http://www.labo-sun.com
labo-sun@supinfo.com

La syntaxe Java

BASES & NOMENCLATURES

Auteur : Olivier Corgeron
Version 2.0 - 19 octobre 2004
Nombre de pages : 21

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Bases

1.1. Les identificateurs

L'identificateur (traduit de "handle") est le nom que l'on choisit pour appeler un élément comme une variable, une méthode, une classe ou encore une interface. Ces identificateurs doivent commencer par une lettre Unicode ou encore un underscore "_".

1.2. Les mots réservés

• abstract • boolean • break • byte • case • catch • char • class • continue • const • default • do • double • else • enum (nouveauté Java 5.0) • extends • false • final • finally • float • for • goto • if • implements • import • instanceof	• int • interface • long • native • new • null • package • private • protected • public • return • short • static • super • switch • synchronized • this • throw • throws • transient • true • try • void • volatile • while

1.3.Les commentaires

1.3.1. Les commentaires explicatifs

Les commentaires sont gérés de deux manières en Java :

Soit avec le style du C :

```
/*  ceci
    est un
    commentaire
*/
```

Soit avec le style du C++

```
// Ceci est un commentaire sur une seule ligne
```

Les commentaires sur une seule ligne servent surtout pour de courtes remarques à l'intérieur des méthodes, celles sur plusieurs lignes sont plus appropriées pour de larges remarques.

1.3.2. Les commentaires Javadoc

La zone de commentaire commencent par **/**** informe un commentaire de documentation. Ce commentaire est dégagé par un générateur automatique de documentation. Ce commentaire se terminera toujours par ***/** :

```
/**
 * Documentation sur la classe
 * @version 12.33, 2004-10-03
 */
```

Les espaces avant les lignes sont ignorés.

Les lignes qui débutent par **@** sont reconnues comme des mots clef spécifiques pour le générateur de documentation (Javadoc).

1.3.3. Les tags

Voici une liste non exhaustive de tags que vous allez pouvoir utiliser pour documenter votre code.

Tag	Rôle	Elément concerné	Version du JDK
@author	permet de préciser l'auteur de l'élément	classe et interface	1.0
@deprecated	permet de préciser qu'un élément est déprécié	package, classe, interface, méthode et champ	1.1
{@docRoot}	représente le chemin relatif du répertoire principal de génération de la documentation		1.3

@exception	permet de préciser une exception qui peut être levée par l'élément	méthode	1.0
{@link}	permet d'insérer un lien vers un élément de la documentation dans n'importe quel texte	package, classe, interface, méthode, champ	1.2
@param	permet de préciser un paramètre de l'élément	constructeur et méthode	1.0
@see	permet de préciser un élément en relation avec l'élément documenté	package, classe, interface, champ	1.0
@since	permet de préciser depuis quelle version l'élément a été ajouté	package, classe, interface, méthode et champ	1.1
@throws	identique à @exception	méthode	1.2
@version	permet de préciser le numéro de version de l'élément	classe et interface	1.0
@return	permet de préciser la valeur de retour d'un élément	méthode	1.0

2. Types primitifs et définition de variables

Toutes les variables et éléments possèdent des types Java reconnus lors de la compilation, comme dans tous les langages statiques.

Les types de données Java sont regroupés en deux catégories bien distinctes :

- *Les types de base* : valeurs simples à fonctionnalités intégrées dans le système (constante, expression, numérique).
  La zone mémoire allouée contient la valeur associée à l'identificateur[1] (comme un caractère pour les « **char** »).

- *Les types de références* : les données passées grâce à leur référence. Ce sont les types de classes. Ils englobent les tableaux et les objets.
  La zone mémoire allouée à l'identificateur contient un hexadécimal représentant l'adresse mémoire où sont situés les champs de l'objet.
  Celle-ci peut être **null**.

2.1. Les types de base (primitifs)

- **char** (16 bits Unicode)
- **short** (16 bits entier signé)
- **int** (32 bits entier signé)
- **long** (64 bits entier signé)
- **float** (32 bits IEEE (spécification internationale) 754)
- **double** (64 bits IEEE (spécification internationale) 754)
- **byte** (8 bits entier signé)
- **boolean** (1 bit, **true**/**false**)

Les opérations à virgule flottante sont standardisées afin de respecter la norme internationale IEEE 754 qui définit que tous les calculs flottants sont identiques sur toutes les plateformes de Java.

2.1.1. Déclaration et initialisation

Les variables sont déclarées comme en C :

[1] Un identificateur désigne une classe, une méthode, une variable

```
int toto;
double t1, t2;
boolean faux;
```

Elles peuvent aussi être initialisées lors de la déclaration :

```
int toto = 10;
double t1=1.25, t2 = 1.26;
boolean faux = false;
```

Les variables déclarées sous la forme de variable d'instance à l'intérieur d'une classe non initialisées ont une valeur par défaut.

Par défaut les types numériques sont à 0, les caractères au caractère **nul** '\0', et les variables booléennes à **false**.

Remarque : Avant qu'une variable puisse être utilisée, elle doit être initialisée !

2.1.1.1. Les entiers

Quand un type numérique est utilisé dans le cadre d'une affectation, il peut être étendu à un type plus important.

Par contre une valeur numérique ne peut jamais être affectée à un type inférieur sans conversion.

```
short x = 2;
int i = x;
```

```
int i = 2;
short x = (short) i;
```

Dans ce second cas, la conversion explicite permet d'éviter une erreur à la compilation.

2.1.1.2. Les nombres flottants

Ils sont toujours de type **double** sauf s'ils sont suivis de la lettre 'f' ou 'F' qui indique le type **float**.

```
double x = 5.23;
float y = 5.125F;
```

2.1.1.3. Les caractères

Spécification d'une valeur de type caractère encadrée par des simples quottes ('a'), le caractère d'échappement étant \, ou encore un numéro de caractère Unicode en hexadécimal.

```
char x = 'x';
char y = '\'';
char z = '\cafe';
```

2.2. Les types référence

Java permet de créer des nouveaux types de données : ainsi en créant une nouvelle classe, implicitement nous créons un nouveau type.

Un élément d'un type particulier peut être affecté à une variable du même type ou passé en argument à une méthode acceptant une valeur de ce type.

Les types de références ont le même nom que les classes qu'ils représentent.

A l'instar des classes, les types enfants sont des types dérivés de la classe parente. **Un objet de type enfant peut être utilisé à la place d'un objet de type parent** étant donné qu'il reprend toutes les spécificités de la classe parent.

Donc un objet de type enfant pourra toujours être utilisé là où un objet de type parent est utilisé. C'est ***la dérivation polymorphique*** (cf. Essentiels « Les Classes – Concept & Héritage »).

Contrairement au passage par valeur des types primitifs (où les valeurs sont toujours dupliquées), les variables de type "référence" contiennent une référence sur un objet de son type. C'est l'équivalent des pointeurs en C/C++ mais en Java, les types sont fixés et la valeur de référence ne peut être consultée directement.

Convention :

Les types de données de référence peuvent être soit des classes, soit des tableaux.

La nomenclature Java veut que les noms des classes et des tableaux commencent par une **majuscule** (ainsi que la première lettre de chaque mot composant un identifiant. Ex : class ClasseLaMaison).

En revanche, les noms de variables et de méthodes doivent commencer par une lettre **minuscule**.

Ex : ***nomDeMethode(…)***, **maVariable**.

En ce qui concerne les noms des packages, ils doivent être entièrement en minuscule.

Ex : **com.company.mypackage**

2.3.Les wrappers

Nous avons vu qu'il existait des types primitifs en java. Cependant il est parfois utile « d'encapsuler » ses types primitifs afin par exemple de les passer par référence.

Java a donc prévu des classes enveloppes à ces primitives. Voici un tableau récapitulatif des *wrappers* disponibles et de leur relation avec les types primitifs :

type	*classes enveloppes*
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Ces *wrappers* ont également d'autres utilités tel la conversion. Si vous souhaiter « parser » (analyser) une chaîne de caractères vers un entier par exemple il vous faudra utiliser le *wrappers* : **Integer**.

```
int entier = Integer.parse("10");
```

2.4.Les méthodes

Tous comme en C/C++, les méthodes Java comprennent un type de retour, un nom, des parenthèses avec d'éventuels paramètres, et le code entre accolades.

Cette méthode peut être de plus précédée d'un mot-clé définissant sa visibilité (cf. Essentiels « Les Classes – Concept & Héritage »)

```
void maMethode(int i) {
	// code
}
```

```
public void maMethode2(int i) {
	// code
}
```

2.5.La méthode *main(…)*

La méthode ***main(…)*** est le point de départ du programme. C'est celle qui sera exécutée par la machine virtuelle Java lorsque vous demanderez l'exécution du programme.

Elle s'écrit **toujours** de la manière suivante (seul le nom du paramètre peut être modifié) :

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

3. Les instructions

3.1. Instructions simples

Les instructions simples sont toujours terminées par un point virgule.

Les instructions les plus communes sont les affectations et les appels de méthodes, elles se trouvent dans des blocs de code délimités par des accolades.

```
{
	int longueur = 10;		// déclaration et affectation
	float poids;			// déclaration de variable
}
```

Remarque : Les déclarations de variables ont une portée limitée au bloc de code et ne peuvent donc pas être vues à l'extérieur des accolades où elles ont été déclarées

3.2. Expression conditionnelle

3.2.1. L'expression if / else

Sémantique de l'expression : si prédicat = vrai alors exécution de l'instruction 1 sinon exécution de l'instruction 2.

```
if (condition) {
	instruction1;
} else if (condition2) {
	instruction2;
} else {
	instruction3;
}
```

L'expression est de type **boolean**. La condition est vérifiée (vrai) ou non (fausse).

3.2.2. L'expression *switch*

L'expression est comparée à l'étiquette, l'exécution s'effectue jusqu'à la fin du **switch** sauf bien sur si une instruction **break** est rencontrée.

Tout comme en C, l'instruction **break** provoque la sortie du **switch**. Si aucune étiquette ne correspond à l'expression, l'étiquette par défaut est sélectionnée.

```
int jours = 7;
switch (jours) {
	case 1:
		System.out.println("lundi");
		break;
	case 2:
		System.out.println("mardi");
		break;
	case 3:
		System.out.println("mercredi");
		break;
	case 4:
		System.out.println("jeudi");
		break;
	case 5:
		System.out.println("vendredi");
		break;
	case 6:
		System.out.println("samedi");
		break;
	case 7:
		System.out.println("dimanche");
		break;
	default:
		System.out.println("Jour erroné!");
}
```

Dans cet exemple la variable "jours" est de type **int**.
Si la variable vaut "1", lundi est affiché.
Si la variable vaut "4", Jeudi est affiché.
Pour toute autre valeur de la variable "jours", "jour erroné" est affiché.

Remarque : la variable utilisée avec l'instruction **switch** doit impérativement être du type **int** ou compatible (plus petit).

3.3.Itérations

3.3.1. L'itérateur while

La sémantique du **while** : Tant que la condition est réalisée, exécuter l'instruction.
La condition est de type **boolean** et l'itération exécute l'instruction tant que la condition donne la valeur **true**. Si la condition est fausse, l'instruction ne sera jamais exécutée.

Forme générale :
while (condition) {instructions};

```
int i = 10, somme = 0, j = 0;
while (j <= i) {
	somme += j++; // j++ est une opération de post-incrémentation
}
System.out.println("boucle while: " + somme);
```

3.3.2. L'itérateur do / while

La sémantique du **do** / **while** : exécute l'instruction tant que la condition est vérifiée.
La condition est évaluée à la fin. Le bloc d'instruction est exécuté une fois au minimum.
La boucle cesse dès que la condition est fausse.

Forme générale :
do {instructions} while (condition);

```
int i = 10, somme = 0, j = 0;
do {
	somme = somme + j;
j = j + 1;
} while (j <= i);
System.out.println("boucle do while " + somme) ;
```

3.3.3. L'itérateur for

L'instruction **for** ressemble beaucoup à celle du C : on peut ajouter de nouvelles variables dans l'expression d'initialisation, ces variables auront par contre une durée de vie limitée à la portée de l'instruction.

Forme générale :

for (initialisation; condition; incrémentation) {instructions;}

```
for (int i = 0; i < 10; i++) {
	System.out.println(i);
}
```

Il est tout à fait possible de ne spécifier aucun des arguments *initialisation*, *condition* et *incrémentation*, dans quel cas l'utilisation du mot clé for correspond à une boucle infinie. Vous pouvez également les omettre individuellement.

Il n'est pas nécessaire que l'instruction incrémentation se reporte impérativement à une opération sur un compteur : vous pouvez très bien mettre une opération que vous souhaitez voir se répéter à la fin de chaque boucle.

Remarque : Java ne gère pas l'opérateur « , » (virgule) qui permet de regrouper plusieurs expressions en une seule.

4. Les opérateurs

L'accès ou la référence à des objets, des variables ou des classes, est effectuée grâce aux opérateurs.

Des propriétés de **priorité** et d'**associativité** sont affectées aux opérateurs. Des règles de priorités existent en Java pour définir quels sont les éléments qui peuvent agir avant les autres.

Les opérateurs sont classés en six catégories bien distinctes : affectation, arithmétique, logique, comparaison, au niveau binaire et ternaire.

4.1. Points communs et différences avec le C

Java gère la quasi-totalité des opérateurs pris en compte dans le langage C.

Les priorités sont en Java les mêmes que celles du C. Toutefois certains opérateurs du langage C n'existent pas en Java comme la virgule pour combiner plusieurs expressions.

Les opérateurs "*&*", "*", et "*sizeof*" ne sont pas gérés, puisque Java ne permet pas de modifier directement les pointeurs.

En compensation, Java ajoute quelques nouveaux opérateurs comme le "+" pour effectuer des concaténations de valeurs de type « String ».

Puisque tous les types numériques sont signés, l'opérateur ">>" effectue un décalage de bits à droite avec une extension pour le signe et l'opérateur ">>>" effectue un décalage de bits à droite sans se préoccuper du signe.

4.2. Les différentes catégories d'opérateurs

4.2.1. Les opérateurs d'affectation

C'est le stockage de la valeur de droite dans celle de gauche avec le signe "=".
Cet opérateur permet de donner une valeur à une variable.

```
int nombre; // déclaration
nombre = 2; // initialisation
```

Opérateur	Définition	Priorité	Associativité
=	Affecte la valeur de droite à la variable de gauche	15	Droite
+=	Ajoute la valeur de droite à la variable de gauche	15	Droite
-=	Soustrait la valeur de droite à la variable de gauche Puis affecte la nouvelle valeur à la variable de droite	15	Droite
*=	Multiplie la valeur de droite avec la variable de gauche. Puis affecte la nouvelle valeur à la variable de droite	15	Droite
/=	Divise la valeur de droite avec la variable de gauche. Puis	15	Droite

	affecte la nouvelle valeur à la variable de droite		

4.2.2. Les opérateurs logiques

Ils permettent de rassembler des expressions de type booléen en indiquant au programme comment comprendre une condition spécifique :

Opérateur	Définition	Priorité	Associativité
!	"Not" (Change "true" en "false" ou inversement	2	Droite
&	"And" renvoie "true" les deux opérandes valent "true"	9	Gauche
^	"XOR" renvoie "true" si une des deux opérandes vaut "true"	10	Gauche
|	"OR" renvoie "true" si au moins un des opérandes vaut "true"	11	Gauche
&&	"AND" conditionnel. N'évalue le second opérande que si le premier est "true"	12	Gauche
||	"OR" conditionnel. N'évalue le second opérande que si le premier n'est pas "true"	13	Gauche

4.2.3. Les opérateurs de comparaison

Leur fonction première est de comparer des expressions à d'autres parties du code.

Opérateur	Définition	Priorité	Associativité
<	Inférieur	7	Gauche
>	Supérieur	7	Gauche
<=	Inférieur ou égale	7	Gauche
>=	Supérieur ou égale	7	Gauche
==	Egale	8	Gauche
!=	Différent	8	Gauche

4.2.4. Les opérateurs arithmétiques

Java peut exécuter des fonctions mathématiques avec les opérateurs utilisés pour effecteur les calculs sur les entiers et sur les nombres à virgule flottante. Les signes mathématiques "+", "-", "*" et "/" sont utilisés.

Opérateur	Description	Type d'opérande	Priorité	Associativité
++/--	Incrémentation et décrémentation	Arithmétique	1	Droite
+/-	Plus et moins unitaires	Booléen	2	Droite
*	Multiplication	Arithmétique	4	Gauche
/	Division	Arithmétique	4	Gauche
%	Modulo	Arithmétique	4	Gauche
+/-	Addition et soustractions	Arithmétique	5	Gauche

4.2.5. Les opérateurs au niveau binaires

Ces opérateurs agissent sur les binaires. Ils préservent le signe du nombre initial.

Opérateur	Description	Priorité	Associativité
~	Not Inverse chaque bit 0 devinent 1 et inversement	2	Droite
<<	Décalage de bits à gauche signé	6	Gauche
>>	Décalage de bits à droite signé	6	Gauche
>>>	Décalage de bits à droite par ajout de zéros	6	Gauche
&	And au niveau des bits	9	Gauche
|	Or au niveau des bits	10	Gauche
^	Xor au niveau des bits	11	Gauche
<<=	Décalage de bits à gauche avec affectation	15	Gauche
>>=	Décalage de bits à droite avec affectation	15	Gauche
>>>=	Décalage de bits à droite par ajout de zéros avec affectation	15	Gauche

4.2.6. L'opérateur conditionnel ternaire

Il existe un opérateur conditionnel ternaire en Java :

Forme générale :
(condition ? condition_validée : condition_invalidé)

```
int j = (i > 2 ? i : 0); // j = i si i supérieur à 2, sinon j = 0
```

5. Les tableaux

Les tableaux sont des structures de données qui contiennent des valeurs du même type.

Les tableaux sont marqués à l'aide de crochets [ ].

5.1. Déclaration du tableau

Pour déclarer un tableau en Java on peut placer les crochets avant ou après l'identificateur.

```
int collection[];
int[] ensemble;
```

Quand un tableau est déclaré, sa taille n'est **pas spécifiée** puisque la déclaration de ce tableau n'engendre pas l'allocation de mémoire.

La taille ne sera définie que lorsque le tableau sera initialisé.

5.2. Initialisation du tableau

L'opérateur **new** est utilisé pour allouer de la mémoire et initialiser le tableau.

5.2.1. Tableau à une dimension

Dans un tableau nouvellement créé, tous les éléments sont **null** ou à 0.
La **première position** du tableau commence à l'index 0 tout comme en C. Dans notre exemple, les éléments sont numérotés **de 0 à 9.**

```
// tableau de 10 éléments de type int initialisé à 0
int collection[] = new int[10];
```

Quand on initialise un tableau, la mémoire est automatiquement affectée au tableau pour y mettre les valeurs initialisées ce qui ne nécessite pas l'utilisation de l'opérateur **new**.

```
char[] genre= {'h', 'f'};
```

5.2.2. Tableau multidimensionnels

Vous pouvez définir des tableaux avec autant de dimensions que vous souhaitez.

Pour les tableaux bidimensionnels, le premier élément détermine le nombre de colonnes et le second le nombre de lignes : [colonnes][lignes].

```
float[][] dimension = new float[10][10];
// tableau bidimensionnel de 10*10 éléments de type float
int[][] coordonnées = {{1, 3}, {7, 5}};
```

Remarque : Lorsque vous déclarez un tableau multidimensionnel, vous pouvez omettre la taille des dimensions sauf pour la première qui est obligatoire.

5.3.Accéder à un tableau

Pour atteindre un élément d'un tableau, il suffit d'indexer la variable du tableau. Pour cela il faut faire suivre le nom du tableau de l'indice entre crochets.

```
char premierElement = genre[0];         // 'h'
char deuxièmeElement = genre[1];        // 'f'
```

Si vous spécifiez un indice incorrect, une erreur (Exception) sera générée. La gestion des erreurs sera traitée dans un autre cours (Exceptions).

5.3.1. Longueur d'un tableau

Il est possible de demander la taille d'un tableau en utilisant la variable **length**.

```
int[] tableau = {1, 10, 12, 9};
System.out.println("Longueur du tableau " + tableau.length);
```