

Quelques rappels de syntaxe

Mise à niveau Java
Y. Boichut & F. Moal



Les bases de Java : syntaxe

- Structures lexicales
 - Les types de données
 - Les instructions
-
- Jeux

Structures lexicales

Codage Unicode

Pour les identificateurs, les commentaires, les valeurs de type caractère ou chaîne de caractères, Java utilise les caractères du code Unicode

Le reste d'un programme Java est formé de caractères ASCII (qui sont les 128 premiers caractères du code Unicode)

le caractère Unicode dont le code est la valeur hexadécimale xxxx peut être représenté par \uxxxx

Identificateurs

- Un identificateur Java
- est de longueur quelconque
- commence par une lettre Unicode (caractères ASCII recommandés)
- peut ensuite contenir des lettres ou des chiffres ou le caractère souligné « _ »
- ne doit pas être un mot-clé ou les constantes : true, false ou null

Mots clés Java

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

*: pas encore utilisé

** : depuis Java SE 5

Commentaires

- Sur une seule ligne :

```
// Voici un commentaire
```

```
int prime = 1500; // prime fin de mois
```

- Sur plusieurs lignes :

```
/* Première ligne du commentaire
```

```
suite du commentaire */
```

Commentaires

- Documentation automatique par javadoc

```
/**  
 * Valide un mouvement de jeu d'Echecs.  
 * @param leDepuisFile    File de la pièce à déplacer  
 * @param leDepuisRangée  Rangée de la pièce à déplacer  
 * @param leVersFile      File de la case de destination  
 * @param leVersRangée    Rangée de la case de destination  
 * @return vrai (true) si le mouvement d'échec est valide ou faux (false) sinon  
 */
```


Conventions et Bonnes pratiques

- nom de variables en minuscules : total
- les mots séparés par des majuscules : monTotalFinal
- noms des classes commencent par des MAJUSCULES : MaPremiereClasse
- évitez les accents !
- nom des constantes (final) en majuscules : VALEUR
- séparation par _ (seule utilisation !) : VALEUR_PAR_DEFAULT
- COMMENT, COMMENT, COMMENT javadoc

Types de données

2 grands types de données

Toutes les données manipulées par Java ne sont pas des objets

- 2 grands groupes de types de données :
 - types primitifs
 - objets (instances de classe)
- Java manipule différemment les valeurs des types primitifs et les objets : les variables contiennent des valeurs de types primitifs ou des références [pointeurs] aux objets

Types primitifs

- **boolean** (true/false)
- Nombres entiers : **byte** (1 octet), **short** (2 octets), **int** (4 octets), **long** (8 octets)
- Nombres non entiers, à virgule flottante : **float** (4 octets), **double** (8 octets).
- Caractère (un seul) : **char** (2 octets) ; codé par le codage Unicode (pas ASCII)

Types primitifs numériques

- byte : compris entre -128 et 127
 - short : compris entre -32.768 et 32.767
 - int : compris entre -2.147.483.648 et 2.147.483.647
 - long : valeur absolue maximum $9,2 \times 10^{18}$ (arrondie)
-
- float : environ 7 chiffres significatifs ; valeur absolue (arrondie) inférieure à $3,4 \times 10^{38}$ et précision maximum (arrondie) de $1,4 \times 10^{-45}$
 - double : environ 17 chiffres significatifs ; valeur absolue inférieure à $1,8 \times 10^{308}$ (arrondie) et précision maximum de $4,9 \times 10^{-324}$ (arrondie)

Valeurs numériques

- Une constante « entière » est de type long si elle est suffixée par « L » et de type int sinon
- Une constante « flottante » est de type float si elle est suffixée par « F » et de type double sinon

35

2589L // constante de type long

456.7 ou 4.567e2 // 456,7 de type double

.123587E-25F // de type float

012 // 12 en octal = 10 en décimal

0xA7 // A7 en hexa = 167 en décimal

Valeurs numériques

- Il est possible d'insérer le caractère « souligné » dans l'écriture des constantes nombres entiers :

1_567_688 ou 0x50_45_14_56

- On peut aussi écrire un nombre en binaire :

0b01010000010001010001010001010110 ou

0b01010000_01000101_00010100_01010110

Valeurs de type char

- Un caractère Unicode entouré par « ' »
- CR et LF interdits (caractères de fin de ligne)

'A'

'\t' '\n' '\r' '\\' '\"' '\'''

'\u20ac' (\u suivi du code Unicode hexadécimal d'un caractère ; représente €)

'α'

Autres constantes

- Type booléen

false

true

- Référence inexistante (indique qu'une variable de type non primitif ne référence rien) ; convient pour tous les types non primitifs

null

Valeurs par défaut

Si elles ne sont *pas initialisées*, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean	false
char	'\u0000'
Entier (byte short int long)	0 0L
Flottant (float double)	0.0F 0.0D
Référence d'objet	null

Opérateurs sur les types primitifs

- Les plus utilisés :

= + - * / % ++ -- += -= *= /=

== != > < >= <=

&& || ! (et, ou, négation)

- x++ : la valeur actuelle de x est utilisée dans l'expression et juste après x est incrémenté
- ++x : la valeur de x est incrémentée et ensuite la valeur de x est utilisée

x = 1; y = ++x * ++x; // Que vaut y ?

Exemples

```
int x = 9, y = 2;
```

```
int z = x/y;      // z = 4 (division entière)
```

```
z = x++ / y;      // z = 4 puis x = 10
```

```
z = --x / y;      // x = 9 puis z = 4
```

```
if (z == y)       // et pas un seul "=" !
```

```
    x += y;       // x = x + y
```

```
if (y != 0 && x/y > 8)    // raccourci, pas d'erreur si y = 0
```

```
    x = y = 3;          // y = 3, puis x = y, (à éviter !)
```

Résultat d'un calcul

- Tout calcul entre entiers donne un résultat de type int (ou long si au moins un des opérandes est de type long)
- Dépassement possibles !

`System.out.print(2000000000 + 2000000000);` affiche -294967296

- Les types numériques « flottants » (non entiers) respectent la norme IEEE 754
- Erreurs de calculs possibles :

`16.8 + 20.1` donne `36.9000000000000006`

- Pour les traitements de comptabilité, utiliser `java.math.BigDecimal`

Priorités

Postfixés	[] . (params) expr++ expr--
Unaires	++expr --expr +expr -expr ~ !
Création et cast	new (type)expr
Multiplicatifs	* / %
Additifs	+ -
Décalages bits	<< >> >>>
Relationnels	< > <= >= instanceof
Egalité	== !=
Bits, et	&
Bits, ou exclusif	^
Bits, ou inclusif	
Logique, et	&&
Logique, ou	
Conditionnel	? :
Affectation	= += -= *= /= %= &= ^= = <<= >>= >>>=

Les opérateurs d'égales priorités sont évalués de gauche à droite, sauf les opérateurs d'affectation, évalués de droite à gauche

Cast (transtypage) de types primitifs

- Java est un langage fortement typé ; dans certains cas, il est nécessaire de forcer le programme à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
- le cast (transtypage) : (type-forcé) expression

```
int x = 10, y = 3;
```

```
// on veut 3.3333... et pas 3.0
```

```
double z = (double)x / y; // cast de x suffit
```

Cast

- Un cast entre types primitifs peut occasionner une perte de données (eg int vers short)
- Un cast peut provoquer une simple perte de précision

la conversion d'un long vers un float peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur

Cast implicite

Une affectation entre types primitifs peut utiliser un cast implicite si elle ne peut pas provoquer de perte de valeur

Par exemple, un short peut être affecté à un int sans nécessiter de cast explicite :

```
short s = ...;
```

```
int i = s;
```

Cast explicite

- Si une affectation peut provoquer une perte de valeur, elle doivent comporter un cast explicite :

```
int i = ...;
```

```
short s = (short)i;
```

- L'oubli de ce cast explicite provoque une erreur à la compilation

Il faut écrire « float f = 1.2f; » et pas « float f = 1.2; »

Exemples

```
short s = 1000000; // erreur !
```

- Affectation statique (repérable par le compilateur) d'un int « petit » :

```
short s = 65; // pas d'erreur
```

- Pour une affectation non statique, le cast explicite est obligatoire :

```
int i = 60;
```

```
short b = (short)(i + 5);
```

- casts de types « flottants » vers les types entiers tronquent :

```
int i = (int)1.99; // i = 1, et pas 2
```

Exemples

```
byte b1 = (byte)20;
```

```
byte b2 = (byte)15;
```

```
byte b3 = b1 + b2;
```

provoque une erreur (b1 + b2 est un int)

La dernière ligne doit s'écrire

```
byte b3 = (byte)(b1 + b2);
```

Problèmes

perte de précision sans cast : risque d'erreur

```
long l1 = 123456789;
long l2 = 123456788;
float f1 = l1;
float f2 = l2;
System.out.println(f1); // 1.23456792E8
System.out.println(f2); // 1.23456784E8
System.out.println(l1 - l2); // 1
System.out.println(f1 - f2); // 8 !
```

cast explicite : risque d'erreur sans aucun avertissement ni message

```
int i = 130;
b = (byte)i; // b = -126 !
int c = (int)1e+30; // c = 2147483647 !
```

Un jeu !

```
int calcArea(int height, int width) {  
    return height * width;  
}
```

```
1. int a = calcArea(7, 12);  
2. short c = 7;  
3. calcArea(c,15);  
4. int d = calcArea(57);  
5. calcArea(2,3);  
6. long t = 42;  
7. int f = calcArea(t,17);  
8. int g = calcArea();  
9. calcArea();  
10. byte h = calcArea(4,20);  
11. int j = calcArea(2,3,5);
```



Les instructions



Blocs

- Les méthodes sont structurées en blocs par les structures de la programmation structurée
 - suites de blocs
 - alternatives
 - répétitions
- Un bloc est un ensemble d'instructions délimité par { et }
- Les blocs peuvent être emboîtés les uns dans les autres

Blocs & Portée des identificateurs

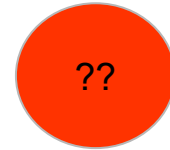
- Les blocs définissent la portée des identificateurs
- La portée d'un identificateur commence à l'endroit où il est déclaré et va jusqu'à la fin du bloc dans lequel il est défini, y compris dans les blocs emboîtés
- Les variables locales peuvent être déclarées n'importe où dans un bloc (pas seulement au début)
- On peut aussi déclarer la variable qui contrôle une boucle « for » dans l'instruction « for » (la portée est la boucle) :

```
for (int i = 0; i < 8; i++) {  
    s += valeur[i];  
}
```

Blocs & Portée des identificateurs

- Attention ! Java n'autorise pas la déclaration d'une variable dans un bloc avec le même nom qu'une variable d'un bloc emboîtant, ou qu'un paramètre de la méthode

```
int somme(int init) {  
    int i = init;  
    int j = 0;  
    for (int i=0; i<10; i++) {  
        j += i;  
    }  
    int init = 3;  
}
```



Conditionnelle

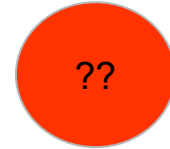
if (expressionBooléenne)
 bloc-instructions ou instruction
[else
 bloc-instructions ou instruction]

```
int x = y + 5;  
if (x % 2 == 0) {  
    type = 0;  
    x++;  
}  
else type = 1;
```

Conditionnelle : bonne pratique

Lorsque plusieurs if sont emboîtés les uns dans les autres, un bloc else se rattache au dernier bloc if qui n'a pas de else !

```
x = 3;  
y = 8;  
if (x == y)  
if (x > 10)  
x = x + 1;  
else  
x = x + 2;
```



Expression conditionnelle

expressionBooléenne ? expression1 : expression2

```
int y = (x % 2 == 0) ? x + 1 : x;
```

est équivalent à

```
int y;
```

```
if (x % 2 == 0)
```

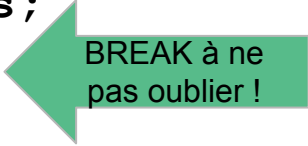
```
    y = x + 1
```

```
else
```

```
    y = x;
```

Switch

```
switch(expression) {  
    case val1: instructions;  
                break;  
    ...  
    case valn: instructions;  
                break;  
    default:   instructions;  
}
```



BREAK à ne
pas oublier !

expression est de type char, byte, short, ou int, ou énumération ou String

- S'il n'y a pas de clause default, rien n'est exécuté si expression ne correspond à aucun case

Exemple switch

```
char lettre;  
int nbVoyelles = 0, nbA = 0,  
nbT = 0, nbAutre = 0;  
. . .  
switch (lettre) {  
    case 'a' : nbA++;  
    case 'e' : // pas d'instruction !  
    case 'i' : nbVoyelles++;  
                break;  
    case 't' : nbT++;  
                break;  
    default :  nbAutre++;  
}
```

Répétition - boucles while

2 formes :

```
while (expressionBooléenne)  
    bloc-instructions ou instruction
```

```
do  
    bloc-instructions ou instruction  
while (expressionBooléenne)
```


Exemple while

```
public class Diviseur {  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        int j = 2;  
        while (i % j != 0) {  
            j++;  
        }  
        System.out.println("PPD de "  
            + i + " : " + j);  
    }  
}
```

Répétition - for

```
for(init; test; incrément){  
    instructions;  
}
```

est équivalent à

```
init;  
while (test) {  
    instructions;  
    incrément;  
}
```

```
int somme = 0;  
for (int i = 0; i <  
    tab.length; i++) {  
    somme += tab[i];  
}  
System.out.println(somme);
```

Répétition - for each

- Une syntaxe pour simplifier le parcours d'un tableau/collection
- Attention, on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)

```
String[] noms = new String[50];  
...  
// Lire « pour chaque nom dans noms »  
// « : » se lit « dans »  
for (String nom : noms) {  
    System.out.println(nom);  
}
```

Répétition - break/continue

- break sort de la boucle et continue après la boucle
- continue passe à l'itération suivante

- break et continue peuvent être suivis d'un nom d'étiquette qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle) [bof !]

Répétitions - exemple

```
int somme = 0;
for (int i = 0; i < tab.length; i++) {
    if (tab[i] == 0) break;
    if (tab[i] < 0) continue;
    somme += tab[i];
}
System.out.println(somme);
```

Qu'affiche ce code avec le tableau :
1 ; -2 ; 5 ; -1 ; 0 ; 8 ; -3 ; 10 ?

Un petit jeu

> java Melange

a-b c-d

```
class Melange {  
    public static void main(String [] args) {
```

```
        if (x == 1) {  
            System.out.print("d");  
            x = x - 1;  
        }
```

```
        if (x == 2) {  
            System.out.print("b c");  
        }
```

```
        if (x > 2) {  
            System.out.print("a");  
        }
```

```
        int x = 3;
```

```
        while (x > 0) {
```

```
            x = x - 1;  
            System.out.print("-");
```

Jeu 2 : un compilateur tu seras !

```
class Exercice1 {  
    public static void main(String [] args) {  
        int x = 1;  
        while ( x < 10 ) {  
            if ( x > 3) {  
                System.out.println("big x");  
            }  
        }  
    }  
}
```

Jeu 2 : un compilateur tu seras !

```
public static void main(String [] args) {  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3) {  
            System.out.println("small x");  
        }  
    }  
}
```


Jeu 2 : un compilateur tu seras !

```
class Exercice3{  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3) {  
            System.out.println("small x");  
        }  
    }  
}
```

Jeu 3 : le bloc manquant !

```
class Test {  
    public static void main(String [] args) {  
        int x = 0;  
        int y = 0;  
        while ( x < 5 ) {  
            Bloc manquant  
            System.out.print(x + "" + y + " ");  
            x = x + 1;  
        }  
    }  
}
```

Jeu 3 : qui fait quoi ?

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;  
if( y > 4 ) {  
  y = y - 1;  
}
```

```
x = x + 1;  
y = y + x;
```

```
if ( y < 5 ) {  
  x = x + 1;  
  if ( y < 3 ) {  
    x = x - 1;  
  }  
}  
y = y + 2;
```

22 46

11 34 59

02 14 26 38

02 14 36 48

00 11 21 32 42

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47