

Bases non objet de Java¹

1 Les types primitifs

Deux types *spéciaux* :

- type `null` (référence non initialisée)
- type `void`

Les types *classiques* : le type booléen, le type caractère et ...

les **booléens** en Java `boolean`, contient les valeurs `{true, false}`;

les **caractères** : le type Java `char` utilise le codage `Unicode` sur 16 bits ;
ex : `'a'`

... les **types numériques** : types entier et réel.

1.1 Les entiers

Les *entiers signés* sont représentés en complément à deux :

les **octets** type Java `byte` sur 8 bits, intervalle $[-2^7, 2^7 - 1]$ avec $2^7 = 128$, ex : 15

les **entiers courts** type Java `short` sur 16 bits, intervalle $[-2^{15}, 2^{15} - 1]$ avec $2^{15} = 32768$, ex : 32189

les **entiers** type Java `int` sur 32 bits, avec $2^{31} \sim 2.10^9$, ex : 1235234567

les **entiers longs** type Java `long` sur 64 bits, avec $2^{63} \sim 9.10^{18}$, suffixés par `L`, ex : 2147483648L

1.2 Les réels

Pour les *réels* ou *flottants* Java adopte le codage en virgule flottante de la norme `IEEE754` en simple et double précision. *Virgule flottante* signifie qu'on représente un réel sous la forme $+/- m * b^e$ où $+/-$ est le signe, m la mantisse (entière), b la base et e l'exposant (qui fait "flotter" la virgule). La norme `IEEE754` impose la représentation plus précise :

- $(-1)^s * (1 + M) * 2^{E-127}$ sur 32 bits (simple précision)
- $(-1)^s * (1 + M) * 2^{E-1023}$ sur 64 bits (double précision)

où s est le bit de signe (sur 1 bit donc), l'exposant E est codé sur 8 bits ou 11 bits, et la mantisse M est codée sur 23 bits ou 52 bits. Dans la syntaxe Java :

simple précision type Java `float`, littéraux suffixés par `F`, intervalle $[-10^{-38}, -10^{38}] \cup [10^{-38}, 10^{38}]$,
ex : `112.2e-45F`, `-34E-555F`, `3.14F`;

double précision type Java `double`, littéraux suffixés optionnellement par `D`, intervalle $[-10^{-308}, -10^{308}] \cup [10^{-308}, 10^{308}]$,
ex : `1.34e56D`, `3.14`

2 Les opérateurs et prédicats de base

Opérateurs numériques :

- addition `+`, soustraction `-` ;
- multiplication `*`, division entière ou réelle `/` ;
- moins unaire `-` ;
- modulo `%` : $x\%y = x - (x/y)*y$, surtout utilisé pour les entiers.

On a aussi les "raccourcis" des post/pré incrément/décément :

- `x++` : ajoute 1 à `x` puis délivre la valeur de `x` ;
- `--x` : délivre la valeur de `x` puis soustrait 1 à `x` ;
- symétriquement `++x` et `x--`.

NB : l'opérateur de puissance n'est pas un opérateur de base. On utilisera `pow` de la classe `java.Math`.

Opérateurs booléens :

- et logique `&&`, ou logique `||` : ces opérateurs sont *pareseux* ;
- et logique `&`, ou logique `|` : ces opérateurs sont *stricts* (non pareseux) ;
- ou exclusif `^` (strict) ;
- négation logique `!`.

Prédicats de comparaison de base :

- les classiques `<` `<=` `>` `>=` ;
- l'égalité `==` ;
- la différence `!=`.

3 Les commentaires

```
// toute la ligne est en commentaire
/* commentaire entre balises */
```

4 La déclaration de variables

Dans la mesure du possible commenter et donner des noms parlants. De la forme² :

```
<type> <nom_var> [= <valeur_init> ] ;
```

```
ex: int nbJoueurs = 3; // nombre de joueurs
    int y; // ordonnée
    boolean fini = false; // le jeu n'est pas fini
```

On peut faire des déclarations en cascade, à éviter.

Syntaxe des identificateurs Java : *(lettre | _)(lettre | _ | chiffre)**

NB : Java est sensible à la casse.

5 Les instructions

Ne pas oublier les ; de fin d'instruction.

5.1 L'affectation

De la forme : *<nom_var>* = *<expression>* ;

```
ex: y = x+3;
    fini = ! fini;
```

Attention : ne pas confondre l'opérateur d'affectation `=` et l'opérateur de comparaison `==` !!

5.2 Les structures de contrôle

5.2.1 La séquence

On comment exprimer *ensuite...* Deux instructions séparées par ; sont exécutées en séquence.

5.2.2 Le bloc d'instructions

Un bloc est délimité par `{` et `}`. Toute variable déclarée dans un bloc a une portée (ou visibilité) limitée à ce bloc (n'est connue que dans ce bloc). Toute séquence d'instructions *doit* être incluse dans un bloc.

```
{ int i = 5; } i = 6; // erreur
```

²*<type>* est à remplacer par un type comme `int` ou `boolean`, *<nom_var>* par un nom de variable, *<valeur_init>* par une valeur initiale, etc. Les `[]` délimitent les parties optionnelles.

¹Merci à Mirabelle Nebut, rédactrice principale de ce document.

7.5 Accès indexé

Les éléments d'un tableau sont numérotés (on dit *indexés* ou *indicés*) à partir de 0. Si $\langle nom_tab \rangle$ est un tableau *déjà créé*, alors on accède à son i ème élément par :

$\langle nom_tab \rangle [\langle expr_indice \rangle]$

ex : `tab1[0]`, ..., `tab1[9]` sont des expressions correctes, mais `tab1[10]` est une expression incorrecte.

On écrira par exemple :

```
tab1[0] = 1;
tab1[1] = 2;
int x = 5;
tab1[x-3] = 3;
tab1[x-2] = 4;
....
tab1[9] = 10;
```

Attention : l'indice doit être compris entre 0 *inclus* et `tab.length` *exclu*. On écrira donc typiquement :

```
for (int i=0; i<tab.length; i++) {
    tab[i] = i+1;
}
```

7.6 Itération sur les éléments d'un tableau

Cette notion n'existe qu'à partir de la version 1.5 de Java.

Lorsque l'on souhaite parcourir un tableau et réaliser une manipulation sur chacun de ses éléments on peut utiliser la syntaxe (dite "à la *for-each*") suivante :

```
float[] tab = ... ;
float somme = 0;
for (float element : tab) {
    // element prend successivement toutes les valeurs
    somme = somme + element;
}
```

qui équivaut à

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

8 Tableaux multi-dimensionnels

Jusqu'ici on a vu les tableaux à une dimension. Un tableau à deux dimensions est une matrice (un tableau de tableaux). Un tableau à trois dimensions est un cube (un tableau de matrices, ou une matrice de tableaux). La déclaration et la création de tableaux multi-dimensionnels suivent le cas mono-dimensionnel :

$\langle nom_tab \rangle \langle type_elt \rangle [\dots]$;
 $\langle nom_tab \rangle = \text{new } \langle type_elt \rangle [\langle expr_nb_1 \rangle] \dots [\langle expr_nb_n \rangle]$;

```
ex : int [] [] mat;
    mat = new int [3] [2];
    mat [1] [0] = 4;
```

`mat` est un tableau de 3 cases, chacune contenant un tableau de 2 cases.

Il faut obligatoirement fixer la première dimension à la création. Mais les autres dimensions peuvent être fixées ensuite. Conséquence : un tableau n'est pas forcément "rectangulaire".

```
ex : int [] [] damier;
    damier = new int [4] [4];
    int [] [] damier_cassé;
    damier_cassé = new int [4] [];
    damier_cassé [0] = new int [4];
    damier_cassé [1] = new int [3];
    ...
    damier_cassé [3] = new int [4];
```