Programmation Orientée Objet en Java

Chapitre1 : le langage Java

Par Emna Fki

Le 30 Avril 2017

Ce travail est sous licence Creative Commons CC-BY-NC-SA

Chapitre1 :

Le langage Java

Le but de ce premier chapitre est de découvrir la *syntaxe* de base d'un langage de haut niveau, en l'occurrence du langage Java. La syntaxe d'un langage est l'ensemble des règles qu'il faut suivre pour écrire un programme correct dans ce langage. On verra la forme générale d'un programme Java et on en profitera également pour tout de suite prendre de bonnes habitudes et savoir différencier un programme Java *« bien écrit »* d'un autre.[1]

En particulier, on verra qu'un programme est constitué de différents types de mots qu'il faut agencer d'une certaine manière. Il y a des mots qu'on doit utiliser et qui ont déjà été définis par d'autres programmeurs et il y en a également que l'on doit choisir soi-même, les *identificateurs*. On verra comment construire un identificateur valide.

Enfin, on s'attardera sur les *types de données élémentaires*, c'est-à-dire les données qu'on est capable de manipuler en Java. Il y a, entre autre, les entiers et les caractères. La fin du chapitre traite des *variables* et *constantes* qui permettent de stocker des données. On doit stocker les données pour pouvoir les utiliser afin d'effectuer diverses opérations et calculs.

1. Structure d'un programme Java

On va commencer par observer un programme Java complet et succinct qui permet d'afficher deux phrases à l'écran. Chaque phrase est affichée sur une ligne séparée. La Figure 1 montre le code source du programme en question [1]

```
1  //---------------------------------------------------
2  // Auteur : Sébastien Combéfis
3  // Version : 4 avril 2015
4  //
5  // Affiche deux phrases à l'écran.
6  //---------------------------------------------------
7  public class Affiche
8  {
9      public static void main (String[] args)
10     {
11         System.out.println ("Voici un exemple de programme Java.");
12         System.out.println ("Ce programme provient de http://www.ukonline.be.");
13     }
14 }
```

Figure 1. Mon premier programme Java : le programme Affiche.

Voici ce que l'on peut lire à l'écran après exécution du programme :

```
Voici un exemple de programme Java.

Ce programme provient de http://www.ukonline.be.
```

1.1 Analyse du programme

Les six premières lignes du programme, commençant par les caractères //, sont des *commentaires*. Ils n'affectent en rien la tâche du programme, mais sont utiles pour que le code source du programme soit plus compréhensible par les humains.

Le reste du programme, de la ligne 7 à la ligne 14, est une *définition de classe* permettant de définir le programme Affiche. Le *corps de la classe* se situe entre l'accolade ouvrante ({) du début (ligne 8) et l'accolade fermante (}) de la fin (ligne 14). Nous verrons plus loin que les classes sont les éléments de base de la programmation Java.

Dans le corps de la classe, on retrouve une *méthode*. Il 'agit d'un groupe d'*instructions* auquel on a donné un nom. Dans cet exemple, elle s'appelle main et contient deux instructions (lignes 11 et 12). Toutes les instructions simples se terminent par un point-virgule (;). Tout programme Java doit comporter une méthode public static void main (String[] args) qui est la *méthode principale*, le point d'entrée du programme.

Les deux instructions de la méthode main font *appel* à une autre méthode, la méthode println. Cette dernière permet d'afficher des caractères à l'écran. Par contre, elle n'est pas définie dans ce programme, mais fait partie de l'objet System.out. On voit donc qu'on a des *définitions* et des *appels* de méthodes.

Voilà ce premier programme Java déjà décortiqué. Un grand nombre de concepts importants s'y cache, et ce n'est pas le but de tous les expliquer maintenant, mais bien de les citer.

1.2.Commentaire

Les *commentaires* sont très utiles pour le programmeur , mais ne servent à rien pour le fonctionnement du programme. Un programme, ce n'est pas un texte qu'on écrit puis qu'on oublie dans un grenier . En quelque sorte, on peut dire qu'un programme « vit » : il est modifié au cours du temps. Ces modifications peuvent survenir après un long intervalle et peuvent être faites par le même programmeur ou par un autre. Sans commentaires, il pourrait être difficile de comprendre le programme pour le modifier, d'où l'importance de bien commenter son code source.

Il y a deux manières d'insérer des commentaires dans le code source d'un programme. La première manière consiste à utilise les caractères // qui permettent de définir un commentaire sur une ligne. On peut, par exemple, placer un commentaire sur la même ligne qu'une instruction pour l'expliquer :

```
System.out.println ("Test");     // Affiche 'Test' à l'écran
```

L'autre manière permet d'insérer un commentaire qui s'étend sur plusieurs lignes. On marque le début du commentaire avec les caractères /* et la fin de celui-ci avec les caractères */ . Le texte du commentaire peut contenir n'importe quoi sauf la chaine de caractères */ , puisque celle-ci marque la fin du commentaire. Voici un exemple de *commentaire multilignes* :

```
/* Ceci est un autre commentaire.
        Il s'étend sur plusieurs lignes sans devoir mettre en début
        de chaque ligne le double slash //.
*/
```

Les commentaires sont très importants, mais il faut absolument les utiliser correctement et surtout systématiquement, chaque fois que vous écrivez un programme. La meilleure chose à faire est d'écrire les commentaires tout en écrivant le programme et non par après. Un commentaire doit apporter un plus par rapport au code. Les commentaires ne sont pas utilisés pour paraphraser le code, ce qui serait complètement inutile et alourdirait le code source pour rien, le rendant illisible.

1.3.Identificateur et mot réservé

Lorsqu'on programme, on utilise de nombreux mots. Dans le programme `Affiche`, ces identificateurs sont `public`, `class`, `Affiche`, `static`, `void`, `main`, `String`, `args`, `System`, `out` et `println`.

On peut classer ces mots en trois catégories :

- ceux que l'on choisit : `Affiche` et `args` ;
- ceux qu'un autre programmeur a choisis : `String`, `System`, `out`, `println` et `main` ;
- et ceux qui ont un sens spécial dans le langage : `class`, `public`, `static` et `void`.

Les mots des deux premières catégories, appelés *identificateurs*, doivent respecter la *règle de formation des identificateurs* que l'on va détailler plus loin. Les identificateurs `String`, `System`, `out`, `println` et `main` ont été choisis par d'autres programmeurs. Ils font partie d'un ensemble de codes prédéfinis, de classes et de méthodes que des gens ont écrits et que nous utilisons.

Les mots de la troisième catégorie, appelés *mots réservés* , font partie intégrante du langage pour lequel ils ont une signification bien définie. Le langage Java compte 50 mots réservés repris dans à la Figure 2. Sachez que `const` et `goto` ne sont pas utilisés dans la version actuelle de Java, mais sont réservés pour une éventuelle utilisation dans le futur.

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

Figure 2. Les 50 mots réservés du langage Java.

2. Type de données élémentaire

Lorsqu'on écrit des programmes, on a besoin de stocker et manipuler des *données*. En Java, toute donnée appartient à un *type* bien précis. Il y a les données dites de *type primitif* et les données de *type objet*. On va uniquement voir les données de type primitif pour le moment.

Il existe huit types primitifs en Java : quatre variantes de nombres entiers, deux variantes de nombres flottants (c'est-à-dire à virgule), un type booléen (vrai ou faux) et un type caractère. Cette section présente les types primitifs de Java et les *littéraux* associés. Un littéral est la représentation dans le code source d'une donnée de type primitif.

2.1 Entier et flottant

Il existe deux sortes de valeurs numériques : *les entiers* qui n'ont pas de partie fractionnaire, et *les flottants* qui en ont une. Les flottants sont parfois appelés nombres à virgule flottante ou nombres réels (attention, contrairement à la définition mathématique des nombres réels, les nombres flottants ont un nombre de décimales limité et par conséquent une précision limitée).

Il y a quatre types de données entières (`byte`, `short`, `int` et `long`) et deux types de flottants (`float` et `double`). Ces types diffèrent par la quantité d'espace mémoire nécessaire pour stocker une valeur du type, et donc par les différentes valeurs qu'il est possible de représenter. La Figure 3 reprend chacun de ces types avec la quantité de mémoire utilisée et les valeurs minimales et maximales représentables.

Type	Bits	Minimum	Maximum
byte	8	-128	127
short	16	-32 768	32 767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
float	32	-3.4e+38 (7 chiffres significatifs)	3.4e+38 (7 chiffres significatifs)
double	64	-1.7e+308 (15 chiffres significatifs)	1.7e+308 (15 chiffres significatifs)

Figure 3. Occupation mémoire et valeurs minimale et maximale représentables pour les types primitifs numériques de Java.

Comme vous pouvez le constater, tous les types numériques sont *signés*, c'est-à-dire qu'ils possèdent un signe et peuvent donc prendre des valeurs positives ou négatives.

Il est important de bien choisir le type de donnée à utiliser afin de ne pas gaspiller de la mémoire. En effet, stocker les résultats de mille lancés de dés dans mille `byte` est préférable à mille `int` puisqu'on prend quatre fois moins de place. Néanmoins, vu que la quantité de mémoire augmente considérablement, on se contentera la plupart du temps des `int` et `double`, ce qui permet souvent des calculs plus rapides, en pratique.

Il est néanmoins important de connaitre la différence entre ces types au cas où vous devriez utiliser du code écrit par d'autres programmeurs ou pour écrire des programmes destinés à des systèmes disposant de peu de mémoire.

2.2.Littéral de type numérique

Pour rappel, un *littéral* est une valeur explicite utilisée dans le code source d'un programme. On représente les entiers simplement en donnant leur valeur sous forme décimale, binaire, octale ou hexadécimale. Ces valeurs sont considérées par Java comme des valeurs de type `int`. On peut aussi avoir des littéraux de type `long` ; il suffit de faire suivre le nombre par la lettre `l` ou `L`.

Pour spécifier un nombre sous forme binaire, on le commence par `0b`. Pour la forme octale, on utilise `0` comme premier chiffre. Enfin, pour la forme hexadécimale, on commence par `0x` ou `0X` et on peut utiliser indifféremment les lettres `a` à `f` en majuscule ou en minuscule. Voici plusieurs littéraux qui représentent tous l'entier 45, avec le type `int` pour les quatre premier et `long` pour les quatre dernier :

```
45          // en décimal (int)
0b101101;   // en binaire (int)
055         // en octal (int)
0x2d        // en hexadécimal (int)

45L         // en décimal (long)
0b101101L;  // en binaire (long)
055L        // en octal (long)
0x2dL       // en hexadécimal (long)
```

En ce qui concerne les nombres flottants, ils sont considérés par défaut comme étant des valeurs de type `double`. On peut ajouter `f` ou `F` derrière le nombre pour qu'il soit de type `float`. On peut également écrire un nombre flottant en notation scientifique en le faisant suivre de `e` ou `E` suivi de la valeur de l'exposant qui peut être positif ou négatif. Voici plusieurs littéraux représentant le nombre flottant 25,5, avec le type `double` pour les deux premiers et `float` pour les deux derniers :

```
25.5      // en notation décimale (double)
2.55e+1   // en notation scientifique (double)

25.5F     // en notation décimale (float)
2.55e+1F  // en notation scientifique (float)
```

On peut également utiliser la lettre `d` ou `D` pour qu'un nombre soit considéré comme un `double`. Ceci est utile pour des entiers que l'on veut voir comme des flottants. Ainsi, le littéral `25D` représente le nombre flottant 25,0 et est de type `double`. Attention qu'on utilise le point comme séparateur décimal, comme en anglais, et pas la virgule comme en français.

Enfin, lorsqu'on veut écrire des littéraux pour des grands nombres, on peut utiliser comme on le souhaite le tiret de soulignement pour regrouper les chiffres pour des raisons de lisibilité. On pourra donc par exemple écrire le littéral suivant :

```
2_147_483_647
```

2.3. Caractère

Les *caractères* sont un autre type fondamental en Java. Comment sont-ils représentés dans la mémoire de l'ordinateur ? Ils sont *encodés* d'une certaine manière. Il faut tout d'abord choisir combien de bits on veut utiliser pour représenter un caractère, ce nombre définissant le nombre total de caractères représentables. Ensuite, il faut une table qui fasse le lien entre une séquence de bits et le caractère qu'il représente : une *table de caractères*.

Un exemple de table de caractères assez courant dans les pays anglophones est l'ASCII (American Standard Code for Information Interchange). Chaque caractère est encodé sur 7 bits, et peut donc prendre une valeur comprise entre 0 et 127. On peut donc encoder 128 caractères différents à l'aide de cette table de caractères.

La Figure 4 montre les 128 caractères représentables avec ASCII. Il s'agit d'une table à double entrées : pour retrouver la valeur d'un caractère, on prend la valeur tout à gauche de sa ligne, puis la valeur tout en haut de sa colonne et à l'intersection, on obtient la valeur du caractère en hexadécimal. Par exemple, le caractère `N` a pour valeur 4E en hexadécimal, ce qui correspond à 78 en décimal.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STH	ETH	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	CD2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	spc	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 4. La table de caractères ASCII (iso-646).

Les caractères sur fond vert sont des *caractères de contrôle*, qu'on appelle parfois aussi caractères invisibles ou caractères non-imprimables, car ils n'ont pas de symbole précis pour les représenter. Ils peuvent néanmoins être stockés comme toute autre valeur. On a par exemple le saut de ligne (`CR`) ou la tabulation horizontale (`HT`).

Comme on demande toujours plus de flexibilité, ASCII a été étendu sur 8 bits (iso-8859-1 ou iso-latin-1) augmentant le nombre de caractères représentables à 256. Parmi les caractères ajoutés par rapport à ASCII, il y a notamment les lettres accentuées non utilisées par les anglophones, mais bien par d'autres Européens de l'Ouest, comme les francophones.

Cependant, malgré cette extension, on ne peut représenter tous les caractères existants. On pense de suite aux langues asiatiques, arabes, cyrilliques, etc. qui utilisent des symboles. C'est pourquoi a été créé le format *Unicode* (http://www.unicode.org), qui a été choisi par les concepteurs de Java.

Le type primitif `char` permet de représenter des caractères au format Unicode, en utilisant 16 bits pour les représenter. On peut utiliser les caractères comme des nombres entiers non-signés compris entre 0 et 65535.

2.4.Littéral de type caractère

Un *littéral de type caractère* est simplement le caractère voulu entouré de guillemets simples (il ne faut pas confondre le guillemet simple ('), également appelé apostrophe, avec le guillemet ouvrant utilisé par les anglais (`)). Mais il est dès lors difficile de représenter les caractères de contrôle ou même encore des caractères non disponibles sur le clavier, comme le caractère marque déposée ( ™ ), par exemple (les littéraux `char` sont limités à ceux en UTF-16). Il y a donc des séquences spéciales appelées *séquences d'échappement* qui permettent de représenter certains caractères particuliers. Ces dernières sont reprises sur la Figure 5. La nouvelle ligne fait passer le curseur à la ligne suivante et le retour chariot fait revenir le curseur au début de la ligne courante.

Séquence	Unicode	Caractère
\n	0x000A	nouvelle ligne
\r	0x000D	retour chariot
\t	0x0009	tabulation
\b	0x0008	backspace
\f	0x000C	saut de page (form feed)
\xxx		le caractère dont la valeur en octal est xxx
\uxxxx		le caractère dont la valeur en hexadécimal est xxxx

Figure 5. Séquences d'échappement pour les caractères Java

Attention, sachez que vous ne pouvez pas utiliser `\u000A` et `\u000D` pour les caractères *nouvelle ligne* et *retour chariot* : vous devez utiliser `\n` et `\r` sans quoi vous serez face à une erreur de compilation (c'est parce que le traitement des séquences d'échappement Unicode se fait avant la compilation du code source du programme). Voici trois littéraux qui représentent le caractère `N` :

```
'N'
'\116'    // en octal
'\u004E'  // en hexadécimal
Type d'erreur
```

2.5.Booléen

Le dernier type primitif permet de représenter des données booléennes, c'est-à-dire des données qui ne peuvent prendre que les deux valeurs différentes vrai et faux. On utilise par exemple le type booléen pour indiquer si une condition particulière est vraie ou non. On peut aussi l'utiliser pour exprimer une situation qui comporte deux états différents, comme un interrupteur enclenché ou non.

Il n'y a que deux littéraux booléens, représentant simplement les valeurs vrai et faux :

```
true     // littéral pour vrai
false    // littéral pour faux
```

3. Variable et constante

On vient de voir les différents types de données. Afin de pouvoir les utiliser dans un programme, il faut pouvoir les stocker dans la mémoire. Pour cela, cette section décrit le concept très important de *variable*.

3.1 Variable

On peut voir une variable comme une boite dans laquelle on peut placer une donnée. Cette boite, on va lui donner *un nom* afin de l'identifier pour accéder et modifier son contenu.

La Figure 6 illustre ce concept. La variable, c'est la boite grise. Elle a pour *nom* `variable` (pas très original, mais bon, c'est pour l'exemple), et sa *valeur*, c'est-à-dire la donnée qu'elle stocke, c'est 17. Toute variable possède aussi un *type*, c'est celui de la donnée qu'elle stocke. Dans notre exemple, la valeur stockée est de type primitif `int`.

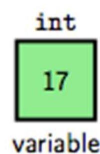

Figure 6. Une variable de nom variable, de type int et contenant la donnée 17.

Déclaration

Java est un langage où toute variable possède un et un seul type qui ne change jamais (on dit que Java est un langage à typage statique, contrairement à PHP (http://www.php.net) ou Python (http://www.python.org), par exemple, qui sont à typage dynamique). Et donc, pour pouvoir utiliser une variable, il va avant tout falloir la *déclarer*. Déclarer une variable, c'est définir son type, c'est-à-dire celui de la donnée qu'elle va contenir, et c'est aussi lui donner un nom. Voici comment on déclare une variable de type `int` dont le nom est `faces` :

```
int faces;
```

On pourrait par exemple utiliser cette variable pour stocker le nombre de faces d'un dé.

On a ainsi déclaré une nouvelle variable, mais celle-ci ne contient aucune donnée pour l'instant. On dit que la valeur de la variable est *indéfinie*, ou que la variable est *non-initialisée*.

Modifier le contenu d'une variable

Pour modifier le contenu d'une variable, il faut utiliser *un opérateur*, c'est le sujet du chapitre suivant. Sachez déjà qu'on utilise *l'opérateur d'affectation* (`=`) pour stocker une donnée dans une variable.

Pour affecter la donnée 6 à la variable dont le nom est `faces`, il faut écrire :

```
faces = 6;
```

On voit donc que pour utiliser une variable (c'est-à-dire manipuler son contenu), on utilise son nom. Une variable ne peut contenir que des données qui correspondent à son type. Toute tentative d'affecter à une variable une donnée ne correspondant pas à son type se soldera par une erreur de compilation (pour être exact, toute tentative d'affecter à une variable une valeur *non compatible* avec son type se soldera par une erreur).

De plus, une variable ne peut contenir qu'une seule donnée. Par conséquent, lorsque vous affectez une nouvelle donnée à une variable, la donnée qui s'y trouvait est supprimée au profit de la nouvelle.

Lorsqu'on déclare une variable, sa valeur est indéfinie. La première fois qu'on modifie son contenu, on *initialise* la variable. On peut combiner la déclaration et l'initialisation d'une variable en une seule instruction :

```
int faces = 6;
```

Afficher le contenu d'une variable

On peut utiliser les méthodes `print` et `println` de l'objet `System.out` pour afficher le contenu d'une variable ou pour afficher la valeur d'un littéral. Attention, si vous tentez d'afficher la valeur d'une variable non-initialisée, cela provoquera une erreur de compilation.

La différence entre `print` et `println` est que la première méthode va afficher une valeur tout en restant sur la même ligne, c'est-à-dire que la prochaine valeur qui sera affichée le sera juste à côté de celle qui vient de l'être. En utilisant `println`, un retour à la ligne est ajouté, et le curseur passe donc à la ligne suivante. La Figure 7 montre un exemple de programme affichant des variables et des littéraux à l'écran. Le résultat de l'exécution du programme est :

```
6.6
.
```

```
1  public class Print
2  {
3      public static void main (String[] args)
4      {
5          int faces = 6;
6          System.out.print (faces);
7          System.out.print ('.');
8          System.out.println (faces);
9          System.out.print ('.');
10     }
11 }
```

Figure 7. Afficher le contenu d'une variable à l'écran.

3.2. Constante

Parfois, on manipule des données qui restent constantes durant toute l'exécution du programme. Pour stocker de telles valeurs, on peut utiliser des *constantes* qui sont des variables dont on ne peut changer la donnée qu'elles contiennent. Pour signaler qu'une variable est une constante, on utilise le mot réservé `final` lors de sa déclaration :

```
final int FACES;
FACES = 6;
```

On ne peut plus changer la valeur d'une constante une fois qu'elle a été initialisée. Si on tente de le faire, on se retrouve face à une erreur de compilation. On peut aussi déclarer et initialiser une constante en une seule ligne :

```
final int FACES = 6;
```

Utiliser les constantes est bénéfique pour deux principales raisons. La première est que cela évitera d'accidentellement modifier la valeur d'une variable qui ne devrait pas changer. Le compilateur veillera en effet à ce que cela n'arrive pas. La deuxième est que cela permet au compilateur de faire des optimisations pouvant accélérer l'exécution du code.

3.3. Convention de nommage

Le nom d'une variable est un identificateur ; il faut donc que celui-ci suive la règle que nous avons vue en début de chapitre. De plus, il existe des conventions (que vous n'êtes pas obligés de suivre) pour donner un nom aux variables et constantes : c'est ce qu'on appelle les *conventions de nommage*.

Par convention, les noms de variables commencent par une minuscule et ne contiennent des majuscules qu'à chaque changement de mot, pour des noms composés. Cette façon de faire est appelée la *casse chameau*. Pour les constantes, le nom est complètement en majuscules avec les mots séparés par des tirets de soulignement (_).

Voici un exemple qui suit cette convention :

```
int vitesseMoteur;   // La vitesse du moteur
final int NB_ROUES;  // Le nombre de roues du véhicule
```

De plus, il convient de choisir des noms de variable qui soient le plus concis et explicites possibles. Un nom de variable comme `maSuperVariableQuiContientMonAge` est tout à fait à éviter et on préférera `monAge` ou même simplement `age`.

4. Type d'erreur

Lorsqu'on écrit un programme, on est confronté à différents types d'erreurs. On peut distinguer trois types d'erreurs : les *erreurs de compilation*, les *erreurs d'exécution* et, les plus terribles et difficiles à détecter, les *erreurs logiques*.

Avant de découvrir ces différents types d'erreurs, revenons encore une fois sur les différentes étapes pour avoir un programme Java. La première étape consiste à écrire le programme. On l'écrit sous forme d'un fichier texte : il s'agit du *code source*. Ensuite, on va le compiler grâce au compilateur Java (javac) ; cette phase est appelée *compilation*.

Une fois la phase de compilation terminée, on obtient un nouveau fichier binaire qui contient le programme Java sous une forme compréhensible par la machine : le *bytecode* (ou code

machine). On va pouvoir exécuter le programme en utilisant la machine virtuelle Java (java). La phase durant laquelle le programme est exécuté est appelée *exécution*.

La Figure 8 présente les différentes étapes à franchir pour obtenir, à partir du code source du programme Java, le code machine qui pourra être exécuté par l'ordinateur.[1]

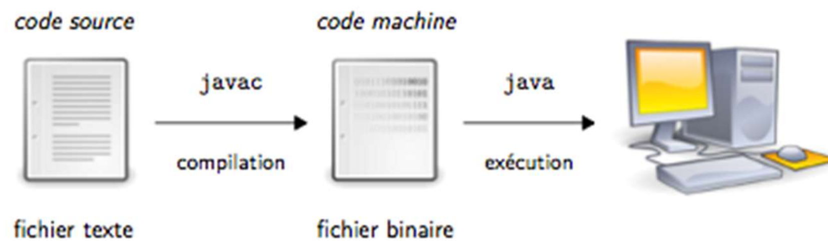

Figure 8. Compilation et exécution d'un programme Java.

- *Rôle de la machine virtuelle [2]*

La machine virtuelle travaille sur le *bytecode*, en général obtenu à partir de fichiers sources Java. Elle interprète le *bytecode* contenu dans les .class ou .jar. Elle peut aussi les compiler à la volée (*just-in-time* compiler, JIT). La plupart des machines virtuelles modernes peuvent interpréter ou compiler le *bytecode*. Enfin, certains outils permettent de compiler du *bytecode* en code natif.

- *Le bytecode [2]*

Le *bytecode* est une séquence d'instruction pour la machine virtuelle. La JVM stocke pour chaque classe chargée le flot de bytecode associé à chaque méthode.

4.1 Erreur de compilation

Les *erreurs de compilation* sont détectées durant la phase de compilation. De quel genre d'erreurs s'agit-il ? Écrire un programme Java, c'est comme écrire un roman ou un poème : on ne peut pas écrire n'importe quoi, n'importe comment; il y a des règles à respecter.

Pour rédiger un texte dans une langue donnée, on respecte les règles d'orthographe et de syntaxe (communément appelées grammaire) de cette langue. De même, pour écrire un programme informatique, on se conforme à la *syntaxe* propre au langage utilisé. Celle-ci détermine si un code source est valide ou non dans le langage utilisé.

On a déjà vu certaines règles de la syntaxe de Java, comme par exemple que les identificateurs ne peuvent pas contenir le caractère `&`, que toute accolade ouvrante doit avoir une accolade fermante qui lui correspond, que la méthode `main` doit se trouver dans une classe, etc.

D'autres erreurs, qui ne sont pas des erreurs de syntaxe, sont également trouvées par le compilateur. On en a aussi déjà rencontrées quelques-unes. Par exemple, une variable doit être déclarée avant d'être utilisée, on ne peut pas modifier la valeur d'une constante initialisée, etc.

La Figure 9 montre des erreurs de compilation avec le message d'erreur correspondant.

Message d'erreur	Description
`';' expected`	Point-virgule oublié à la fin d'une ligne de code

`incompatible types`
`found: XXX`
`required: YYY`
	Affectation d'une valeur de type xxx à une variable de type `YYY`

`cannot assign a value to a final variable XXX`	Affectation d'une valeur à une constante xxx déjà initialisée
`variable XXX might not have been initialized`	Utilisation de la variable xxx non-initialisée
`not a statement`	Ligne de code invalide
`illegal start of expression`	Le compilateur trouve quelque chose qui ne devrait pas se trouver à cet endroit

`cannot find symbol`
`symbol: XXX`
`location: YYY`
	Le compilateur trouve un mot xxx qu'il ne comprend pas

Figure 9. Quelques erreurs de compilation produites par le compilateur Java

La dernière erreur présentée dans le tableau peut se produire dans beaucoup de situations différentes : vous avez oublié de déclarer une variable, vous vous êtes trompé dans un mot réservé (par exemple `duble i;` au lieu de `double i;`), etc.

4.2.Erreur d'exécution

Une fois que le programme compile sans erreur, le bytecode est produit par le compilateur et on peut l'exécuter. On passe dans la phase d'exécution et le programme peut parfois se terminer de manière anormale et brutale : une *erreur d'exécution* s'est produite.

Ce genre d'erreur se produit lorsque le programme effectue une opération illégale. La Figure 10 montre un exemple qui va produire une erreur d'exécution. Dans ce programme, on tente de diviser une valeur par zéro. Sachez qu'on utilise `/` pour la division. La compilation de ce programme se passe sans erreur, et c'est au moment de l'exécuter qu'une erreur se produit :

```
> javac RuntimeError.java

> java RuntimeError
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at RuntimeError.main(RuntimeError.java:8)
```

```
public class RuntimeError
{
    public static void main (String[] args)
    {
        int taille = 0;
        int poids = 75;

        double rapport = poids / taille;
        System.out.print (rapport);
    }
}
```

Figure 10. Erreur d'exécution : une division par zéro.

Un autre exemple d'erreur d'exécution se produit lorsque vous tentez d'exécuter un programme qui ne contient pas de méthode `public static void main (String[] args)`. L'erreur est de type *Exception in thread "main" java.lang.NoSuchMethodError: main*.

Les erreurs d'exécution peuvent arriver, mais il est possible d'écrire nos programmes de manière *défensive* en utilisant les tests et le mécanisme d'exception. Un programme qui évite autant que possible les erreurs d'exécution est dit *robuste*.

Il est possible de s'assurer qu'un programme ne s'arrêtera jamais suite à une erreur d'exécution en utilisant par exemple des techniques de vérification formelle. Ces techniques sont essentiellement utilisées dans le monde académique ou pour des applications spécifiques ; elles ne sont pas encore très répandues dans le monde industriel (sauf dans certains cas particuliers, comme à la NASA, par exemple, où les systèmes développés sont des *systèmes critiques*. Ils utilisent par exemple l'outil *Java™ PathFinder* qui permet d'explorer toutes les exécutions possibles d'un programme Java pour détecter des situations non désirées (http://babelfish.arc.nasa.gov/trac/jpf)).

4.3.Erreur logique

Les *erreurs logiques* sont sans conteste les erreurs les plus difficiles à détecter et à corriger. Cependant, elles existent et apparaissent souvent sans même que le programmeur ne les remarque. Et pour cause, le programme compile et s'exécute sans aucun problème, mais ne fait pas ce pour quoi il a été conçu. Le programme de la Figure 11 calcule et affiche la surface d'un rectangle, surface dont la valeur est donnée par la formule $longueur \times largeur$, rappelez-vous de vos primaires.

```
public class LogicalError
{
    public static void main (String[] args)
    {
        int l = 5;
        int L = 7;

        int surface = l * l;
        System.out.print (surface);
    }
}
```

Figure 11. Erreur logique : mauvais calcul de la surface d'un rectangle.

Le programmeur a fait une faute en calculant la surface, il aurait dû mettre `L * l`. Peut-être la touche `Shift` du clavier était-elle un peu bloquée par des miettes de croissant. En tout cas, le programme compile et s'exécute sans erreur, mais il affiche 25, alors qu'il devrait afficher 35.

Pour détecter ces erreurs, il faut utiliser des programmes appelés *débuggeur* qui permettent d'exécuter le programme instruction par instruction tout en permettant de voir, entre autre, le contenu de toutes les variables à tout moment.

Une autre manière de faire est de décrire précisément ce que le programme doit faire, puis d'exécuter une série de tests qui couvrent tous les cas possibles et vérifient que, dans tous les cas, le programme fait bien ce qu'il faut (le framework JUnit est utilisé pour effectuer des tests de manière systématique pour des programmes Java (http://www.junit.org)).

5. Allure d'un programme

Cette dernière section va s'intéresser à quelques *conventions* sur la manière d'écrire le code source d'un programme de manière *« propre »*. Il s'agit évidemment de lignes de conduite que l'on peut suivre ou non.

La première chose à ne pas oublier est qu'on va écrire une classe dans un fichier texte et que le nom du fichier doit absolument être le même que le nom de la classe. Par exemple, le programme de la Figure 12 doit absolument être enregistré dans un fichier dont le nom est `Allure.java`.

```
1 public class Allure
2 {
3     public static void main (String[] args)
4     {
5         int value = 123456;
6         System.out.print ("Voici un programme Java");
7     }
8 }
```

Figure 12. Le fichier `Allure.java`.

Le texte d'un programme est composé de mots, composants élémentaires. Ce sont les noms, les mots réservés, les littéraux, les opérateurs et les signes de ponctuation. La *contrainte d'indivisibilité* entraine que les mots d'un programme ne peuvent en aucun cas contenir des blancs (espace, tabulation, saut de ligne, etc.).

On a déjà parlé de quelques conventions dans ce chapitre; on va maintenant les reprendre. Il ne s'agit que de conventions, vous n'êtes de nouveau pas du tout obligés de les suivre. Sachez néanmoins que la plupart des gens les suivent, ce qui permet d'échanger des programmes qui seront plus facilement lisibles par tout le monde.

Par convention, lorsqu'il faut choisir un nom pour une classe, on va commencer par une majuscule, tandis que pour une variable, on va commencer par une minuscule. Pour les constantes, on choisit un nom complètement en majuscule.

Rappelez-vous également que Java est sensible à la casse ; donc, si on a besoin de trois variables, on peut les appeler var, vAR, vAr... Cette pratique est néanmoins tout à fait déconseillée puisqu'elle conduit à des distractions et rend le code illisible et incompréhensible.

Références bibliographiques :

[1] « Apprendre Java et la Programmation Orientée Objet », 2015 , By Sébastien Combéfis, sous licence Creative Commons CC-BY-NC-SA, www.ukonline.be/cours/java/apprendre-java,

[2] « Prog. orientée objet avancée: Java », By Jean-Francois Lalande - April 2016, sous licence Creative Commons CC-BY-NC-SA, www.univ-orleans.fr/lifo/Members/Jean-Francois.Lalande/enseignement/java/presentation-java.html#slide1,