

Chapitre 1 : Les bases du langage Java

1. Introduction

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

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

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

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

ces concepts permet de minimiser les erreurs liées à la gestion de la mémoire, qui constituent dans certains cas des sources de vulnérabilité.

- Gratuit : il existe des outils gratuits de qualité pour le développement (IDE comme Eclipse, Netbeans,...) et l'exécution des applications Java

2. Java et Portabilité

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

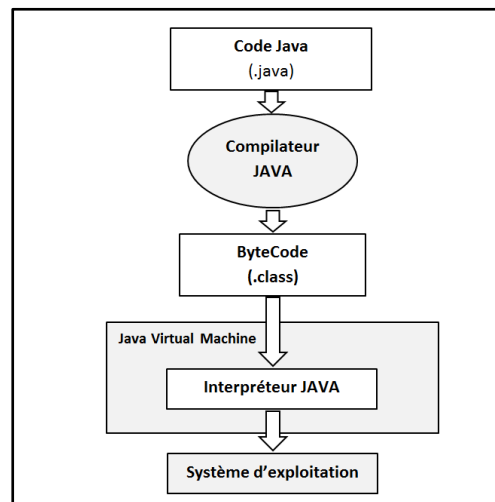


Figure 1

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

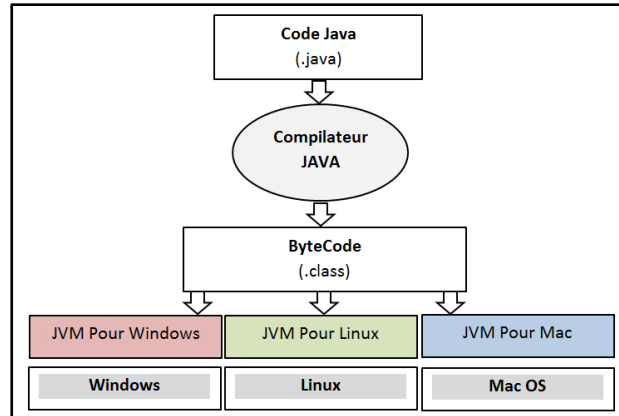


Figure 2

3. Plateforme de développement et d'exécution

3.1. JVM, JRE, et JDK

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

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

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

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

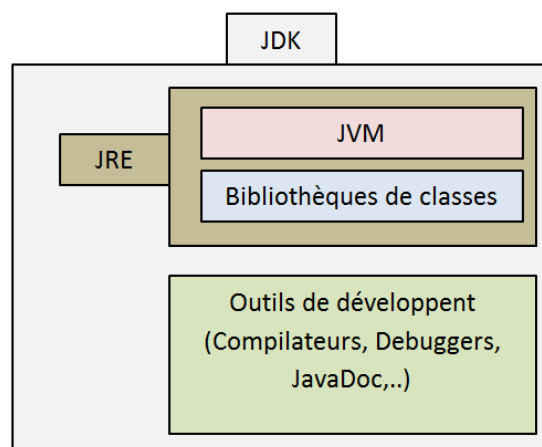


Figure 3

3.2. Environnement de développement intégré IDE.

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

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

4. Types primitifs

4.1. Les types entiers

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

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

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

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

4.2. Types réels

Il existe deux types de réels :

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

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

Exemples :

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

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

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

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

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

4.3. Type char

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

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

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

4.4. Type boolean :

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

5. Déclaration et initialisation d'une variable

5.1. Déclaration d'une variable

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

IDENTIFICATEUR_DE_TYPE variable1, variable2, ..., variableN ;

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

Les variables peuvent être initialisées lors de leur déclaration.

Exemples :

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

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

Exemples d'identificateurs valides :

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

Exemples d'identificateurs invalides :

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

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

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

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

5.2. Déclaration d'une constante

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

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

Exemple : Déclaration de la constant Pi : **final float** PI=3.14f;

5.3. Initialisation d'une variable

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

```
int lCoeff = 2 ;
```

Cette instruction est équivalent deux instructions suivantes :

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

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

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

6. Opérateurs

6.1. Les opérateurs arithmétiques :

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

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

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

6.2. Les opérateurs relationnels

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

6.3. Les opérateurs logiques

Opérateurs	Signification
!	négation
&	et
^	ou exclusif
	ou inclusif
&&	et (avec court-circuit)
	ou inclusif (avec court-circuit)

6.4. L'opérateur d'affectation usuel

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

6.5. Les opérateurs d'incrément et de décrémentation

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

6.6. Les opérateurs d'affectation élargie

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

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

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

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

6.7. Les opérateurs de manipulation de bits

Opérateurs	Signification
&	et (bit à bit)
	ou inclusif (bit à bit)
^	ou exclusif (bit à bit)
<<	décalage à gauche
>>	décalage arithmétique à droite
>>>	décalage logique à droite
~ (unaire)	complément à un (bit à bit)

6.8. L'opérateur conditionnel ou opérateur ternaire

variable = *expression_logique* ? *valeur1* : *valeur2* est équivalent à :

si *expression_logique* alors *variable* = *valeur1* sinon *variable* = *valeur2*.

6.9. Priorité des opérateurs

La priorité d'un opérateur sur un autre et son associativité déterminent dans quel ordre seront appliqués les opérateurs. Comme en mathématiques, une opération est rendue prioritaire grâce à

l'usage de parenthèses. Le tableau ci-dessous récapitule les opérateurs et leur priorité. Les opérateurs figurant sur une même ligne ont la même priorité ; ils sont rangés ligne par ligne du plus prioritaire au moins prioritaire.

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

7. Expressions et conversions

7.1. Les conversions implicites dans les expressions

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

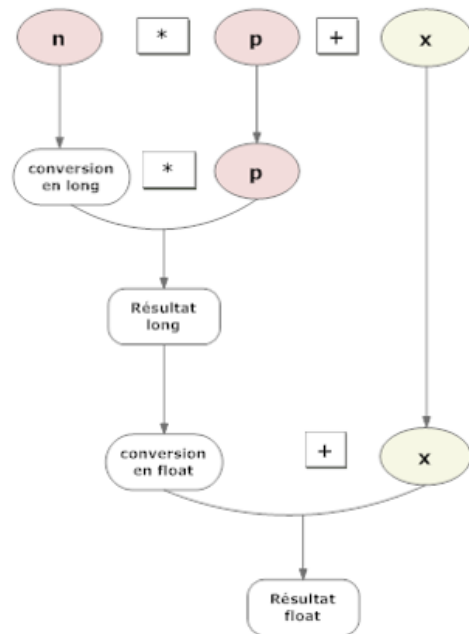
Les conversions d'ajustement de type : Elles se font selon cette hiérarchie :

int ➔ *long* ➔ *float* ➔ *double*

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

Exemples :

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



Conversion de n en *long* et multiplication par p

Le résultat de * est de type *long*

Le résultat est converti en *float* pour être additionné à x

Le résultat final est de type *float*

Figure 4

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

$n * p + m + x$ est évaluée ainsi :

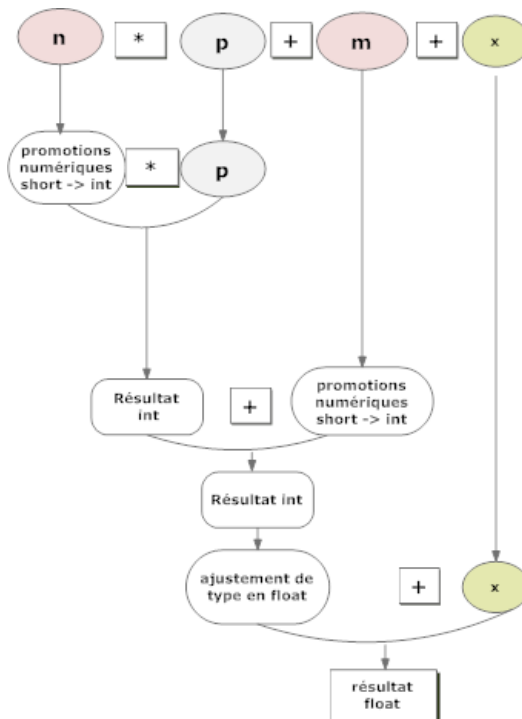


Figure 5

7.2. Conversions implicite par affectation

Les différentes possibilités sont :

byte ➔ *short* ➔ *int* ➔ *long* ➔ *float* ➔ *double* et *char* ➔ *int*

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

```
short p ;
```

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

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

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

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

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

8. La déclaration var

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

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

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

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

On considère un autre exemple :

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

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

9. Premier exemple de programme Java

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

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

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

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

10. Les commentaires

Il existe 3 types de commentaires :

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

Exemple :

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

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

Exemple :

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

Le dernier type est les commentaires utilisés pour écrire la documentation du code (commentaires javadoc). Généralement ce type de commentaires est utilisé pour documenter une classe et ses membres.

Ils commencent par `/**` et se terminent par `*/`. L'intérêt de ces commentaires est la possibilité de les transformer par des outils automatiques en une documentation sous un format HTML par exemple.

Exemple :

```
/**
 * Cette classe est un exemple de cours
 *
 * @author Tarik BOUDAA
 *
 */
public class Main {
```

```

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

```

11. Les conditions avec l'instruction if

11.1. Syntaxe de l'instruction if

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

```

if(booleanExpression)
    instruction ;
ou

```

```

if(booleanExpression) {
    instruction ;
}

```

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

```

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

```

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

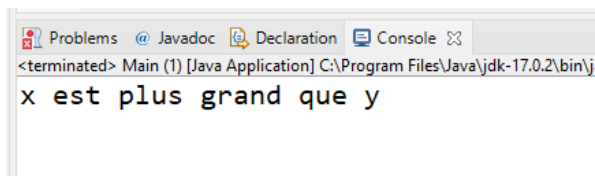
Exemple

```

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

```

A l'exécution on obtient :



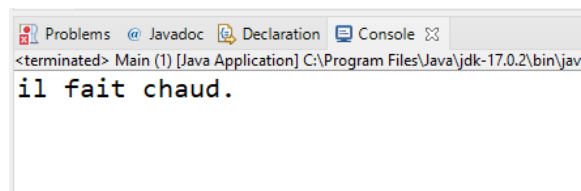
11.2. Syntaxe de l'instruction If else

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

Exemple

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

A l'exécution on obtient :



11.3. L'instruction else if

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

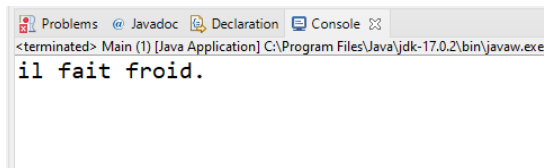
```
public class Main {  
    public static void main(String[] args) {  
        int temperature = 20;  
        if (temperature < 28 && temperature > 22) {  
            System.out.println("il fait beau temps.");  
        } else if (temperature > 40) {  
            System.out.println("il fait très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

```

        else{
            System.out.println("il fait froid.");
        }
    }
}

```

A l'exécution on obtient :



```

Problems Javadoc Declaration Console
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17.0.2\bin\javaw.exe
il fait froid.

```

12. Les conditions avec l'instruction switch

12.1. L'instruction switch

```

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

```

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

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

Exemple :

```

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

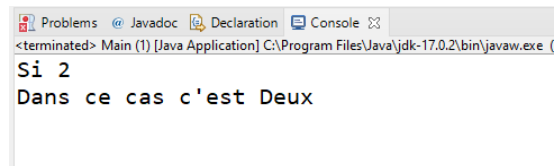
```

```

        default:
            System.out.println("Si aucune");
            System.out.println("Dans ce cas aucune des valeurs");
            break;
    }
}
}

```

A l'exécution on obtient :



```

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

```

Remarque :

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

Exemple :

```

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

        int monthNumber = 0;
        switch (month) {
            case "Janvier":
                monthNumber = 1;
                break;
            case "Février":
                monthNumber = 2;
                break;
            case "Mars":
                monthNumber = 3;
                break;
            case "Avril":
                monthNumber = 4;
                break;
            case "Mai":
                monthNumber = 5;
                break;
            case "Juin":
                monthNumber = 6;
                break;
            case "Juillet":
                monthNumber = 7;
                break;
        }
    }
}

```



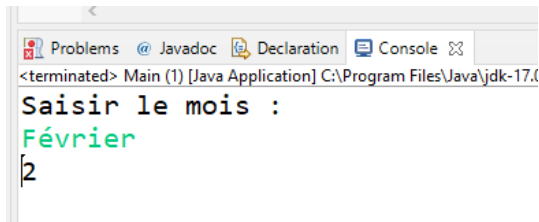
```

        case "Août":
            monthNumber = 8;
            break;
        case "Septembre":
            monthNumber = 9;
            break;
        case "Octobre":
            monthNumber = 10;
            break;
        case "Novembre":
            monthNumber = 11;
            break;
        case "Decembre":
            monthNumber = 12;
            break;
        default:
            monthNumber = 0;
            break;
    }

    if (monthNumber == 0) {
        System.out.println("Mois invalide");
    } else {
        System.out.println(monthNumber);
    }
}

```

A l'exécution on obtient :



12.2. Nouvelle syntaxe de l'instruction switch

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

Syntaxe :

```

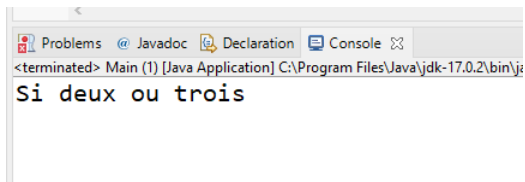
switch (expression)
{
    case suite_de_constantes_1 -> instruction_1
    case suite_de_constantes_2 -> instruction_2
    .....
    case suite_de_constantes_n -> instruction_n
    [ default -> instruction ]
}

```

Exemple :

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

A l'exécution on obtient :



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

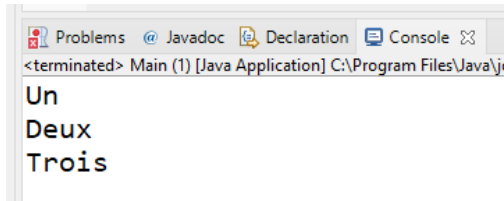
Syntaxe :

```
switch (expression)  
{  
    case suite_de constantes_1 -> {  
        instruction_1_1 ;  
        ...  
        instruction_1_n ;  
    }  
    case suite_de constantes_2 -> {  
        instruction_2_1 ;  
        ...  
        instruction_2_n ;  
    }  
    .....  
    case suite_de constantes_n -> {  
        instruction_n_1 ;  
        ...  
        instruction_n_n ;  
    }  
    [ default -> {  
        instruction_d_1 ;  
        ...  
        instruction_d_n ;  
    }  
]  
}
```

Exemple :

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

A l'exécution on obtient :



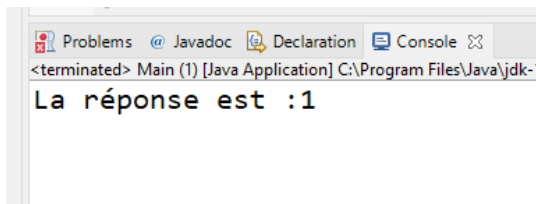
Remarque :

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

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

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

A l'exécution on obtient :



On peut réécrire le code précédent de la façon suivante :

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

12.3. L'instruction *yield*

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

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

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

```

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

```

13. Les boucles

13.1. Boucle *while*

```

while (booleanExpression)
    instruction1 ;

```

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

```

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

```

13.2. Boucle *do ... while*

```

do    instruction;
while (booleanExpression);

```

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

```

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

```

13.3. Boucle *for*

```

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

```

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

```

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

```

13.4. Boucle *for each*

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

13.5. Branchement inconditionnel

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