    public VoitureHybride(String marque, String pays, String modele, double capaciteElectrique) {
        super(marque, pays, modele);
        this.capaciteElectrique = capaciteElectrique;
    }

    protected void demarrer(boolean modeElectrique) {
        if(modeElectrique) {
            if(capaciteElectrique >= 2500) {
                this.demarrer();
            } else {
                System.out.println("Impossible de demarrer avec la capacite " + this.capaciteElectrique);
                System.out.println("Veuillez recharger.");
            }
        }
    }
}
```

Nouvelle définition de la méthode `demarrer()` avec une signature différente de celle héritée depuis la classe `Voiture`.

On peut faire l'appel de la méthode `demarrer()` d'origine, dans la méthode de surcharge.

```
public class TestVoiture {
    public static void main(String[] args) {
        VoitureHybride v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 1000);
        v1.demarrer();
    }
}
```

Chaque référence de type **VoitureHybride** possède désormais **deux** méthodes `demarrer()`.

```
public class TestVoiture {
    public static void main(String[] args) {
        VoitureHybride v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 25);
        v1.demarrer(true);
    }
}
```

EXECUTION DU BLOC STATIC

```
Impossible de demarrer avec la capacite 25.0
Veuillez recharger.
```

```
public class TestVoiture {
    public static void main(String[] args) {
        VoitureHybride v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 3000);
        v1.demarrer(true);
    }
}
```

EXECUTION DU BLOC STATIC

```
Demarrage d'une voiture VOLVO C60 :
Inserer clef de contact,
Allumer demarreur,
Demarrer,

Demarrage reussi !!
```

3. Le polymorphisme d'objets et de méthodes

Exemple – 1 :

Surcharge de méthodes.

```
public class Garage {
    public void ajouterReparation(Voiture v) {...5 lines }

    public void ajouterReparation(Moto v) {...5 lines }

    public void ajouterReparation(AutoCar v) {...5 lines }

    public void ajouterReparation(SemiRemorque v) {...5 lines }
}

public class TestVoiture {
    public static void main(String[] args) {
        Garage g1 = new Garage();
        Voiture v1 = new Voiture();
        Moto v2 = new Moto();
        AutoCar v3 = new AutoCar();
        SemiRemorque v4 = new SemiRemorque();
        g1.ajouterReparation(v1);
        g1.ajouterReparation(v2);
        g1.ajouterReparation(v3);
        g1.ajouterReparation(v4);
    }
}
```

Exemple – 2 :

Changement du comportement d'une méthode en fonction du type d'objet qui l'appelle.

```
public class TestVoiture {
    public static void main(String[] args) {
        Voiture v1 = new VoitureHybride("VOLVO", "SUEDE", "C60", 1000);
        v1.demarrer();
        System.out.println("\n **** \n");
        v1 = new VoitureCarte("BMW", "ALLEMAGNE", "745I");
        v1.demarrer();
    }
}
```

Demarrage d'une voiture VOLVO C60 :
Inserer clef de contact,
Allumer demarreur,
Demarrer,

Demarrage reussi !!

Demarrage d'une voiture BMW 745I :
Inserer carte de contact,
Cliquer sur le bouton Start,
Demarrer,

Demarrage reussi !!

Exemple – 3 :

Appel d'une méthode avec un nombre variable de paramètres.

```
/**
 * @author s_elyahyaoui
 */
public class Garage {
    public void peindreVehicules(Vehicule ...v){...5 lines }
}

public class TestVoiture {
    public static void main(String[] args) {
        Garage g1 = new Garage();
        Voiture v1 = new Voiture();
        Moto v2 = new Moto();
        AutoCar v3 = new AutoCar();
        SemiRemorque v4 = new SemiRemorque();

        g1.peindreVehicules(v1);
        g1.peindreVehicules(v1, v3, v4);
    }
}
```

Exemple – 4 :

Appeler la même méthode avec plusieurs types (sous-types du même type parent).

```
Garrage g1 = new Garrage();
Vehicule x;
x = new Voiture();
g1.peindreVehicules(x);
x = new Moto();
g1.peindreVehicules(x);
x = new AutoCar();
g1.peindreVehicules(x);
x = new VoitureCarte("Laraki", "Maroc", "Fulgura");
g1.peindreVehicules(x);
x = new SemiRemorque();
g1.peindreVehicules(x);
```

Exemple – 5 :

Construire un tableau avec plusieurs objets ayant plusieurs types (sous-types du même type parent).

```
Voiture v1 = new Voiture();
Moto v2 = new Moto();
AutoCar v3 = new AutoCar();
SemiRemorque v4 = new SemiRemorque();

Vehicule liste[] = {v1, v2, v3, v4};
```

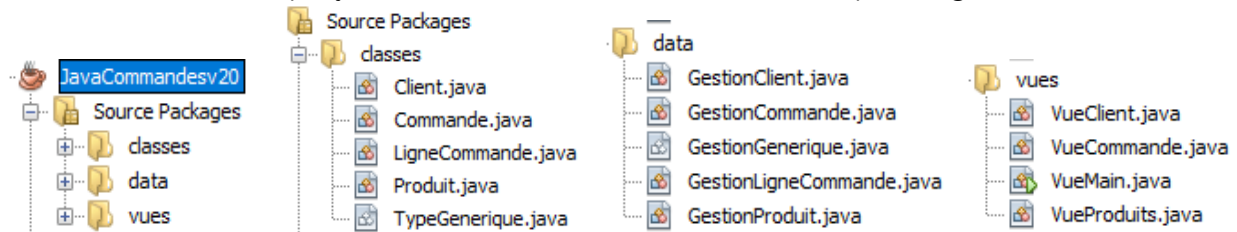
Exemple – 6 :

Utiliser la même référence de type T, pour des objets appartenant à plusieurs sous-types du type T.

```
Vehicule x;
x = new Voiture();
x = new Moto();
x = new AutoCar();
x = new VoitureCarte("Laraki", "Maroc", "Fulgura");
x = new SemiRemorque();
```

TP : Gestion des commandes v2

- Dans un nouveau projet « JavaCommandes20 », créer les packages suivants :



- Les classes Client, Commande, Produit et LigneCommande doivent hériter de la classe abstraite Model qui doit contenir l'attribut **id**, qui doit être hérité dans les classes filles.
- Les classes du package **data** doivent hériter de la classe abstraite **GestionGenerique**, qui doit contenir les méthodes du CRUD (*create, retrieve, update, delete*).
- Les listes (liste de clients, liste des commandes, ...) seront gérées dans des tableaux.
- La classe **VueMain** du package vues doit contenir la méthode principale, qui doit proposer tous les menus de gestion de produits, de commandes, de clients, ...
- Chaque menu de la méthode principale sera géré par **une méthode statique**.
- Exemples de menus :

Menu principal :

```

*** menu principal ***
choisissez un sous-menu :
*** tapez 1 pour gerer les clients ***
*** tapez 2 pour gerer les produits ***
*** tapez 3 pour gerer les commandes ***
*** tapez 0 pour quitter ***
  
```

Menu clients :

```

choisissez un sous-menu :
*** tapez 1 pour ajouter un client ***
*** tapez 2 pour modifier un client ***
*** tapez 3 pour supprimer un client ***
*** tapez 4 pour afficher les commandes d'un client ***
*** tapez 0 pour retourner au menu principal ***
  
```


6. Les Interfaces et l'Implémentation

Définition

- Tout comme les classes, une interface est aussi considérée comme un type, mais ce n'est pas vraiment une classe.
- Une interface peut être vue comme un *masque*, ou un *déguisement*, qu'on fait porter à nos objets, pour qu'ils adoptent des attitudes différentes, dans certaines situations.

Règles & définitions

- Une interface peut être déclarée dans son propre fichier java.
- Syntaxe :

```
public interface NomInterface {
    // déclarations
}
```

- Une interface **ne peut pas contenir** :
 - Des attributs ou méthodes protégés ou privés,
 - Des attributs déclarés sans initialisation,
 - Un constructeur,
 - Des méthodes non-abstraites.
- Les attributs d'une interface sont **par obligation** : **publics** et **statiques**, même quand les modificateurs `public` `static` ne sont pas explicitement utilisés.
- Vu que ce sont des *attitudes à adopter*, et non des classes, les interfaces sont **implémentées** - et non héritées - par les classes.

```
public class UneClasse implements UneInterface {
    // implémentation des méthodes
}
```

- Sauf si elle est abstraite, une classe qui implémente une interface doit définir **toutes** les méthodes de l'interface.
- Une classe peut implémenter plusieurs interfaces à la fois, à condition qu'elle implémente toutes les méthodes de toutes ses interfaces :

```
public class C1 implements Interface1, Interface2 {
    /* implémentation des méthodes de Interface1 */
    /* implémentation des méthodes de Interface2 */
}
```

- Une classe peut implémenter des interfaces et hériter une classe, mais en respectant l'ordre suivant :

```
// ordre correct
public class Class1 extends Class2 implements Interface1, Interface2 { ... }

// erreur de compilation
public class Class1 implements Interface1, Interface2 extends Class2 { ... }
```

- Les règles de compatibilité entre types génériques et types spécifiques appliquées à l'héritage sont aussi appliquées à l'implémentation des classes.
- Une interface peut elle-même hériter une autre interface, dans ce cas elle hérite tous les attributs et méthodes de l'interface mère :

```
public interface Interface1 extends Interface2 { ... }
```

- Si une classe implémente une interface qui hérite d'une autre interface, alors cette classe devra implémenter toutes les méthodes des deux interfaces.

ELYAHYAOU.S - JAVA

CHAPITRE 3. POO AVANCEE EN JAVA

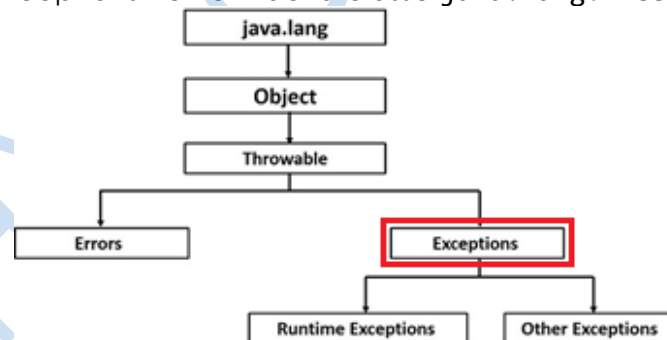
1. La gestion des exceptions

Définition : Exception

- Une exception est un problème qui survient lors de l'exécution - et non la compilation - d'un programme.
- Une exception peut se produire pour plusieurs raisons, soit par la faute de l'utilisateur du programme, soit par celle du programmeur, soit à cause d'un problème technique :
 - L'utilisateur entre des données erronées (un champ de texte vide par ex.)
 - Interruption lors d'une connexion réseau,
 - Tentative d'ouverture d'un fichier inexistant,
 - Recherche d'un indice incorrect dans un tableau (inf. à 0 ou sup. à l'indice maximal),
 - Recherche dans une table qui n'existe pas dans une BD,
 - Etc.
- Quand une exception se produit, le programme est brutalement arrêté sans fournir d'explications à l'utilisateur.

Traiter des exceptions

- Pour éviter l'arrêt du programme et exécuter des instructions de remplacement en cas d'exception, on doit placer le code susceptible de lever l'exception dans un bloc `try{ ... }`, le code de remplacement doit être placé dans le bloc `catch(Type_de_l'exception){ ... }`. On parle alors « d'attraper » une exception.
- Chaque type d'exception (réseau, lecture / écriture sur fichier, données incorrectes, ...) est représenté par une classe java.
- Toutes les classe d'exceptions héritent de la classe `java.lang.Exception`.



- Souvent une exception e_1 peut causer une autre e_2 , qui peut à son tour causer une exception e_3 , etc, jusqu'à arriver à l'exception finale e_N qui cause explicitement l'arrêt du programme.
- Principales méthodes d'un objet Exception :

getMessage()	Retourne un message (String) qui décrit l'exception levée.
getCause()	Retourne la cause d'origine (objet de type Throwable) d'une exception.
toString()	Retourne le nom de classe de l'objet exception, concaténé avec le résultat de <code>getMessage()</code> .

printStackTrace() Affiche (dans la sortie *System.err*) le résultat de *toString()* de l'exception finale, avec la trace de toutes les exceptions intermédiaires *e₁*, *e₂*, *e₃* ...

Exemple :

```
run:
Exception in Application start method
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.sun.javafx.application.LauncherImpl.launchApplicationWithArgs(LauncherImpl.java:389)
    at com.sun.javafx.application.LauncherImpl.launchApplication(LauncherImpl.java:328)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at sun.launcher.LauncherHelper$FXHelper.main(LauncherHelper.java:767)
Caused by: java.lang.RuntimeException: Exception in Application start method
    at com.sun.javafx.application.LauncherImpl.launchApplication1(LauncherImpl.java:917)
    at com.sun.javafx.application.LauncherImpl.lambda$launchApplication$154(LauncherImpl.java:182)
    at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.NullPointerException: Location is required.
    at javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:3207)
    at javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:3175)
    at javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:3148)
    at javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:3124)
    at javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:3104)
    at javafx.fxml.FXMLLoader.load(FXMLLoader.java:3097)
    at vues.VueMain.start(VueMain.java:31)
    at com.sun.javafx.application.LauncherImpl.lambda$launchApplication$161(LauncherImpl.java:863)
    at com.sun.javafx.application.PlatformImpl.lambda$runAndWait$174(PlatformImpl.java:326)
    at com.sun.javafx.application.PlatformImpl.lambda$null$172(PlatformImpl.java:295)
    at java.security.AccessController.doPrivileged(Native Method)
    at com.sun.javafx.application.PlatformImpl.lambda$runLater$173(PlatformImpl.java:294)
    at com.sun.glass.ui.InvokeLaterDispatcher$Future.run(InvokeLaterDispatcher.java:95)
    at com.sun.glass.ui.win.WinApplication._runLoop(Native Method)
    at com.sun.glass.ui.win.WinApplication.lambda$null$147(WinApplication.java:177)
    ... 1 more
Exception running application vues.VueMain
C:\Users\youss\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

- On peut personnaliser ces méthodes en héritant la classe *Exception*.
- Traitement d'une exception :

```
try {
    /*
        instruction qui peut lever une exception de type Type1
    */
} catch (Type1 ex) { /* "attraper" l'exception */
    /* traitement de remplacement, afficher l'erreur à l'écran,
        enregistrer l'erreur dans un fichier log, etc. */
}
```

- Traiter plusieurs types d'exceptions dans le même bloc *try* :

```
try {
    /*
        Instruction1 → peut lever une exception de type Type1
        Instruction2 → peut lever une exception de type Type2
        Instruction3 → peut lever une exception de type Type3
    */
} catch (Type1 ex) {
```

```

...
} catch(Type2 ex) {
    ...
} catch(Type3 ex) {
    ...
}

```

- Traiter plusieurs exceptions avec un seul bloc catch :

// 1iere solution : en utilisant un seul objet Exception, vu que tous les types d'exceptions héritent de la classe Exception.

```

try {
    /*
        Instruction1 → peut lever une exception de type ArrayIndexOutOfBoundsException
        Instruction2 → peut lever une exception de type IOException
        Instruction3 → peut lever une exception de type NullPointerException
    */
} catch(Exception ex) {
    ...
}

```

// 2ieme solution (utilisable depuis Java 7)

```

try {
    /*
        Instruction1 → peut lever une exception de type Type1
        Instruction2 → peut lever une exception de type Type2
        Instruction3 → peut lever une exception de type Type3
    */
} catch(Type1 | Type2 | Type3 ex) {
    ...
}

```

Déclaration d'une (ou plusieurs) exceptions

- Il arrive qu'une méthode contient des instructions qui peuvent lever une (ou plusieurs) exception, mais elle ne traite pas elle-même ces exceptions.
- Dans ce cas la méthode doit déclarer l'exception :

```

type_retour méthode(paramètres) throws TypeException1, TypeException2 {
    instructions
}

```

- Si une méthode M1 fait l'appel d'une méthode M2 qui déclare une exception, alors :
 - Soit M1 doit traiter l'exception (TRY - CATCH).
 - Soit M1 doit elle aussi déclarer l'exception avec (throws),
 - Soit M1 est la méthode principale (main), dans ce cas elle doit traiter l'exception.
- Lever explicitement une exception :

```
throw new Exception("un message");
```

- Si une méthode – dans sa classe d'origine – déclare une exception dans sa signature, alors dans les classes filles, cette méthode doit déclarer la même exception.

Exemple 1

```

11  /**
12   * @author s_elyahyaoui
13   */
14  public class GestionTableaux {
15      public static void insert(Vehicule[] liste, Vehicule element, int index) {
16          if(liste[index] == null)
17              liste[index] = element;
18          else
19              System.out.println("La cellule choisie est occupée, choisir une autre.");
20      }
21
22      public static void main(String[] args) {
23          Vehicule vehicules[] = new Vehicule[10];
24          SemiRemorque v1 = new SemiRemorque();
25          GestionTableaux.insert(vehicules, v1, 10);
26      }
27  }

```

Output - JAVA-11 (run) × Javadoc

```

run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at testExceptions.GestionTableaux.insert(GestionTableaux.java:16)
    at testExceptions.GestionTableaux.main(GestionTableaux.java:25)

```

Exemple 2

```

public class GestionTableaux {
    public static void insert(Vehicule[] liste, Vehicule element, int index) {
        try {
            if(liste[index] == null)
                liste[index] = element;
            else
                System.out.println("La cellule choisie est occupée, choisissez une autre.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Erreur d'indexation : \n");
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Vehicule vehicules[] = new Vehicule[10];
        SemiRemorque v1 = new SemiRemorque();
        GestionTableaux.insert(vehicules, v1, 10);
    }
}

```

Erreur d'indexation :

```

java.lang.ArrayIndexOutOfBoundsException: 10
    at testExceptions.GestionTableaux.insert(GestionTableaux.java:17)
    at testExceptions.GestionTableaux.main(GestionTableaux.java:30)

```

Exemple 3

```

public class GestionTableaux {

    public static void insert(Vehicule[] liste, Vehicule element, int index)
        throws ArrayIndexOutOfBoundsException {
        if (liste[index] == null) {
            liste[index] = element;
        } else {
            System.out.println("La cellule choisie est occupée, choisissez une autre.");
        }
    }

    public static void main(String[] args) {
        Vehicule vehicules[] = new Vehicule[10];
        SemiRemorque v1 = new SemiRemorque();
        try {
            GestionTableaux.insert(vehicules, v1, 10);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Methode MAIN, erreur d'indexation : \n");
            e.printStackTrace();
        }
    }
}

```

```
Methode MAIN, erreur d'indexation :
java.lang.ArrayIndexOutOfBoundsException: 10

    at testExceptions.GestionTableaux.insert(GestionTableaux.java:18)
    at testExceptions.GestionTableaux.main(GestionTableaux.java:29)
```

Créer ses propres classes d'exceptions

- L'héritage de la classe Exception est une manière encore plus professionnelle de gérer les exceptions dans une application.

```
class MyException extends Exception {
    /* Redéfinition ou surcharge des méthodes qu'on souhaite modifier */
}
```

Exemple 4

/* création de deux nouveaux types d'exception pour gérer les erreurs : indice incorrecte et cellule occupée */

```
/**
 * @author s_elyahyaoui
 */
public class ExceptionIndexation extends Exception {

    private int indice;

    public int getIndice() {
        return indice;
    }

    public void setIndice(int indice) {
        this.indice = indice;
    }

    public ExceptionIndexation(int indice) {
        super();
        this.indice = indice;
    }

    @Override
    public String getMessage() {
        return "Erreur d'indexation.\nIndice : " + this.indice;
    }
}

public class ExceptionCelluleOccupee extends Exception {

    private int indice;

    public int getIndice() {
        return indice;
    }

    public void setIndice(int indice) {
        this.indice = indice;
    }

    public ExceptionCelluleOccupee(int indice) {
        super();
        this.indice = indice;
    }

    @Override
    public String getMessage() {
        return "La cellule : " + this.indice + " est occupée.\n"
            + "Choisissez une autre.";
    }
}
```

```
public class GestionTableaux {  
  
    public static void insert(Vehicule[] liste, Vehicule element, int index)  
        throws ExceptionIndexation, ExceptionCelluleOccupee {  
  
        if(index < 0 || index >= liste.length)  
            throw new ExceptionIndexation(index);  
        else {  
            if (liste[index] != null) {  
                throw new ExceptionCelluleOccupee(index);  
            } else  
                liste[index] = element;  
        }  
    }  
  
    public static void main(String[] args) {  
        Vehicule vehicules[] = new Vehicule[10];  
        SemiRemorque v1 = new SemiRemorque();  
  
        try {  
            GestionTableaux.insert(vehicules, v1, 10);  
        } catch (ExceptionIndexation|ExceptionCelluleOccupee e) {  
            System.out.println("Methode MAIN : \n");  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Methode MAIN :

Erreur d'indexation.

Indice : 10

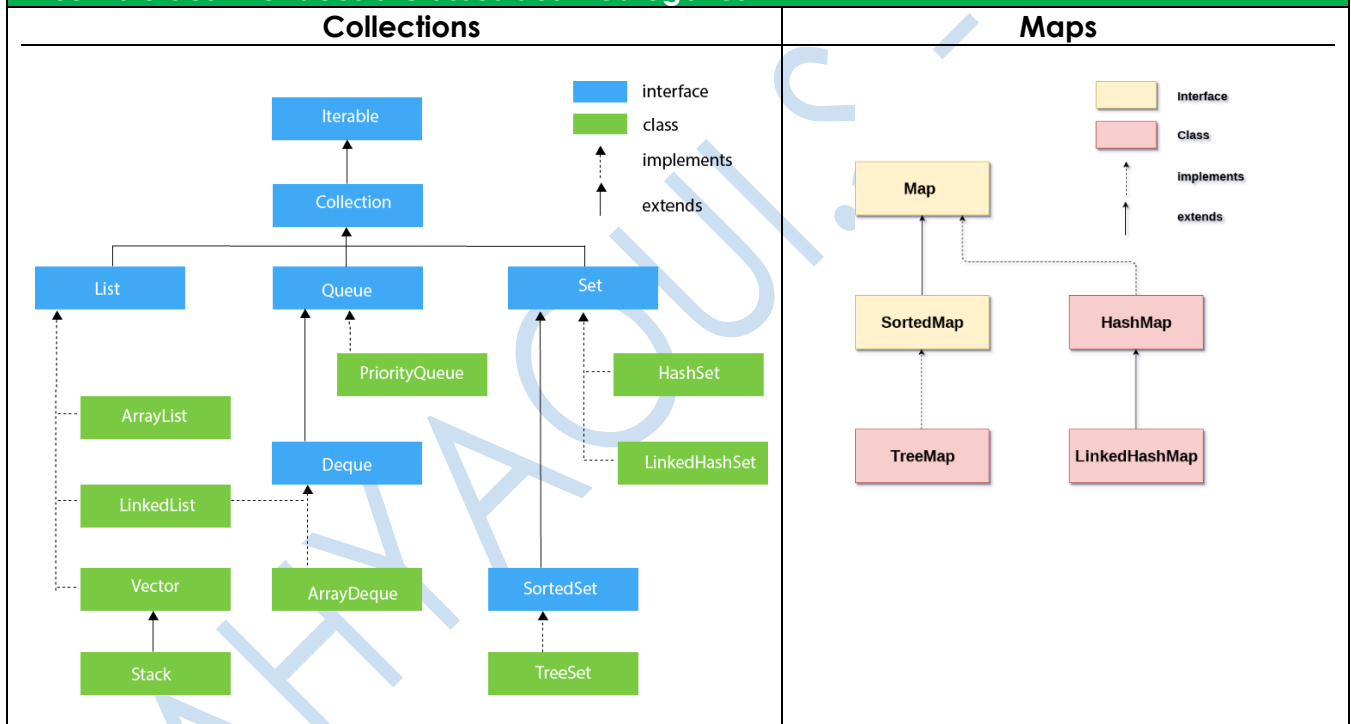
2. Les collections d'objets

1. Généralités

Définitions

- Les collections d'objets sont des tableaux dynamiques d'objets, leur taille n'est pas fixe, elle varie selon les actions d'ajout / suppression d'éléments.
- Il existe plusieurs types de collections d'objets en Java, regroupés dans 2 grandes catégories : les **collections** et les **maps**.
- Ces 2 catégories sont représentées par les interfaces :
 - **Collection** : un ensemble d'objets ordonnés soit par un ordre d'indexation, soit à la manière d'un arbre (père / fils).
 - **Map** : un ensemble d'objets qui forme un dictionnaire (**clef, valeur**).
Chaque objet (valeur) est recherché – non pas par son indice – mais par sa clef.
- Ces interfaces sont implémentées par plusieurs classes, chacune d'elles définit sa propre manière d'organisation des éléments.

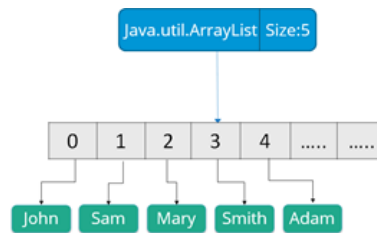
Ensemble des interfaces & classes des 2 catégories



2. 1^{er} exemple : les classes ArrayList & Vector

Définition : ArrayList

- **ArrayList** est une **liste** d'objets, ayant des indices, le premier indice est **0**.



- Création :

```
ArrayList listeVoitures = new ArrayList();
// ou bien (syntaxe préférable)
ArrayList<Voiture> listeVoitures = new ArrayList<Voiture>();
```

- Ajout : insérer à la fin

```
liste.add(un objet Voiture);
```

- Insertion : insérer au milieu

```
liste.add(i, un objet Voiture);
/* le nouvel élément sera inséré à l'indice i et les autres éléments se situant de
l'indice i jusqu'à la fin seront décalés vers la droite.
```

- Ajout / insertion de plusieurs éléments :

```
ArrayList<Voiture> nouvelle_liste = new ArrayList<Voiture>();
nouvelle_liste.add(v1);
nouvelle_liste.add(v2);
liste.addAll(nouvelle_liste); // ajouter à la fin
liste.addAll(i, nouvelle_liste); // insérer au milieu
```

- Remplacement d'un élément :

```
liste.set(i, v);
```

- Suppression :

```
liste.remove(i); /* ou bien liste.remove(element); */
```

- Parcours des éléments :

```
for(Voiture v : liste) {
    // instructions sur v
}
/* Ou bien */
Voiture v;
for(int i = 0; i < liste.size(); i++) {
    v = liste.get(i);
    // instructions sur v
}
```

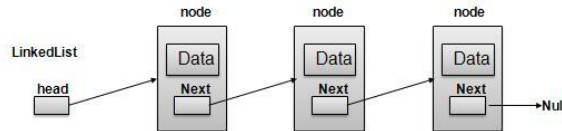
ArrayList / LinkedList / Vector

▪ Vector

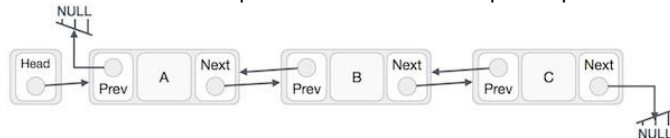
- Identique à ArrayList, sauf que la classe Vector est dite *thread safe*, c.à.d gère les accès concurrents, au contraire de la classe ArrayList.
- Un cas d'accès concurrents se produit quand plusieurs processus tentent d'effectuer des opérations d'écriture sur le même objet.

▪ LinkedList : C'est aussi une liste, mais elle présente les différences suivantes :

- Chaque élément est lié à celui qui le suit (*liste chaînée simple*),



ou bien à celui qui le suit et celui qui le précède (*liste chaînée double*)



- Possibilité d'ajouter ou supprimer le premier et le dernier élément via les méthodes **addFirst**, **removeFirst**, ...

3. 2^{ème} exemple : la classe Stack

Définition

- Représente une pile - et non une liste - d'éléments.
- Chaque élément ajouté est positionné au-dessus des autres.
- Méthodes :
 - **bool empty()** : teste si la pile est vide.
 - **Object peek()** : retourne l'élément qui est en haut de la pile.
 - **Object pop()** : retourne l'élément qui est en haut de la pile en le supprimant.
 - **Object push(Object element)** : ajoute l'élément dans la pile, et le retourne.
 - **int search(element)** : retourne la position de l'élément depuis le haut. Si non trouvé, retourne -1.

Exemple

```
public class TestCollections {
    public static void main(String[] args) {
        Stack st = new Stack();
        st.push(5);
        System.out.println(st);
        st.push(10);
        st.push(20);
        st.push(30);
        System.out.println(st);
        st.pop();
        System.out.println(st);
        System.out.println(st.search(10));
    }
}
```

```
put - JAVA-II (run) x
run:
[5]
[5, 10, 20, 30]
[5, 10, 20]
2
```

4. 3^{ème} exemple : la classe HashMap

Définition

- Représente un dictionnaire d'éléments : clef-valeur.
- Chaque élément doit être ajouté / supprimé / remplacé / récupéré **par sa clef**.
- La clef et la valeur doivent être de type référence.
- Syntaxe :

```
public class TestCollections {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(2, "sbah lkhir"); /* put ajoute un element. si la clef existe deja,
                                   l'element est remplacé (on peut aussi remplacer avec replace)
                                   */
        map.put(4, "good morning");
        map.put(5, "bonjour");

        for(String message : map.values()) {
            System.out.println(message);
        }
        System.out.println(" ----- ");
        map.remove(5);
        for(String message : map.values()) {
            System.out.println(message);
        }
    }
}
```

put - JAVA-II (run) ×

```
run:
sbah lkhir
good morning
bonjour
-----
sbah lkhir
good morning
```

5. Ordonner une liste d'objets

Définition

- Les éléments d'une `ArrayList<C1>` (ou autre) peuvent être ordonnés de deux manières :
 - Soit la classe **C1** doit implémenter l'interface `Comparable`, pour définir le critère d'ordre dans la méthode `compareTo(C1)`. Cette méthode doit retourner 1 (supérieur), 0 (égal) ou -1 (inférieur). Mais dans ce cas on ne peut définir qu'un seul critère d'ordre. Puis on utilise la méthode `Collections.sort(liste)`.



REMARQUE

Cette méthode signale une erreur de syntaxe si la classe C1 n'implémente pas l'interface `Comparable`.

- Soit on utilise une implémentation de l'interface `Comparator<C1>`, pour définir le critère d'ordre dans la méthode `compare(C1 a, C1 b)`. Cette méthode doit retourner 1 (supérieur), 0 (égal) ou -1 (inférieur). Dans ce cas on peut définir plusieurs critères d'ordre en multipliant les implémentations de l'interface `Comparator`. Puis on utilise la méthode `Collections.sort(liste, objet_comparateur)`.

Exemple : utilisation de l'interface `Comparator` pour ordonner une liste de clients.

```
// classe Client
public class Client {
    private int id;
    private String firstname, name, address, email, phone;

    public Client(int id, String firstname, String name, String address, String email, String phone) {
        this.id = id;
        this.firstname = firstname;
        this.name = name;
        this.address = address;
        this.email = email;
        this.phone = phone;
    }

    @Override
    public String toString() {
        return this.id + " - " + this.firstname;
    }
}

// comparateur
public class SortByID implements Comparator<Client> {
    @Override
    public int compare(Client o1, Client o2) {
        if (o1.getId() > o2.getId()) {
            return 1;
        } else {
            if (o1.getId() < o2.getId()) {
                return -1;
            } else {
                return 0;
            }
        }
    }
}

// classe de test
```

```
public class TestComparaison {  
  
    public static void main(String[] args) {  
        ArrayList<Client> liste = new ArrayList<>();  
        liste.add(new Client(1, "clt 1", "clt 1", "addr 1", "email 1", "phone 1"));  
        liste.add(new Client(4, "clt 4", "clt 4", "addr 4", "email 4", "phone 4"));  
        liste.add(new Client(2, "clt 2", "clt 2", "addr 2", "email 2", "phone 2"));  
  
        System.out.println("avant Collections.sort");  
        for(Client c : liste) {  
            System.out.println(c.toString());  
        }  
  
        Collections.sort(liste, new SortByID());  
  
        System.out.println("\naprès Collections.sort");  
        for(Client c : liste) {  
            System.out.println(c.toString());  
        }  
    }  
}
```

avant Collections.sort

1 - clt 1
4 - clt 4
2 - clt 2

après Collections.sort

1 - clt 1
2 - clt 2
4 - clt 4

TP

Reprendre le TP de gestion des commandes. Dans la partie de gestion des produits proposer d'ordonner les produits par plusieurs critères d'ordre.

3. Les expressions Lambdas

1. Les expressions lambdas

Définition

- Les « expressions lambdas » sont des expressions qu'on utilise pour faciliter certains aspects de la programmation en Java, et pour réduire la quantité de code qu'on doit écrire pour avoir le même résultat.
- Elles ont été introduites avec la version 8 de Java.
- Elles permettent – entre autres – de
 - Faciliter l'implémentation d'une interface ayant une seule méthode à définir,
 - Passer une méthode comme paramètre à une méthode,
 - Faciliter le parcours des collections,
 - ...
- La syntaxe générale d'une expression lambda est comme ceci :

```
(paramètre1, paramètre2, ...) -> {
    Instruction1;
    Instruction2;
    .....
}
```

S'il y a un seul paramètre on peut enlever les parenthèses :

```
un_parametre -> {
    Instruction1;
    Instruction2;
    .....
}
```

S'il y a une seule instruction on peut enlever les accolades et le point-virgule :

```
(paramètre1, paramètre2, ...) -> instruction
```

Une expression lambda peut aussi ne pas avoir de paramètres :

```
() -> {
    Instruction1;
    Instruction2;
    .....
}
```

2. Cas 1 : Les interfaces fonctionnelles

Définition

- L'appellation « interface fonctionnelle » correspond à une interface qui ne contient qu'une seule méthode à implémenter.
- Pour utiliser ce type de l'interface il faut l'implémenter. Cela implique :
 - Soit de créer une classe spécifiquement pour implémenter l'interface, puis utiliser la classe créée,
 - Soit de l'implémenter dans une classe anonyme. Et c'est là que les expressions lambda entrent en jeu.

Exemple

On suppose l'interface appelée IFonctionnelle qui contient une méthode « action ». Cette méthode prend en paramètre un entier et une chaîne, et retourne une chaîne.

```
package lambdas;
/**
 * @author s_elyahyaoui
 */
public interface IFonctionnelle {
    String action(int a, String b);
}
```

On suppose en suite la classe ActionTest qui contient la méthode « executerAction » :

```
public String executerAction(IFonctionnelle i, int a, String b) {
    return i.action(a, b);
}
```

Pour pouvoir utiliser cette méthode, sans utiliser les expressions lambdas, on doit :

- implémenter l'interface IFonctionnelle dans une classe A,
- instancier la classe A,
- passer l'instance en paramètre à la méthode executerAction.

Solution en utilisant les expressions lambdas :

```
package lambdas;
/**
 * @author s_elyahyaoui
 */
public class ActionTest {

    public String executerAction(IFonctionnelle i, int a, String b) {
        return i.action(a, b);
    }

    public static void main(String[] args) {
        ActionTest actiontest = new ActionTest();
        System.out.println(actiontest.executerAction((a, b) -> {
            return a + b;
        }, 1, " - bonjour"));
    }
}
```


3. Cas 2 : Parcourir une collection

Exemple

Cette boucle :

```
for (int i = 0; i < list.size(); i++) {  
    String s = list.get(i);  
    System.out.print(s);  
}
```

Peut être simplifiée par la syntaxe suivante :

```
list.forEach((s) -> System.out.print(s));
```

ou bien :

```
list.forEach(s -> System.out::print);
```