Introduction au langage JAVA

Présenté par : Djamel **MOUCHENE**

Propriétaire	High Tech Compass

Versions

Version	Date	Auteur	Modifications
V 1.0	27/07/2017	Djamel MOUCHENE	Version Initiale
V 1.1	23/01/2019	Djamel MOUCHENE	Relecture + Mise en page

Table ds matières

CHAPITRE I : Introduction au langage Java

Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création 1995 par Sun Microsystems. Il permet une programmation orientée-objet (à l'instar de SmallTalk et, dans une moindre mesure, C++), modulaire (langage ADA) et reprend une syntaxe très proche de celle du langage C.
Outre son orientation objet, le langage Java a l'avantage d'être modulaire (on peut écrire des portions de code génériques, c-à-d utilisables par plusieurs applications), rigoureux (la plupart des erreurs se produisent à la compilation et non à l'exécution) et portable (un même programme compilé peut s'exécuter sur différents environnements).

1.1 Environnement Java

Java est un langage interprété, ce qui signifie qu'un programme compilé n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur. La figure 1.1 illustre ce fonctionnement:

Exemple :

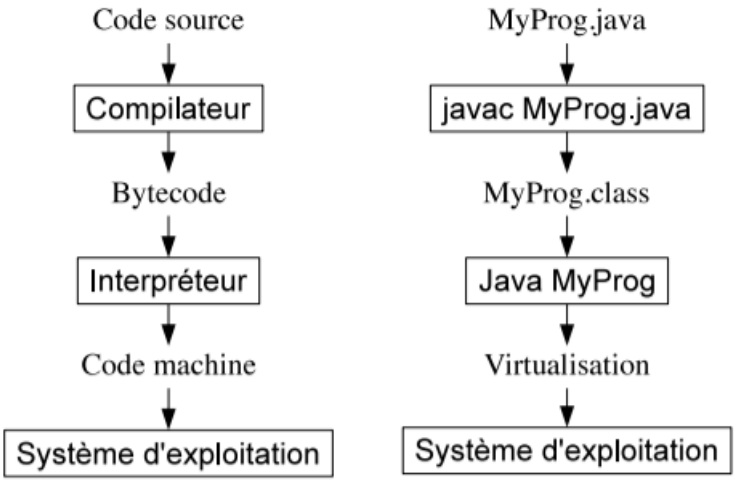

FIGURE 1.1 – Interprétation du langage

Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est .java. Ce code source est alors compilé par le compilateur ***javac*** en un langage appelé ***bytecode*** et enregistre le résultat dans un fichier dont l'extension est ***.class***. Le ***bytecode*** ainsi obtenu n'est pas directement utilisable. Il doit être interprété par la machine virtuelle de Java (***JVM***) qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation.
C'est la raison pour laquelle Java est un langage portable : le ***bytecode*** reste le même quel que soit l'environnement d'exécution.
En 2009, Sun Microsystems est racheté par Oracle Corporation qui fournit dorénavant les outils de développement Java SE (Standard Edition) contenus dans le Java Development Kit (JDK).

1.1.1 Compilation

La compilation s'effectue par la commande ***javac*** suivie d'un ou plusieurs nom de fichiers contenant le code source de classes Java. Par exemple, ***javac MyProg.java*** compile la classe ***MyProg*** dont le code source est situé dans le fichier ***MyProg.java***. La compilation nécessite souvent la précision de certains paramètres pour s'effectuer correctement, notamment lorsque le code source fait référence à certaines classes situées dans d'autres répertoires que celui du code compilé. Il faut alors ajouter l'option ***-classpath*** suivie des répertoires (séparés par un ; sous Windows et : sous Unix) des classes référencées. Par exemple :

javac -classpath /prog/exos1:/cours MyProg.java

compilera le fichier ***MyProg.java*** si celui-ci fait référence à d'autres classes situées dans les répertoires **/prog/exos1** et **/cours**. Le résultat de cette compilation est un fichier nommé ***MyProg.class*** contenant le ***bytecode*** correspondant au fichier source compilé. Ce fichier est créé par défaut dans le répertoire où la compilation s'est produite. Il est cependant fortement souhaitable de ne pas mélanger les fichiers contenant le code source et ceux contenant le ***bytecode***. Un répertoire de destination où sera créé le fichier ***MyProg.class*** peut être précisé par l'option ***-d***, par exemple :

javac -d /prog/exos1 -classpath /cours MyProg.java

1.1.2 Interprétation

Le bytecode obtenu par compilation ne peut être exécuté qu'à l'aide de l'interpréteur. L'exécution s'effectue par la commande java suivie du nom de la classe à exécuter (sans l'extension .class). Comme lors de la compilation, il se peut que des classes d'autres répertoires soient nécessaires.

Il faut alors utiliser l'option ***-classpath*** comme dans l'exemple qui suit :

java -classpath /prog/exos1 :/cours MyProg

CHAPITRE II : Syntaxe du langage

Le langage C a servi de base pour la syntaxe du langage Java :

- le caractère de fin d'une instruction est " ;" ➔ ***a = c + c ;***
- les commentaires (non traités par le compilateur) se situent entre les symboles "/*" et */" ou commencent par le symbole "//" en se terminant à la fin de la ligne

```
int a ; // Ce commentaire tient sur une ligne
int b ;

/* Ce commentaire nécessite
2 lignes */
int c ;
```

- les identificateurs de variables ou de méthodes acceptent les caractères {a..z}, {A..Z}, $, _ ainsi que les caractères {0..9} s'ils ne sont pas le premier caractère de l'identificateur. Il faut évidemment que l'identificateur ne soit pas un mot réservé du langage (comme ***int*** ou ***for***).

2.1 Types de données

2.1.1 Types primitifs

Le tableau ***2.1*** liste l'ensemble des types primitifs de données de Java. En plus de ces types primitifs, le terme ***void*** est utilisé pour spécifier le retour vide ou une absence de paramètres d'une méthode. On peut remarquer que chaque type primitif possède une classe qui encapsule un attribut du type primitif. Par exemple, la classe ***Integer*** encapsule un attribut de type ***int*** et permet ainsi d'effectuer des opérations de traitement et des manipulations qui seraient impossibles sur une simple variable de type ***int***. A l'inverse du langage C, Java est un langage très rigoureux sur le typage des données. Il est interdit d'affecter à une variable la valeur d'une variable d'un type différent si cette seconde variable n'est pas explicitement transformée. Par exemple :

```
int a ;
double b = 5.0 ;
a = b ;
```

est interdit et doit être écrit de la manière suivante :

```
int a ;
double b = 5.0 ;
a = (int)b ;
```

TABLE 2.1 – Type primitifs de données en Java

Type	Classe éq.	Valeurs	Portée	Défaut
boolean	Boolean	true ou false	N/A	false
byte	Byte	entier signé	{-128..128}	o
char	Character	caractère	{\uoooo..\uFFFF}	\uoooo
short	Short	entier signé	{-32768..32767}	0
int	Integer	entier signé	{-2147483648..2147483647}	0
long	Long	entier signé	$\{-2^{31}..2^{31}-1\}$	0
float	Float	réel signé	$\{-3,4028234^{38}..3,4028234^{38}\}$ $\{-1,40239846^{-45}..1,40239846^{-45}\}$	0.0
double	Double	réel signé	$\{-1,797693134^{308}..1,797693134^{308}\}$ $\{-4,94065645^{-324}..4,94065645^{-324}\}$	0.0

2.1.2 Tableaux et matrices

Une variable est déclarée comme un tableau dès lors que des crochets sont présents soit après son type, soit après son identificateur. Les deux syntaxes suivantes sont acceptées pour déclarer un tableau d'entiers (même si la première, non autorisée en C, est plus intuitive) :

```
int[] mon_tableau ;
int   mon_tableau2[];
```

Un tableau a toujours une taille fixe qui doit être précisée avant l'affectation de valeurs à ses indices, de la manière suivante :

```
int[] mon_tableau = new int[20];
```

De plus, la taille de ce tableau est disponible dans une variable **length** appartenant au tableau et accessible par

```
int taille = mon_tableau.length;
```

On peut également créer des matrices ou des tableaux à plusieurs dimensions en multipliant les crochets (ex : ***int[][] ma_matrice***;).
On accède aux éléments d'un tableau en précisant un indice entre crochets :

```
int valeurRang4 = mon_tableau[3]; //est le quatrième entier du tableau
```

et un tableau de taille n stocke ses éléments à des indices allant de 0 à n-1.

2.1.3 Chaînes de caractères

Les chaînes de caractères ne sont pas considérées en Java comme un type primitif ou comme un tableau. On utilise une classe particulière, nommée String, fournie dans le ***package java.lang***.
Les variables de type String ont les caractéristiques suivantes :

- Leur valeur ne peut pas être modifiée
- On peut utiliser l'opérateur + pour concaténer deux chaînes de caractères :

```
String s1 = "hello" ;
String s2 = "world" ;
String s3 = s1 + " " + s2 ;  //Après ces instructions s3 vaut "hello world"
```

- l'initialisation d'une chaîne de caractères s'écrit :

```
String s = new String(); //pour une chaine vide
String s2 = new String("hello world"); // La valeur de s2 est "hello word"
```

- Un ensemble de méthodes de la classe ***java.lang.String*** permettent d'effectuer des opérations ou des tests sur une chaîne de caractères

2.2 Opérateurs

2.1.1 Les opérateurs de calcul

Opérateur	Dénomination	Effet	Exemple	Résultat (int x=7)
+	opérateur d'addition	Ajoute deux valeurs	x+3	10
-	opérateur de soustraction	Soustrait deux valeurs	x-3	4
*	opérateur de multiplication	Multiplie deux valeurs	x*3	21
/	opérateur de division	Calcul le quotient de la division de deux valeurs	x/3	2
%	opérateur de congruence	Calcul le reste de la division de deux valeurs	x%3	1
=	opérateur d'affectation	Affecte une valeur à une variable	x=3	Met la valeur 3 dans la variable x

2.2.2 Les opérateurs d'assignation

Opérateur	Effet
+=	addition deux valeurs et stocke le résultat dans la variable (à gauche)
-=	soustrait deux valeurs et stocke le résultat dans la variable
*=	multiplie deux valeurs et stocke le résultat dans la variable
/=	divise deux valeurs et stocke le quotient dans la variable
%=	divise deux valeurs et stocke le reste dans la variable

2.2.3 Les opérateurs d'incrémentation

Opérateur	Dénomination	Effet	Syntaxe	Résultat (int x=7)
++	Incrémentation	Augmente d'une unité la variable	x++ ou ++x	8
--	Décrémentation	Diminue d'une unité la variable	x-- ou --x	6

2.2.4 Les opérateurs de comparaison

Opérateur	Dénomination	Effet	Exemple	Résultat

= = **À ne pas confondre avec le signe d'affectation =**	opérateur d'égalité	Compare deux valeurs et vérifie leur égalité	x==3	Retourne *true* si x est égal à 3, sinon *false*
<	opérateur d'infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur	x<3	Retourne *true* si x est inférieur à 3, sinon *false*
<=	opérateur d'infériorité	Vérifie qu'une variable est inférieure ou égale à une valeur	x<=3	Retourne *true* si x est inférieur ou égal à 3, sinon *false*
>	opérateur de supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur	x>3	Retourne *true* si x est supérieur à 3, sinon *false*
>=	opérateur de supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur	x>=3	Retourne *true* si x est supérieur ou égal à 3, sinon *false*
!=	opérateur de différence	Vérifie qu'une variable est différente d'une valeur	x!=3	Retourne *true* si x est différent de 3, sinon *false*

2.2.5 Les opérateurs logiques (booléens)

Opérateur	Dénomination	Effet	Syntaxe
||	OU logique	Retourne *true* si au moins une des deux conditions vaut *true* (ou *false* sinon)	condition1 || condition2
&&	ET logique	Retourne *true* si les deux conditions valent *true* (ou *false* sinon)	condition1 && condition2
!	NON logique	Retourne *true* si la variable vaut *false*, et *false* si elle vaut *true*)	!condition

2.3 Structures de contrôle

Les structures de contrôle permettent d'exécuter un bloc d'instructions soit plusieurs fois (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple). Dans tous ces cas, un bloc d'instruction est

- Soit une instruction unique ;
- Soit une suite d'instructions commençant par une accolade ouvrante "{" et se terminant par une accolade fermante "}".

2.3.1 Instructions conditionnelles

Syntaxe :

if (<condition>) <bloc1> [else <bloc2>] ou

<condition>?<instruction1>:<instruction2>

<condition> : doit renvoyer une valeur booléenne. Si celle-ci est vraie c'est ***<bloc1>*** qui est exécuté sinon ***<bloc2>*** est exécuté. La partie ***else <bloc2>*** est facultative.

Exemple

```
if (a == b) {
    a = 50;
    b = 0;
} else {
    a = a - 1;
}
```

2.3.2 Instructions itératives

Les instructions itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse. Les trois types d'instructions itératives sont les suivantes :

TantQue...Faire... L'exécution de cette instruction suit les étapes suivantes :

1. La condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 4 ;
2. Le bloc est exécuté ;
3. Retour à l'étape 1
4. La boucle est terminée et le programme continue son exécution en interprétant les instructions suivant le bloc.

Syntaxe :

while (<condition>) <bloc>

Exemple :

```
while (a != b) {
    a++;
}
```

L'exécution de cette instruction suit les étapes suivantes :

1. le bloc est exécuté ;
2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on retourne à l'étape 1, sinon on passe à l'étape 3;
3. la boucle est terminée et le programme continue son exécution en interprétant les instructions suivant le bloc.

Syntaxe:

do <bloc> while (<condition>);

Exemple:

```
do {
    a++;
} while (a != b);
```

Pour...Faire Cette boucle est constituée de trois parties : (i) une initialisation (la déclaration de variables locales à la boucle est autorisée dans cette partie) ; (ii) une condition d'arrêt ; (iii) un ensemble d'instructions à exécuter après chaque itération (chacune de ces instructions est séparée par une virgule). L'exécution de cette instruction suit les étapes suivantes :

1. les initialisations sont effectuées ;
2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 6;
3. le bloc principal est exécuté ;
4. les instructions à exécuter après chaque itération sont exécutées ;
5. retour à l'étape 2 ;
6. la boucle est terminée et le programme continue son exécution en interprétant les instructions suivant le bloc principal.

Syntaxe :

for (<init>;<condition>;<instr_post_itération>) <bloc>

Exemple:

```
for (int i = 0; i < 25; i++) {
    if (tab[i] != 0 ) {
        int tampon = tab[i];
    }
}
```

2.3.3 Instructions break et continue

L'instruction break est utilisée pour sortir immédiatement d'un bloc d'instructions (sans traiter les instructions restantes dans ce bloc). Dans le cas d'une boucle on peut également utiliserl'instruction continue avec la différence suivante :

- **break** : l'exécution se poursuit après la boucle (comme si la condition d'arrêt devenait vraie) ;
- **continue** : l'exécution du bloc est arrêtée mais pas celle de la boucle. Une nouvelle itération du bloc commence si la condition d'arrêt est toujours vraie.

Exemple:

```
for (int i = 0; i < 100; i++) {
    if (i > tab.length) {
        break;
    }

    if (tab[i] == 0) {
        continue;
    }
    System.out.println(tab[i]);
}
```