Bases non objet de Java[1]

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 $+/-\,\mathtt{m} * \mathtt{b\hat{}e}$ 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 :

- $(-1)^{\mathtt{s}} * (1 + \mathtt{M}) * 2^{\mathtt{E}-127}$ sur 32 bits (simple précision)
- $(-1)^{\mathtt{s}} * (1 + \mathtt{M}) * 2^{\mathtt{E}-1023}$ 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/décrément :

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

- `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 *paresseux* ;
- et logique &, ou logique `|` : ces opérateurs sont *stricts* (non paresseux) ;
- 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 forme[2] :

⟨*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'affection = et l'opérateur de comparaison == !!

5.2 Les structures de contrôle

5.2.1 La séquence

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

[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.

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
```

5.2.3 Structures conditionnelles

Le *si... alors...* s'écrit : `if (` ⟨*condition*⟩ `)`
⟨*code_vrai*⟩

ce qui peut donner :

```
if (x == y)
  z = 0;
```

ou

```
if (x == y) {
   z = 0;
}
```

Le *si... alors... sinon...* s'écrit : `if (` ⟨*condition*⟩ `)`
⟨*code_vrai*⟩
`else`
⟨*code_faux*⟩

ce qui peut donner :

```
if (x%2 == 0)
  x = x / 2;
else
  x = x + 1;
```

```
if (x == y) {
  fini = true; x = 0;
} else fini = false;
```

5.2.4 Les structures itératives

Ou comment exprimer la répétition en boucle.

La boucle tant que `while (` ⟨*condition*⟩ `)`
⟨*corps*⟩

1. si la condition est fausse,
   la boucle termine
   sinon 2. exécuter le corps
   3. retour en 1.

ex :

```
{
int x = 2;
boolean fini = x < 0; // fin de boucle
int nb = 0; // nb tours de boucle
while ( ! fini ) {
   x = x-1;
   nb = nb + 1;
   fini = x < 0;
}}
```

invariant de boucle : `x + nb = 2`
sortie de boucle `x = -1` $\Longrightarrow$ `nb = 3`

```
{
int x = 3;
int nb = 0; // nb tours de boucle
while ( x > 0 ) {
   x = x-1;
   nb = nb + 1;
}}
```

invariant de boucle : `x + nb = 3`
sortie de boucle `x = 0` $\Longrightarrow$ `nb = 3`

La boucle for `for (` ⟨*initialisation*⟩`;`
⟨*condition*⟩`;`
⟨*expr_progression*⟩ `)`
⟨*corps*⟩

1. effectuer l'initialisation
2. si la condition est fausse le for termine
   sinon 3. effectuer le corps
   4. évaluer l'expression
   5. retour en 2.

ex : `for (i=0, j=0; i<3; i=i+1)`
`j = j+2;`

ou

```
i = 0; j = 0;
while (i<3) {
   j = j+2;
   i = i+1;
}
```

invariant `j = i*2`
sortie de boucle `i = 3` $\Longrightarrow$ `j=6`

Possibilité de déclarer localement les "variables de boucles" lors de l'initialisation :

```
int j = 2;
for (int i=0; i<3; i=i+1)
   j = j+2;
```

Boucle faire ... jusqu'à

```
do
   ⟨corps⟩
while ( ⟨condition⟩ )
```

éq à

```
< corps >;
while ( < condition > )
   < corps >;
```

donc idem boucle tant-que mais en testant la condition à la fin. Le corps de la boucle est nécessairement exécuté au moins une fois.

6 Les types énumérés

Cette notion n'existe qu'à partir de la version 1.5 de Java.
Ils se définissent dans un fichier à part. Pour le type ⟨*NomEnum*⟩ le nom de ce fichier sera nécessairement "⟨NomEnum⟩.java" et contiendra la définition du type énuméré ⟨*NomEnum*⟩, ainsi :

```
// fichier ⟨NomEnum⟩.java
   public enum ⟨NomEnum⟩ {
      ⟨val_1⟩, ⟨val_2⟩, ···, ⟨val_n⟩;
   }
```

où les ⟨*val_i*⟩ représentent les valeurs du type énuméré. On référence ces valeurs par : ⟨*NomEnum.val1*⟩

ex :

```
public enum Saison { hiver, printemps, ete, automne }
```

on peut alors utiliser :

```
Saison s = Saison.hiver;
```

Cette notion sera développée et détaillée dans la suite du semestre.

7 Les tableaux

Les tableaux permettent de stocker un nombre prédéfini d'objets d'un même type simple (type de base). On aura par exemple un tableau de 10 entiers, un tableau de 2 booléens... Pour utiliser un tableau il faut le *déclarer puis* le *créer*.

7.1 Déclaration d'un tableau

Pour déclarer un tableau on précise le type de ses éléments, pas son nombre de cases (le nombre d'éléments qu'il contient) :

⟨*type_elt*⟩ [] ⟨*nom_tab*⟩ ;

ex :
```
int[] tab1;
boolean[] tab2;
```

7.2 Création d'un tableau de taille fixée

La création d'un tableau ⟨*nom_tab*⟩ *déjà déclaré* se fait en utilisant new et en précisant sa taille (son nombre de cases) :

⟨*nom_tab*⟩ = new ⟨*type_elt*⟩ [⟨*expr_nb*⟩];

La taille n'est pas nécessairement connue statiquement (cad qu'elle peut n'être connue qu'à l'exécution du programme).
ex :
```
tab1 = new int[10];
tab2 = new boolean[x+3]
```
Alors tab1 contient 10 cases numérotées de 0 à 9. tab2 contient x+3 cases.

7.3 Taille

La taille d'un tableau fait partie des propriétés fournies par Java : si ⟨*nom_tab*⟩ est un tableau *déjà créé*, alors

⟨*nom_tab*⟩.length

est son nombre d'éléments.
ex : tab1.length vaut 10.

7.4 Initialisation à la déclaration

On peut initialiser un tableau lors de sa déclaration par une liste de valeurs séparées par des `,`, entourée d'accolades. Dans ce cas il n'y pas besoin de créer explicitement le tableau par `new`.
ex : `int[] tab = {1,2,3,4,12};`
On a ici `tab.length`$= 5$.

7.5 Accès indexé

Les éléments d'un tableaux sont numérotés (on dit *indexés* ou *indicés*) à partir de 0. Si ⟨*nom_tab*⟩ est un tableau *déjà créé*, alors on accède à son ième élément par :

⟨*nom_tab*⟩`[`⟨*expr_indice*⟩`]`

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

```
tab1[O] = 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 élements d'un tableau

Cette notion n'existe qu'à partir de la version 1.5 de Java.
Lorsque l'on souhaire 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-dimentionnels

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-dimentionnels suivent le cas mono-dimentionnel :

⟨*nom_tab*⟩ ⟨*type_elt*⟩ `[]...[];`
⟨*nom_tab*⟩ `=` `new` ⟨*type_elt*⟩ `[`⟨*expr_nb*$_1$⟩ `]` ... `[` ⟨*expr_nb*$_n$⟩ `];`

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];
```