Errata 3

Les types

« Un type qui se trompe en disant quelque chose de faux dit peut-être quelque chose de vrai. »

Philippe Geluck

En langage Java, toute donnée a un type.

Les données manipulées dans un programme peuvent être écrites dans le code du programme ou être mémorisées le temps de l'exécution de celui-ci ou encore être une entrée du programme.

- Une valeur écrite dans un programme est appelée « *littéral* ». Cette *valeur* a une valeur et un type.
- Pour qu'une valeur soit mémorisée le temps de l'exécution — ou tout du moins pendant une partie de l'exécution — elle doit être stockée dans une variable (voir section 7.4 page 84). Une variable a une valeur *variable* et un type.

Contenu

3.1 Les types primitifs **2**
3.1.1 Les types primitifs numériques entiers 2
3.1.2 Les littéraux numérique entiers (excepté char) . . . 3
3.1.3 Le type primitif numérique entier particulier char . 5
3.1.4 Les littéraux numériques entiers char 5
3.1.5 Les types primitifs numériques à virgule flottante . 7
3.1.6 Les littéraux numériques à virgule flottante 7
3.1.7 Le type primitif booléen 9
3.1.8 Les littéraux booléens 9
3.2 Les types références **9**
3.2.1 Le type référence `String` 10
3.2.2 Les littéraux de type `String` 10
3.2.3 Les types références *tableau* 11
3.2.4 Durée de vie des données sur le tas 12

3.1 Les types primitifs

Il existe, en Java, 8 types primitifs : des types primitifs numériques entiers, numériques à virgule flottante, les caractères et les booléens. Voici ce que dit la grammaire :

> *PrimitiveType:*
> *NumericType*
> `boolean`
>
> *NumericType:*
> *IntegralType*
> *FloatingPointType*
>
> *IntegralType:*
> *(one of)*
> `byte short int long char`
>
> *FloatingPointType:*
> *(one of)*
> `float double`

Chaque type a une taille déterminée.

La figure 3.1 représente tous les types primitifs en Java. Voyons les en détails.

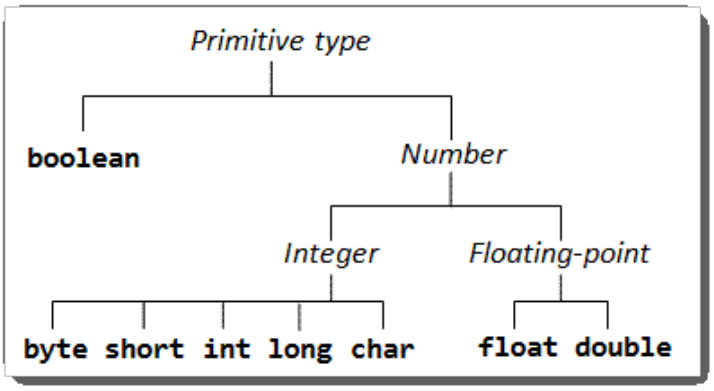


FIGURE 3.1 – Les types primitifs Java

3.1.1 Les types primitifs numériques entiers

Il existe 5 types primitifs entiers. Les 4 premiers représentent des nombres entiers signés codés en notation en complément à 2. Il s'agit des types : **byte**, **short**, **int** et **long**.

La quantité mathématique montrée à la figure 3.3 page ci-contre peut être représentée de différentes manières : en utilisant des chiffres romains — VII — ou arabes en base 10 — 7 — ou en base 2 — 111 — ou encore de moult autres façons.

Pour représenter un nombre négatif, l'habitude est de l'affubler du signe '-' devant mais il est également possible d'utiliser la notation en complément à 2.

Les types byte, short, int et long représentent une partie des nombres entiers signés. Ils sont stockés en notation en complément à 2. Ils se différencient par la taille qu'ils occupent en mémoire et donc par l'intervalle de nombres qu'ils représentent.

La figure 3.2 rassemble les types primitifs entiers (excepté char) avec leurs intervalles.

Type	Taille	Intervalle
byte	8 bits	[-128, 127]
		$[-2^7, 2^7 - 1]$
short	16 bits	[-32 768-, 32 767]
		$[-2^{15}, 2^{15} - 1]$
int	32 bits	[-2 147 483 648, 2 147 483 647]
		$[-2^{31}, 2^{31} - 1]$
long	64 bits	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]
		$[-2^{63}, 2^{63} - 1]$

FIGURE 3.2 – Types primitifs entiers (exceptés char) et leur taille

Le type int est le type numérique entier privilégié. C'est celui qui est le plus utilisé et est le type par défaut dans les opérations mathématiques usuelles. L'intervalle qu'il représente est généralement suffisant [1].

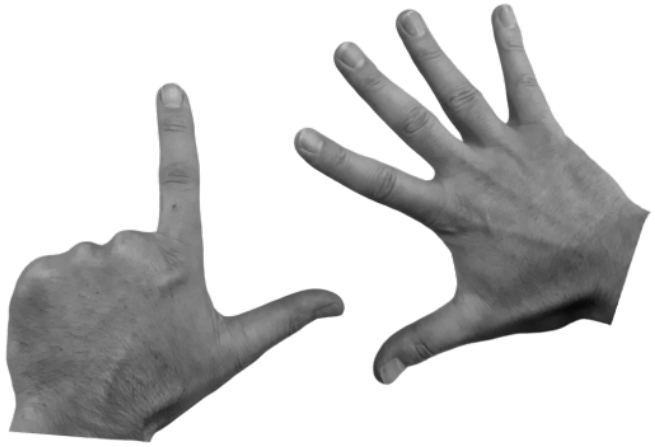


FIGURE 3.3 – La quantité 7 montrée avec les doigts

3.1.2 Les littéraux numérique entiers (excepté char)

Les nombres entiers sont représentés en utilisant les chiffres habituels. Il est cependant possible d'écrire un nombre dans différentes bases. Il faudra alors pouvoir les distinguer. En mathématiques il est d'habitude d'utiliser un indice pour préciser la base comme : $7_{10} = 111_2$. En informatique nous utiliserons un **préfixe**.

1. Ce qui n'est pas toujours vrai puisqu'en décembre 2014, la vidéo *Gangnam Style* est la première vidéo Youtube à dépasser ±2 milliard de vues (2 147 483 647 pour être précis) et oblique Google à revoir la variable dans laquelle elle stocke ce nombre de vues pour passer à 64 bits.

Littéral numérique décimal

Respecte les règles suivantes :

- ▷ les chiffres `0123456789` et `_` ;
- ▷ un littéral est de type int ou long, jamais de type byte ou short [2] ;
- ▷ pour distinguer un littéral long du type par défaut int, suffixer d'un `l` ou `L` ;

Exemple, la quantité 100 :

```java
int myInt = 100;
int myOtherIntSomeValue = 1_00;
long myLong = 100L;
```

Littéral numérique octal

Respecte les règles suivantes :

- ▷ les chiffres `01234567` et `_` ;
- ▷ commence par un `0` ;
- ▷ un littéral est de type int ou long, jamais de type byte ou short ;
- ▷ pour distinguer un littéral long du type par défaut int, suffixer d'un `l` ou `L` ;

Exemple, la quantité 100 :

```java
int myOctalInt = 0144;
int anoherOctalInt = 01_44;
long myOctalLong = 0144L;
```

Littéral numérique hexadécimal

Respecte les règles suivantes :

- ▷ les chiffres `0123456789ABCDEFabcdef` et `_` ;
- ▷ commence par un `0x` ou `0X` ;
- ▷ un littéral est de type int ou long, jamais de type byte ou short ;
- ▷ pour distinguer un littéral long du type par défaut int, suffixer d'un `l` ou `L` ;

Exemple, la quantité 100 :

```java
int myHexadecimalInt = 0x64;
long myHexadecimalLong = 0X64l;
```

2. Ceci ne nous empêchera pas d'écrire byte b = 5 par exemple. Nous verrons les conversions en Développement II

Littéral numérique binaire

Respecte les règles suivantes :

- ▷ les chiffres `01` et `_` ;
- ▷ commence par un `0b` ou `0B` ;
- ▷ un littéral est de type int ou long, jamais de type byte ou short ;
- ▷ pour distinguer un littéral long du type par défaut int, suffixer d'un `l` ou `L` ;

Exemple, la quantité 100 :

```java
int myBinaryInt = 0b01100100;
int anotherBinaryInt = 0B0110_0100;
```

Remarque Pour toutes les variables de cette section, dès lors que la quantité — *100* dans nos exemples — est stockée dans les variables : `myInt`, `myOtherIntSomeValue`, `myLong` ... `anotherBinaryInt`, c'est toujours la même quantité.

Afficher toutes ses variables, affichera `100`. Par exemple :

```java
System.out.println(myBinaryInt); // 100
```

3.1.3 Le type primitif numérique entier particulier char

Le type char est un entier non signé de 16 bits représentant le code Unicode codé en UTF-16 du caractère[3]. Un caractère Unicode codé en UTF-16 fait une taille de 16 bits ou de 32 bits en fonction du caractère qu'il représente.

Le type char en Java ne permet de ne représenter que le sous ensemble BMP (*Basic Multilingual Plane*) des caractères Unicodes codés en UTF-16. Ceux ayant leur code compris entre `\u0000` et `\uffff`. La figure 3.7 page 9 montre quelques caractères et leurs code Unicode.

Type	Taille	Intervalle
char	16 bits	[0, 65 535] $[0, 2^{16} - 1]$

FIGURE 3.4 – Type primitif entier char et sa taille

3.1.4 Les littéraux numériques entiers char

Un littéral de type char se caractérise par les guillemet-apostrophes (*single quote* ou guillemet-simples) qui l'entourent.

Le caractère *a* par exemple, se représente `'a'`.

3. Pour en savoir plus sur l'Unicode, UTF8, UTF16 et UTF32, lire « Unicode, UTF8, UTF16, UTF32... et tutti quanti » [Bet09]

Et l'on pourra déclarer une variable de type char et l'initialiser avec le caractère *a* par une instruction de la forme :

```java
char myChar = 'a';
```

La variable myChar de type char est initialisée avec le littéral 'a' de type char également.

À ceci s'ajoute la possibilité **d'échapper des caractères**. Un caractère peut parfois avoir plusieurs significations. Par exemple le caractère *single quote* (') signale le début d'un littéral de type char mais peut vouloir signaler simplement le caractère '. Comment distinguer les deux ?

Lorsque l'on écrit simplement un caractère, il a sa signification première, son sens premier : a signifie *a*, b signifie *b*, Z signifie *Z*, ' signifie *voici le début ou la fin d'un caractère*, etc. Certains caractères ont un **deuxième sens** lorsque qu'ils sont *échappés*, c'est-à-dire précédés d'un *backslash* (\). La figure 3.5 montre les principaux.

Caractère	Sens premier	Sens second
n	n	passage à la ligne
t	t	tabulation
'	début ou fin d'un littéral char	'
"	début ou fin d'une chaine	"
\	signale une séquence d'échappement (ou *attention, le caractère qui suit n'a pas son sens premier*)	\

FIGURE 3.5 – Quelques séquences d'échappement (la liste complète [GJS+17] section 3.10.6)

3.1.5 Les types primitifs numériques à virgule flottante

Les nombres pseudo-réels, ou encore les nombres à virgule flottante, sont un sous ensemble des réels parce que — comme pour les entiers — il y aura un *plus petit nombre représentable* et *un plus grand*. C'est l'**intervalle** de nombre que l'on peut représenter. À ceci, s'ajoute la **précision** que l'on pourra atteindre. En effet, les réels est un ensemble continu de nombre — entre deux nombre réels, il existe toujours, au moins, un autre réel — tandis que les pseudo-reéls est un ensemble discret.

Les nombres à virgule flottante (*floating numbers*) sont codés suivant la norme IEEE 754 [4].

Selon cette norme, un nombre est représenté avec un signe, une mantisse et un exposant. Le tout en base 2. Un bit est utilisé pour le signe.

$$nombre = signe\ mantisse\ 2^{exposant}$$

4. https://fr.wikipedia.org/wiki/IEEE_754

Il existe 2 types primitifs à virgule flottante : **`float`** et **`double`** dont les tailles sont données à la figure 3.6.

Type	Taille (*bit*)	Exposant	Mantisse
float	32 bits	8	23
double	64 bits	11	52

FIGURE 3.6 – Types primitifs numériques à virgule flottante et leur taille

3.1.6 Les littéraux numériques à virgule flottante

Les littéraux à virgule flottante sont composés de plusieurs parties : une partie entière, un point décimal (ou hexadécimal), une partie décimale, un exposant et un suffixe.

| partie entière | . | partie décimale | exposant | suffixe |

Un littéral à virgule flottante peut être exprimé en base 10 (décimal) ou en base 16 (hexadécimal).

Un littéral à virgule flottante est de type float s'il est suffixé d'un f ou F sinon, il est de type double. Et ce qu'il soit suffixé d'un d ou D ou non.

Pour un **littéral décimal à virgule flottante**, au minimum un chiffre dans la partie entière ou décimale et soit le point décimal, soit l'exposant, soit le suffixe sont requis. Les autres parties sont optionnelles.

- ▷ la partie entière et la partie décimale sont des littéraux décimaux entiers (le caractère _ étant autorisé) ;
- ▷ le point décimal est un point (.) ;
- ▷ l'exposant est la lettre e ou E suivie par un littéral décimal entier (le caractère _ étant autorisé) ;
- ▷ le suffixe est f, F, d ou D

Exemples :

```java
double myDouble;
myDouble = 1.;      // 1.0
myDouble = .1;      // 0.1
myDouble = 1e1;     // 10.0
myDouble = 1d;      // 1.0
myDouble = 1.e0;    // 1.0
myDouble = 1_000.45;  // 1000.45
myDouble = 1.45e3;    // 1450.0
myDouble = .45e3d;    // 450.0

float myFloat;
myFloat = 1f;     // 1.0
myFloat = 1.f     // 1.0
```

java

Pour un **littéral hexadécimal à virgule flottante**, au minimum un chiffre dans la partie entière ou décimale et l'exposant sont obligatoires. Le suffixe est optionnel.

- ▷ le littéral débute par 0x ou 0X ;
- ▷ la partie entière et la partie décimale sont des chiffres hexadécimaux (le caractère _ étant autorisé) ;
- ▷ le point décimal est un point (.) ;
- ▷ l'exposant est la lettre p ou P suivie par un littéral décimal entier (le caractère _ étant autorisé) représentant une puissance de 2 ;
- ▷ le suffixe est f, F, d ou D

Exemples :

```java
double myDouble;
myDouble = 0x1p0;   // 1.0 = 1 * 2^0
myDouble = 0x1.1p0; // 1.0625 = 1 + (1/16)
myDouble = 0x1p1;   // 2.0 = 1 * 2^1
myDouble = 0xA.Bp0; // 10.6875= 10 + 11*1/16
myDouble = 0x1E2p2; // 1928 = (1*16^2 + 14*16^1 + 2*16^0) * 2^2

float myFloat;
myFloat = 0X.1p4f;    // 1.0 = (1*16^-1) * 2^4
```

3.1.7 Le type primitif booléen

Le type primitif booléen permet de représenter les deux valeurs logiques *vrai* et *faux*. Ce sont les deux seules valeurs de ce type.

3.1.8 Les littéraux booléens

Les littéraux booléens sont simplement : true pour *vrai* et false pour *faux*.

Exemple :

```java
boolean myBoolean = true;
```

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

FIGURE 3.7 – Table Unicode *Basic Latin (ASCII)*

3.2 Les types références

Pour rappel (cfr. section 7.4 page 84) une variable de type référence est une variable qui ne contient pas directement la valeur qui lui est assignée mais une référence vers cette valeur (qui se trouve à un autre endroit).

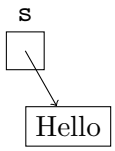
Tous les types qui ne sont pas un des 8 types primitifs présentés à la section précédente sont des types références. Ce sont les types les plus répandus bien que nous n'en verrons que quelques-uns dans ce premier cours de développement.

Dans cette section, nous parlerons essentiellement de `String` et de *tableaux* bien que l'on ait déjà rencontré d'autres types références comme `Scanner` ou `Random`. Pas d'inquiétude, nous leur règlerons leur compte plus tard, en développement II.

3.2.1 Le type référence `String`

`String` est un type référence.

Pour rappel, déclarer une variable de type `String` a pour effet de réserver un emplacement mémoire sur la pile (*stack*). Initialiser cette variable avec une valeur a pour effet de placer cette valeur sur le tas (*heap*). La variable *référencera* cette valeur comme nous pouvons le voir sur la figure ci-contre.



La variable `s` se trouve sur la pile.

La valeur *"Hello"* se trouve sur le tas.

Il est bien sûr possible de changer la valeur d'une variable de type référence, et dans ce cas, c'est la référence qui change. Si la valeur initiale de `s` est *Hello* et qu'elle devient *Bye bye*, la situation sera par exemple comme illustré à la figure 3.8. Nous y reviendrons.

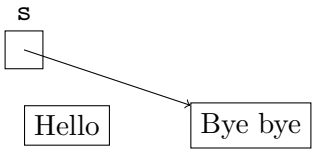


FIGURE 3.8 – Exemple : `s` reçoit *Hello* et ensuite Bye bye

3.2.2 Les littéraux de type `String`

Un littéral de type `String` se caractérise par les guillemets doubles (*double quote*) qui l'entourent (voir [GJS+17] section 3.10.5).

La chaine *Hello* par exemple, se représente `"Hello"`. Et l'on pourra déclarer une variable de type `String` et l'initialiser avec la chaine *Hello* par une instruction de la forme :

```java
String myString = "Hello";
```

Une chaine de caractères peut contenir n'importe quel caractère en ce compris, des caractères d'échappement. La chaine `"Hello\nWorld"` a donc un sens et en cas d'affichage de cette chaine, nous verrons un passage à la ligne entre *Hello* et *World*.

Un littéral de type `String` donné — par exemple `"Hello"` — référence toujours le même emplacement mémoire. Par exemple, le code suivant s'illustre par la figure 3.9.

```java
String s1 = "Hello";
String s2 = "Hello";
```

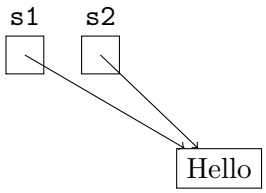


FIGURE 3.9 – Exemple : `s1` et `s2` reçoivent *Hello*

Remarque : ceci est vrai pour les littéraux de type `String` pas pour des valeurs de type `String` qui seraient, par exemple, reçues au *runtime*.

Le code suivant par exemple s'illustrerait par la figure 3.10.

```java
Scanner keyboard = new Scanner(System.in);
String s1 = keyboard.nextLine();  // INPUT Hello
String s2 = keyboard.nextLine();  // INPUT Hello
```

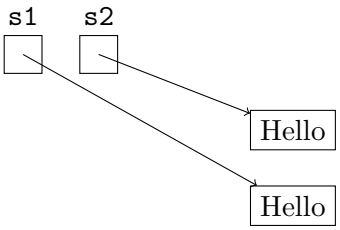


FIGURE 3.10 – Exemple : `s1` et `s2` reçoivent *Hello*

3.2.3 Les types références *tableau*

Les tableaux sont tous de types références.

Pour rappel (cfr. section 10 page 113), **déclarer** un variable de type *tableau de* a pour effet de réserver un emplacement mémoire sur la pile (*stack*). **Créer** le tableau réservera l'emplacement mémoire sur le tas (*heap*) et l'initialiser placera les valeurs dans les cases du tableau comme illustré sur la figure 3.11 page suivante.

Il est bien sûr possible de changer la valeur d'un élément du tableau, voire même de changer la valeur du tableau.

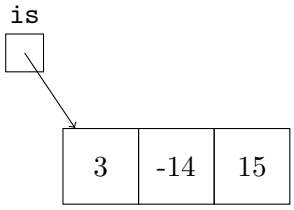


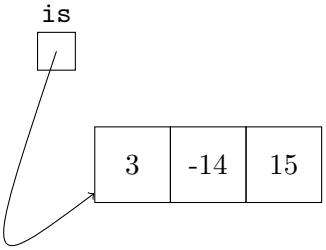
FIGURE 3.11 – Exemple : `is` est un tableau de `int` (par exemple) ayant reçu les valeurs 3, -14 et 15

Déclaration du tableau `is`.
C'est un tableau d'entiers. Déclaré par :

```java
int[] is;
```

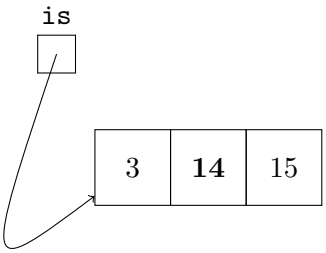
Création du tableau d'entiers et **initialisation**. La variable `is` contient une référence vers la mémoire qui contient le tableau.

```java
int[] is = {3, -14, 15};
```



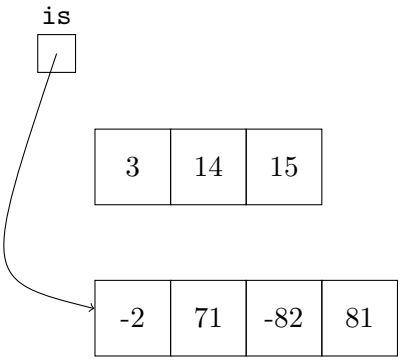
Remplacement d'une valeur de type primitif dans le tableau. La variable `is[1]`, de type primitif (`int`) reçoit directement une nouvelle valeur.

```java
int[] is = {3, -14, 15}
is[1] = 14;
```



Remplacement du tableau par un autre tableau. La variable `is`, de type référence (`int[]`), reçoit une référence vers le nouveau tableau.

```java
int[] is = {3, -14, 15};
is[1] = 14;
is = new int[] {-2, 71, 82,
    81};
```



3.2.4 Durée de vie des données sur le tas

Les données de type référence sont créées sur le tas. Chaque fois que l'on crée un tableau, une nouvelle chaine ou une nouvelle donnée de type référence, un emplacement mémoire est réservé sur le tas pour contenir les valeurs. La variable contiendra une référence vers ces valeurs.

Nous avons vu qu'il était possible de changer la valeur d'une variable de type référence. Dans ce cas, l'ancienne valeur n'est plus référencée. Lorsque plus aucune variable ne référence une valeur en mémoire, elle n'est plus utile à personne. Comment la récupérer pour libérer de l'espace mémoire qui, bien que grand, n'est pas infini ?

Dans certains langages, c'est la tâche du développeur ou de la développeuse de *libérer* les emplacements mémoires qu'iel n'utilise plus. Dans d'autres langages, c'est le langage qui s'en occupe.

En langage Java, c'est le langage qui se charge de la gestion de la mémoire. Lorsqu'il le décide, il lance un programme qui s'appelle le ramasse-miette (*garbage collector*[5]) qui parcourt la mémoire à la recherche des valeurs non référencées et rend les emplacements mémoires à nouveau disponibles. Il est possible de demander explicitement le passage du ramasse-miette mais ce sera la machine virtuelle qui décidera de son passage ou non.

5. Notez que la traduction n'est pas littérale.

Références

[Bet09] Pierre Bettens. Unicode, utf8, utf16, utf32, ... et tutti quanti. `http://blog.namok.be/?post/2009/11/30/unicode-UTF8-UTF16-UTF32-et-tutti-quanti`, novembre 2009.

[GJS+17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification. Java SE 9 Edition*. Oracle America Inc., 2017.