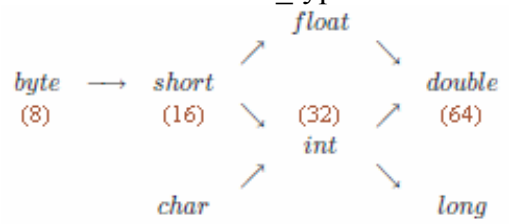
Cours 5

I. Types primitifs

I.1 Déjà vus : `boolean, int, long, double`
I.2 "Nouveaux" :
I.2.A `float` (réel/32 bits) : 6 chiffres significatifs, constantes littérales = `3.14F`
I.2.B `short` (entier/16 bits, signé)
I.2.C `byte` (entier/8 bits, signé)
I.2.D `char` :
- chaque caractère est codé par un entier sur 16 bits (et non plus 7 ou 8)
- C/C++: même nom, mais 8 bits, signé, ASCII
- JAVA: 16 bits, non signé, Unicode
- mais Unicode ⊃ ASCII (128 premiers caractères)
0 < contrôles < * < espace < * < chiffres < * < minuscules < * < majuscules < * ≤ 127 < ascii_étendu ≤ 255
- constantes littérales 'a' ou '\t' (1 ≠ '1' ≠ "1")
- pas d'opérateurs sauf conversion ⟷ `int`
- `String s = "Bonjour"; char c = s.charAt(0); => c` vaut `'B'`

I.3 Résumé :
types primitifs = `boolean` + types "numériques"
types "numériques" = `char` + types entiers + types réels
types entiers = `byte, short, int, long`
types réels = `float, double`
I.4 Conversions (entre types primitifs)
I.4.A) Syntaxe : (type_primitif_souhaité)expression_d'un_autre_type_primitif
I.4.B) impossible : `boolean` ⟷ autre_type
I.4.C) sans perte :



I.4.D) avec perte : les autres *(y compris si même nombre d'octets ; pourquoi ?)*

II. Tableaux

II.1 Utilité
- regroupement de données homogènes *(et pour données hétérogènes ?)*
- manipulation en une seule entité (notamment passage de paramètre)
- possibilité de faire des boucles *(comment afficher tous les attributs d'un objet ?)*

II.2 Définition
- tableau de taille fixe : à la compilation (C/C++) ou à l'exécution (Java)
- regroupement de données de même type, accessibles par leur rang, dont la taille est fixée à l'exécution, mais une fois pour toutes

II.3 Nouveau type
- éléments de type `T` => tableau de type `T[]`
- considéré comme une classe => hérite d'`Object` => un tableau est un objet => référence
- attribut public `length` = taille du tableau = nombre de cases ≠ nombre d'éléments utiles *(et ne pas confondre avec la méthode `length()` de la classe `String`)*

II.4 Déclaration
- *type var;* => `T[] aTab;`
- T est n'importe quel type, primitif ou objet, y compris tableau
- déclaré mais pas créé => 0 cases mémoire

II.5 Création (=>allocation mémoire)

- `aTab = new T[taille];`
- taille est une expression entière positive (pas `long`), sinon NegativeArraySizeException
- pour les variables locales, souvent en une seule fois : `T[] vTab = new T[taille];`

II.6 Accès à une valeur (lecture/écriture)

- par le rang, appelé indice
- expression entière (pas `long`) entre 0 et aTab.length-1, sinon ArrayIndexOutOfBoundsException ! *(gros avantage sur C/C++)*
- 1ère valeur = `aTab[0]`, ou `aTab[2*i-1] = -x/2;`

II.7 Initialisation

- type primitif : `typePrimitif var = expression;`, exemple: `double r = 3.14;`
- type objet : `Classe var = new Classe( valeurs );`, exemples: `Point p = new Point(2,4);`
- classe particulière : `String s = new String( "Bonjour" );` peut être abrégée en `String s = "Bonjour";`
- Tableau **(syntaxe particulière)** : `int[] t = new int[] {10,20,30};` peut être abrégée en `int[] t = {10,20,30};` **uniquement dans une initialisation**
- Exemple : `int[] nbJours = {31, 28, 31, ..., 30, 31};` remplace les 13 lignes
  `int[] nbJours = new int[12];`
  `nbJours[0] = 31; nbJours[1] = 28; ... nbJours[11] = 31;`

II.8 Saisie/affichage : rien de prévu => à redéfinir soi-même ! (`nbJours.toString()` retourne `"[I@abcdef"`)

II.9 fonctionnements non souhaités :

= ne recopie pas les valeurs
== ne compare pas les valeurs
`final` ne protège pas les valeurs

II.10 java.util.Arrays (equals, toString, fill, sort, binarySearch)

- méthodes statiques => un paramètre de plus !
- `void fill(tab,val)` remplit toutes les valeurs de tab avec val
- `String toString(tab)` retourne `"[10, 20, 30]"`
- `boolean equals(tab1,tab2)` compare les tailles et les valeurs des 2 tableaux
- `void sort(tab)` trie les valeurs de tab
- `int binarySearch(tab,val)` retourne l'indice de val trouvée dans tab

II.11 Cas particulier : caractères

- `String` ≠ `char[] tc;`
- mais `String s = new String(tc);` et `tc = s.toCharArray();`
- car `String` ne peut être modifiée alors que `char[]` si !

II.12 Multi-dimensionnels

II.12.a Un élément de tableau peut être de n'importe quel type, donc aussi un tableau ! Un tableau de tableau donne un tableau bi-dimensionnel.

II.12.b `int[][] bidim = new int[3][2];` ==> nouveau type (`int[][]`) et 2 indices. `bidim[ligne][colonne]` pour accéder à une case.
`int[][]` peut être vu comme un tableau de `int[]` (puisque `X[]` est un tableau de `X`)
Attention ! `new int[3][2]` ne signifie pas un tableau de `2 int[3]` **mais** un tableau de `3 int[2]` !

II.12.c initialisation possible : `int[][] bidim = { {11, 12}, {21, 22}, {31, 32} };`

II.12.d Un tableau peut avoir plus de 2 dimensions, et même plus de 5 ! Exemple :

```
Pixel[][][][][][][] journee = new Pixel[20][10][30][25][3][1920][1080];
                                        ^   ^   ^   ^  ^     ^     ^
                          plages par journée    ^   ^  ^     ^     ^
                                 spots par plage    ^  ^     ^     ^
                                   secondes par spot   ^     ^     ^
                                      images par seconde     ^     ^
                                   plans couleurs par image        ^
                                                 lignes par plan     ^
                                                    colonnes par ligne
```

et pourquoi pas par an, par chaîne, etc ...

II.12.e `Arrays.deepToString( tab )` et `Arrays.deepEquals( tab1, tab2 )`

III. Boucles

III.1 Déjà vue : la boucle TANT QUE

`while (` expression booléenne `) {` instructions à répéter`}`

III.2 Nouvelle : la boucle POUR

III.2.A) Équivalente à la boucle `while` (intérêt syntaxique) :

initialisation`;` `while (` *condition de continuation* `) {` *instructions à répéter*`;` *progression*`;` `}`

initialisation: type_entier variable `=` expression_entière *par exemple:* `int i=0`

condition de continuation: comparaison sur la variable *par exemple:* `i <= 15`

progression: modification de la variable *par exemple:* `i = i + 1`

III.2.B) Syntaxe :

`for (` initialisation; condition de continuation; progression `) {` instructions `}`

par exemple: `for ( int i=0; i<=15; i=i+1 ) {` instructions `}`

Attention ! Seul cas en Java où le caractère **;** ne termine pas une instruction.

III.2.C) Opérateurs souvent utilisés

`i++` est équivalent à `i=i+1` et non à `i+1`

`i--` est équivalent à `i=i-1` et non à `i-1`

`i+=4` est équivalent à `i=i+4` et non à `i+4`

`i-=3` est équivalent à `i=i-3` et non à `i-3`

`i*=2` est équivalent à `i=i*2` et non à `i*2`

III.2.D) Imbrication : 2 for, 2 compteurs (1 lent, 1 rapide), notamment pour les tableaux bi-dimensionnels

III.3 Nouvelle : la boucle FAIRE TANT QUE

- "contraire" de la boucle *Répéter jusqu'à* que l'on trouve dans d'autres langages
- intérêt : instructions toujours exécutées au moins une fois avant le test de continuation

III.4 Caractéristiques et choix de boucles

III.4.A) `while` :

`while (` *expression booléenne* `) {` *instructions à répéter* `}`

- nombre de tours inconnu a priori
- test au début, peut tourner 0 fois
- pb d'initialisation de l'expression booléenne
- pb de modification de l'expression booléenne

III.4.B) `do while` :

`do {` *instructions à répéter* `} while (` *expression booléenne* `);`

- nombre de tours inconnu a priori
- test à la fin, tourne au moins une fois
- les 2 pbs de la boucle while peuvent souvent être évités
- équivalente à :

instructions à répéter `while (` *expression booléenne* `) {` *instructions à répéter* `}`

III.4.C) `for` :

`for (` *instruction d'initialisation* `;` *condition de continuation* `;` *instruction de progression* `) {` *instructions à répéter* `}`

- nombre de tours calculable a priori
- test au début, peut tourner 0 fois
- les 2 pbs de la boucle while sont traités
- il est fortement conseillé d'inclure la déclaration de variable dans l'*instruction d'initialisation* pour que la variable reste locale
- équivalente à :

instruction d'initialisation`;` `while (` *condition de continuation* `) {` *instructions à répéter instruction de progression*`;` `}`

III.4.D) Critères de choix : 2 questions successives

- première question à se poser :
le nb de tours peut-il être connu à l'avance ?
- si OUI, c'est une boucle `for`
- si NON, seconde question à se poser :
l'instruction dans la boucle doit-elle être exécutée au moins une fois ?
- si OUI, c'est une boucle `do while`
- si NON, c'est une boucle `while`

IV. Développer sans BlueJ (et sans IDE)

IV.1 Édition de texte (`nedit, emacs, nano, vi, vim, notepad, kwrite, kate, ultraedit, ...`)

```
nedit UneClasse.java &
```

IV.2 Compilation

```
javac UneClasse.java
```
ou
```
javac *.java
```

IV.3 Exécution

```
java UneClasse
```
ou
```
java UneClasse mot1 mot2
```
mais quelle méthode est lancée ? (sinon `NoSuchMethodError`)

IV.4 Signature obligatoire :

```
public static void main( String[] pArgs ) { instructions }
^^3^^^ ^^4^^^ ^2^^ ^1^^  ^^^5^^^^ ^^6^^     ^^^^^7^^^^^^
```

- 1: le nom est imposé (veut dire "principal")
- 2: car c'est une procédure (qui ne retourne rien)
- 3: car elle est appelée en dehors de la classe (par le système d'exploitation)
- 4: car il serait ennuyeux que le système d'exploitation doive d'abord créer un objet avant de pouvoir appeler cette procédure
- 5: un et un seul paramètre obligatoire (aucun=interdit, 2 ou plus=interdit) et forcément du type *tableau de* `String`
la taille est facilement disponible et `pArgs[0]` contient le premier argument après `java NomClasse` (alors que c'est `argv[1]` en C ou C++)
- 6: seule liberté : le choix du nom du paramètre (mais presque tout le monde utilise `args` !)
- 7: Si le corps est vide, le programme ne fera rien. S'il se termine par `System.exit( valeur );` , la *valeur* sera retournée au système d'exploitation comme code d'erreur.

IV.5 Documentation

publique (pour les utilisateurs de la classe) : `javadoc *.java -author -version -d`*répertoire*

privée (pour les programmeurs, maintenance de la classe) : `javadoc *.java -author -version -d`*répertoire* `-private -linksource`

V. Style d'écriture des programmes en Java : recommandations à respecter (rappels)

Lire le poly :

tout jusqu'à la section 2.4.1, sections 2.5, 3.1, 3.2.2, 3.6, 4, **5.1**, 5.2.0, 6, 7.1, 7.2, 8.1, 8.2.1.1, **8.2.2**, 9.1 à 9.5, 13.3, et annexes 6 & 7