Annexes Programmation en langage Java

tv <tvaira@free.fr>

Table des matières

1 Annexe n°1 : Les types primitifs

En Java, on appelle *type primitif* tout ce qui n'est pas objet et qui a une représentation immédiate en mémoire. Les tableaux ne sont pas des types primitifs, mais ne peuvent pas non plus être considérés complètement comme des objets. Nous en parlerons néanmoins dans cette section.

Du fait du vœu de portabilité de Java, les types primitifs ont une taille indépendante de la plateforme d'exécution du code. Il existe quatre catégories de type primitifs : les types entiers, les types réels, le type booléen et le type caractère.

1.1 Types entiers et réels

Le tableau ci-dessous récapitule les types entiers, leur taille et leurs valeurs extrêmes.

Type	Taille (en bits)	Valeur minimale	Valeur maximale
`byte`	8	-128	127
`short`	16	-32768	32767
`int`	32	-2147483648	2147483647
`byte`	64	-9223372036854775808	9223372036854775807

Contrairement au langage C, tous les types entiers sont signés (les nombres négatifs sont codés en complément à 2). En revanche, comme en langage C, on peut exprimer un entier (par exemple 15) en décimal (`15`), en octal (`017`) ou en hexadécimal (`0x0F`).

Le tableau suivant donne les mêmes informations que le précédent en ce qui concerne les types flottants (codés conformément à la norme IEEE 754).

Type	Taille	Chiffres significatifs	Valeur minimale	Valeur maximale
`float`	32	7	1.4239846E-45	3.40282347E38
`double`	64	15	4.9406564584124654E-234	1.797693134862316E308

Deux notations sont utilisables : décimale (`-3.25`) ou exponentielle (`-3.232323E-12`, le `E` peut être aussi écrit `e`). Une valeur immédiate flottante est implicitement de type `double`, ce qui peut poser problème si on essaye de l'affecter à une variable de type `float`. En langage C, ce genre de problème se règle par un *forçage de type* (aussi appelé *transtypage* ou *cast*). Il en est de même en Java, le cast ayant la même syntaxe, consistant à faire précéder la valeur du type compatible entre parenthèses. Dans le cas du cast en `float`, il existe une abbréviation qui consiste à rajouter uniquement la lette `f` derrière la valeur flottante (`-3,2E-4f`).

1.2 Les types booléen et caractère

Le type booléen est désigné par le mot-clé `boolean` et représente une valeur binaire. Il existe deux constantes qui peuvent être valeurs d'une variable de type booléen : `true` et `false`.

Le type caractère est désigné par le mot-clé `char`. Sa taille est de 16 bits, elle permet de stocker une valeur *Unicode*.[1] On note une valeur de ce type entre apostrophes : `char a = 'a';`.

Comme le type caractère sert à représenter un caractère *Unicode* mais qu'il est probable que l'éditeur utilisé ne supporte pas cette norme, il est néanmoins toujours possible de faire référence à un tel caractère (forcément supporté par la machine virtuelle) en écrivant `\uxxxx` où `xxxx` est le code sur 16 bits du caractère en question. Comme les codes attribués aux caractères se suivent, on peut souhaiter effectuer des opérations arithmétiques sur des caractères pour passer de l'un à l'autre ou pour changer la casse. Ainsi, `(char)('a'+1)` représente le caractère `b`.

1.3 Le type tableau

Le type *tableau* est désigné par le mot-clé du type (primitif ou objet) suivi de `[]`. Par exemple, `byte[]` représente un tableau d'octets.

1. *Unicode* (`http://www.unicode.org`) est une norme qui définit un jeu de caractères universel (qui comprend tous les caractères disponibles dans toutes les langues écrites) et plusieurs types de codages de longueurs fixes (16 bits ou 32 bits par caractère, respectivement appelés UTF-16 et UTF-32) ou variables (de 8 à 24 bits par caractère, appelé UTF-8). Java est conçu de base pour manipuler les caractères et les chaînes de caractères UTF-16 (il est même possible d'utiliser des caractères *Unicode* dans les codes sources.

1.4 Le mot-clé `void`

`void`, contrairement à son interprètation en C, n'est pas considéré comme un type. En Java, toutes les variables doivent avoir un type bien défini et il n'existe pas de pointeurs ni d'arithmétique de pointeurs. `void` en Java est utilisé uniquement en valeur de retour de méthode pour indiquer que la méthode ne renvoie ni référence d'objet ni valeur de type primitif.

1.5 Déclaration de variables, affectation

La déclaration d'un entier, d'un flottant, d'un booléen ou d'un caractère s'effectue de la même manière qu'en C, à ceci près qu'il est possible de déclarer une variable n'importe où à l'intérieur d'un bloc d'instructions (délimité par {}). La portée d'une variable est le bloc d'instructions dans lequel elle est déclarée. Il est possible d'initialiser une variable lors de sa déclaration. Ci-dessous est présenté un exemple de déclaration, initialisation et affectation de variables.

```
public static void main(String[] args)
{
    int i = 0;
    float c;
    c = 3;              // une instruction
    // notez au passage la syntaxe d'un commentaire de fin de ligne ...
    /* ceci est un autre commentaire qui n'invalide pas le reste de la ligne... */
    boolean b = (c>i);
    System.out.println(b); // affiche la valeur de b sur la console et passe a la ligne
}
```

La déclaration d'un tableau en Java est en revanche différente de celle en C, elle fait appel à la notion de *constructeur* (à travers le mot-clé `new`) que nous reverrons bientôt. Ci-dessous est présenté un exemple de déclaration, initialisation et manipulation de tableau.

```
public static void main(String[] args)
{
   // args est un tableau de chaînes de caractères représentant les arguments
   // passés à l'application sur la ligne de commande
   // Contrairement à C, args[0] représente le premier "vrai" argument.

   // Déclaration d'un tableau statique de 5 valeurs de type short
   // (l'espace mémoire réservé n'est pas extensible)
   short[] s = new short[5];

   // Les indices, comme en C, varient de 0 à taille-1
   s[0] = 3;
   s[1] = 4;

   // le nom du tableau postfixé de .length désigne la capacité du tableau
   s[2] = s.length;

   // la précédence des opérateurs est identique à celle de C
   s[3] = s[2]*s[0]-s[1];

   // On ne peut pas demander de manière simple l'affichage d'un tableau,
   // on obtiendrait non pas la liste, mais une valeur unique n'ayant rien à voir.
   // L'argument de println est une chaîne de caractères.
   // Java se charge de convertir n'importe quel type en chaîne de caractères.
   // On doit cependant concatener plutôt qu'écrire s[0]+s[1] ...
   // car sinon on afficherait non pas la liste mais la somme.
   // s[0]+s[1] est converti automatiquement en int,
   // s[0]+"" ("" est la chaîne vide) est converti en chaîne de caractères (String)

   // Affiche (sur la console) les éléments du tableau sur une ligne et passe à la ligne
   System.out.println(s[0]+""+s[1]+""+s[2]+""+s[3]+""+s[4]);
}
```

1.6 Opérateurs arithmétiques, relationnels et logiques

Cette sous-section décrit rapidement l'ensemble des opérateurs utilisables en Java.

1.6.1 Les opérateurs arithmétiques et leurs priorités

On trouve en Java les mêmes opérateurs arithmétiques qu'en C (à l'exception de l'opérateur puissance) :

- l'addition, notée `+`
- la soustraction, notée `-`
- l'opposé, (opérateur unaire) noté `-`
- l'identité (opérateur unaire), notée `+`
- la multiplication, notée `*`
- la division (entière si les deux opérandes sont de type entier), notée `/`
- le reste de la division entière, noté `%`

La priorité s'opère comme suit :

1. les opérateurs unaires `-` et `+` sont les plus prioritaires,
2. les opérateurs `*` et `/` ont ensuite la priorité la plus élévée,
3. viennent ensuite les opérateurs binaires `+` et `-`.
4. l'associativité s'effectue de gauche à droite pour des opérateurs de même priorité.

1.6.2 Les opérateurs relationnels et logiques et leurs priorités

Les opérateurs relationnels sont au nombre de six :

- *inférieur à*, noté `<`
- *supérieur à*, noté `>`
- *égal à*, noté `==`
- *inférieur ou égal à*, noté `<=`
- *supérieur ou égal à*, noté `>=`
- *différent de*, noté `!=`

Le langage Java compte également six opérateurs logiques :

- la négation logique, notée `!`
- le ET logique, noté `&`
- le OU logique, noté `|`
- le OU exclusif logique, noté `^`
- la conjonction, notée `&&`
- la disjonction, notée `||`

L'opérateur `&` (respectivement `|`) produit le même résultat que `&&` (respectivement `||`), cependant il est préférable d'utiliser le second dans une condition booléenne car il a la propriété de court-circuiter le calcul si la première opérande permet de déterminer directement le résultat.

La négation logique est l'opérateur le plus prioritaire. Les priorités des autres opérateurs sont identiques et l'associativité s'effectue de gauche à droite.

2 Annexe n°2 : Les instructions de contrôle

Avant de présenter la syntaxe des différentes instructions de contrôle, rappelons d'abord qu'une instruction élémentaire (affectation, déclaration, appel de méthode, ...) en Java doit être terminée par un point-virgule. Dans la suite, lorsque nous utiliserons le mot *instruction*, nous désignerons :

- soit une instruction élémentaire,
- soit une instruction de contrôle,
- soit un bloc d'instructions (des deux types précédents), délimité par {}.

Dans ce qui suit, les crochets ([]) sont utilisés pour dénoter le caractère optionnel d'une construction.

2.1 `if/else`

```
if condition
  instruction
[else instruction]
```

La condition doit être exprimée entre parenthèses, exemple :

```
if (fini==true) System.out.println("stop");
else System.out.print(".");
```

2.2 `switch`

La construction `switch` permet d'exprimer une liste de décisions. Sa syntaxe est la suivante :

```
switch (expression)
{
  case valeur1 :instruction
  [case valeur2 :instruction
   ...
   case valeurn : instruction]
  [default : instruction]
}
```

Exemple :

```
if ((a>=0)&&(a<=15))
{
 switch (a)
 {
   case 10 : System.out.println('A'); break;
   case 11 : System.out.println('B'); break;
   case 12 : System.out.println('C'); break;
   case 13 : System.out.println('D'); break;
   case 14 : System.out.println('E'); break;
   case 15 : System.out.println('F'); break;
   default : system.out.println(a);
 }
}
```

Il faut remarquer que les `case` ne sont que des étiquettes de branchement, aucun test n'y est effectué. L'expression est testée au début du `switch` et, selon sa valeur, le programme branche à l'étiquette correspondante et le flot d'instruction suit son cours. Ceci veut dire que s'il n'existe pas de rupture de flot de contrôle (voir `break` plus loin dans cette sous-section) les instructions à l'étiquette `case` adéquate et toutes les instructions aux étiquettes suivantes sont exécutées.

L'étiquette `default` permet de spécifier un comportement par défaut (lorsqu'aucune étiquette ne correspond à la valeur de l'expression) ou un post-traitement.

2.3 for

```
for (instruction1; expression; instruction2)
  instruction
```

L'instruction `instruction1` est une instruction élémentaire exécutée une et une seule fois avant le premier tour de boucle. C'est en général la déclaration et l'initialisation du compteur de boucle.

L'expression booléenne `expression` est évaluée avant chaque tour de boucle et dénote une condition. Si cette condition n'est pas vérifiée, on ne rentre plus dans le corps de la boucle.

L'instruction `instruction2` est une instruction élémentaire exécutée à chaque fin de tour de boucle. C'est en général l'opération d'incrémentation ou de décrémentation du compteur de boucle.

Exemple : `for(int i=0;i<7;i++) {System.out.println(i);}`

2.4 while

```
while (expression)
 instruction
```

2.5 do while

```
do
{
}
while (expression)
```

La différence entre cette instruction de contrôle et la précédente tient dans le fait que l'expression dénotant la condition d'arrêt est évaluée à la fin de chaque tour de boucle plutôt qu'au début (ce qui signifie que l'instruction constituant le corps de la boucle est exécutée au moins une fois).

2.6 break et continue

Ces deux instructions sont des instructions de rupture de flot de contrôle. Alors que `break` peut être utilisée dans n'importe quelle instruction de contrôle ou bloc d'instructions, `continue` ne peut être utilisée que dans les itérations (c'est à dire `for` et les variantes de `while`).

L'instruction `break` permet de sortir immédiatement d'un bloc d'instruction ou d'une boucle. Par exemple, si une telle instruction termine chaque `case` d'un `switch`, alors uniquement le bloc d'instructions associé au `case` correspondant à une valeur de l'expression sera exécuté (et le `default` sera dans ce cas vraiment un traitement par défaut).

L'instruction `continue` permet quant à elle de court-circuiter le reste du corps d'une boucle pour redémarrer de suite un nouveau tour (en effectuant tout de même les traitements intervenant à chaque fin de tour).

3 Annexe n°3 : Les chaînes de caractères

Cette section débute avec quelques éléments théoriques sur l'utilisation de la classe `String` et se termine avec des exercices d'application.

La classe `String` permet la manipulation des chaînes de caractères. Il existe deux écritures pour obtenir une référence sur un nouvel objet de type `String` sans faire explicitement appel à un des constructeurs :

1. String s1 = "La surface d'un cercle de rayon r est égale à π * r^2";
2. String s2 = s1 + ".";

Dans le second cas, on peut remarquer au passage que l'opération `+` est la concaténation de deux chaînes. l'affectation `+=` est autorisée également, mais par contre pas question de `*`, / et autres opérations arithmétiques.

Un élément important à retenir est que lorsque l'on veut concaténer à une chaîne de caractères une valeur qui n'est pas de type chaîne, cette dernière est d'abord convertie en chaîne de caractères comme suit :

- s'il s'agit d'un type primitif, on construit une chaîne représentant sa valeur (par exemple la valeur entière `2` devient `"2"`, la valeur booléenne `true` devient `"true"`),
- s'il s'agit d'un type objet, on invoque sa méthode `toString()`,
- s'il s'agit d'un tableau, le résultat de la conversion est équivalent à un appel à `toString` sur un objet (ce qui veut dire que si l'on veut afficher les éléments du tableau, il faut construire soit même la chaîne correspondante).

La particularité de la classe `String` est qu'elle n'autorise pas la modification du contenu des chaînes de caractères qu'elle encapsule ; ce qui revient à dire que des objets de type `String` représentent des chaînes de caractères **en lecture seule**. Les méthodes proposées par cette classe sont alors de deux nature :

- celles qui permettent d'obtenir des informations sur la chaîne encapsulée (comme `length`, ...),
- celles qui permettent de dériver une nouvelle chaîne de caractères par transformation (comme `substring`) et qui créent donc un nouvel objet et renvoient sa référence.

A noter que la comparaison de chaînes ne s'effectue pas avec `==` (ceci revient à comparer les références) mais via une méthode `equals` (qui est la redéfinition de la même méthode définie dans la classe `Object`).

Il est souvent utile d'obtenir la représentation sous la forme d'une chaîne de caractères d'un type primitif (`byte`, `int`, `boolean`, ...). Il existe une alternative à la concaténation qui consiste à utiliser les méthodes `valueOf` de la classe `String`.

L'opération inverse est elle aussi souvent utile, ne serait-ce que pour obtenir la valeur d'un entier dont la représentation sous forme de chaîne de caractères a été fournie à l'application à travers les arguments de la ligne de commande (paramètres `String[] args` de la méthode `main`). Les méthodes associées ne se trouvent cependant pas dans la classe `String` mais dans chacune des classes (appartenant au paquetage `java.lang`) encapsulant les types primitifs (présentées à la fin du TP précédent) : `Boolean`, `Byte`, ...Dans chacune de ces classes, que l'on appellera `X`, on trouve une méthode statique `X.parseX(String s)` (`Byte.parseByte(String s)`, ...).

4 Annexe n°4 : La variable d'environnement `CLASSPATH`

La variable d'environnement `CLASSPATH` est utilisée de manière différente au moment de la compilation (par le compilateur donc) et au moment de l'exécution (par la machine virtuelle).

Le processus de compilation conduit (en cas de réussite) à la génération d'un fichier `.class` contenant le code interprétable de la classe spécifiée en entrée. Lorsque le compilateur détecte une instruction où l'utilisation d'une autre classe est faite (appel de méthode, accès à un attribut), il tente de se procurer le fichier `.class` associé à cette classe afin de vérifier simplement que la-dite classe est correctement utilisée (i.e. l'attribut ou la méthode existent, sont visibles, ...). Le compilateur sait implicitement localiser ces fichiers pour les classes des bibliothèques fournies avec le SDK (par exemple la classe `System`). Pour les autres, et notamment celles écrites par les développeurs tiers, il utilise le contenu de la variable d'environnement `CLASSPATH` (qu'il est donc utile de bien initialiser et de rendre publique). La valeur de cette variable est une suite de chemins de recherche séparés par `:` (convention Linux) ou `;` (convention Dos/Windows). L'ordre des chemins dans cette variable est important, le compilateur cessant de chercher lorsqu'il a trouvé un chemin le conduisant au fichier souhaité. Lorsque cette variable n'est pas définie, il est considéré par défaut que la recherche s'effectue uniquement dans le répertoire courant.

Lors de l'exécution d'une application, les classes sont chargées au fur et à mesure de leur première utilisation, en commençant par celle contenant le `main`. La machine virtuelle a donc besoin, comme le compilateur, d'obtenir les fichiers `.class` correspondant aux différentes classes. La variable d'environnement `CLASSPATH` est alors utilisée à cette fin, de la même manière. Il est cependant important d'avoir à l'esprit que le fichier `.class` utilisé par la machine virtuelle n'est pas forcément celui ayant été utilisé par le compilateur. Si le contenu de ces deux fichiers n'est pas identique, il peut survenir des erreurs lors de l'exécution.

5 Annexe n°5 : Javadoc

Le `SDK` fournit un outil de génération de documentation (avec pour cible privilégiée le langage HTML) nommé `javadoc` basé sur l'utilisation de balises (*tags*) dans le code source. Vous pouvez consulter la partie de la documentation en ligne traitant de cette outil afin de vous familiariser avec les balises et les paramètres de la ligne de commande. La finalité de cet outil est de fournir une documentation pour chaque classe, détaillant notamment la signification des attributs, des méthodes, des paramètres et valeurs de retour.

Les commentaires JavaDoc s'insèrent toujours avant ce qu'ils sont censés décrire, et peuvent mélanger du texte qui sera inséré tel quel dans la documentation et des balises interprétées par l'outil.

Il est par exemple possible, en utilisant cette syntaxe, de préciser à quoi sert la classe, qui en est l'auteur et quel est le numéro de version :

```
/**
 * La classe Lampe ...
 * @author S.Jean
 * @version 1.0
 */
public class Lampe {...}
```

Il est aussi possible de préciser à quoi sert un attribut :

```
/**
 * La puissance de la lampe
 */
public int puissance;
```

Il est également possible de préciser à quoi sert une méthode (ou un constructeur) :

```
/**
 * Constructeur avec paramètre.
 * @param p La puissance de la lampe.
 */
```

```
public Lampe(int p) {...}

/**
 * Obtention de la puissance de la lampe.
 * @return la puissance de la lampe.
 */
public int getPuissance() {...}
```