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 => 0..65535
- mais Unicode ⊃ ASCII (128 premiers caractères)
0 < contrôles < * < espace < * < chiffres < * < majuscules < * < minuscules < * ≤ 127 < ascii_étendu ≤ 255 < suite_unicode ≤ 65535
- constantes littérales '0'..'9', 'A'..'Z', 'a'..'z', '+' ou '\t'
- 3 affichages identiques, mais bien distinguer : 1 ≠ '1' ≠ "1" :

00000000	00000000	00000000	00000001

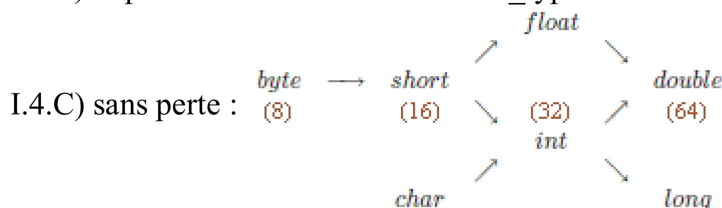
00000000	00110001

10010011	01011100	00110001	11011100

- opérateurs de comparaison : les 6 habituels
- opérateur de conversion ←→ `int`
- opérateurs arithmétiques `+` et `-` : exemple: maj+'a'-'A' → min
- `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 ?)*
- Exemples sans tableau et avec tableau

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; 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

III.3.A)

- "contraire" de la boucle *Répéter jusqu'à* que l'on trouve dans d'autres langages
- intérêt : instructions toujours exécutées au moins 1 fois avant le test de continuation

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
- équivalente à :

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

III.4 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 ? (signature recherchée, sinon `NoSuchMethodError`)

IV.4 Signature obligatoire pour cette méthode :

```
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++)
par exemple, si la commande est celle en orange ci-dessus, `pArgs` sera composé des 2 *String* `"mot1"` et `"mot2"`
- 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 (0 s'il n'y a pas 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

Dernière mise à jour : dimanche 1er avril 2012 à 22h41.

```
import javax.swing.JOptionPane;

/**
 * Classe SansTableau, c'est fastidieux !
 * Apres avoir lu la classe, imaginez qu'il y ait 24 notes, ou 180 ...
 *
 * @author DB
 * @version 2012.03.31
 */
public class SansTableau
{
    public static final int TAILLE = 8;
    private double aNote1;
    private double aNote2;
    private double aNote3;
    private double aNote4;
    private double aNote5;
    private double aNote6;
    private double aNote7;
    private double aNote8;

    /**
     * procedure principale (pour tester)
     */
    public static void test()
    {
        SansTableau vObj = new SansTableau( TAILLE );
        vObj.saisie();
        vObj.affiche();
        System.out.println( "Moyenne = " + vObj.moyenne() );
    } // test()

    /**
     * Constructeur
     * @param pTaille le nombre de cases souhaite pour ce tableau
     */
    public SansTableau( final int pTaille)
    {
        if ( pTaille != TAILLE )  System.out.println( "La taille doit forcement etre "+
TAILLE+" !" );
    } // SansTableau()

    /**
     * procedure de saisie de tous les nombres au clavier
     */
    public void saisie()
    {
        this.aNote1 = litNombreAuClavier( "Saisir nombre numero 1 : " );
        this.aNote2 = litNombreAuClavier( "Saisir nombre numero 2 : " );
        this.aNote3 = litNombreAuClavier( "Saisir nombre numero 3 : " );
        this.aNote4 = litNombreAuClavier( "Saisir nombre numero 4 : " );
        this.aNote5 = litNombreAuClavier( "Saisir nombre numero 5 : " );
        this.aNote6 = litNombreAuClavier( "Saisir nombre numero 6 : " );
        this.aNote7 = litNombreAuClavier( "Saisir nombre numero 7 : " );
        this.aNote8 = litNombreAuClavier( "Saisir nombre numero 8 : " );
    } // saisie()
```

```java
    /**
     * procedure d'affichage des nombres du tableau
     */
    public void affiche()
    {
        System.out.println( this.aNote1 );
        System.out.println( this.aNote2 );
        System.out.println( this.aNote3 );
        System.out.println( this.aNote4 );
        System.out.println( this.aNote5 );
        System.out.println( this.aNote6 );
        System.out.println( this.aNote7 );
        System.out.println( this.aNote8 );
    } // affiche()

    /**
     * fonction qui calcule la moyenne des nombres du tableau
     */
    public double moyenne()
    {
        double vSomme = this.aNote1+this.aNote2+this.aNote3+this.aNote4
                        +this.aNote5+this.aNote6+this.aNote7+this.aNote8;
        return vSomme/TAILLE;
    } // moyenne()

    /**
     * fonction de saisie d'un nombre au clavier (avec message)
     * @param pMes message a afficher avant la saisie
     * @return le nombre saisi au clavier
     * @throws NumberFormatException si ce n'est pas un nombre
     */
    private double litNombreAuClavier( final String pMes )
    {
        return Double.parseDouble( JOptionPane.showInputDialog( pMes ) );
    } // litNombreAuClavier(.)
} // SansTableau
```

```java
import javax.swing.JOptionPane;

/**
 * Classe AvecTableau, c'est plus facile !
 *
 * @author DB
 * @version 2012.03.31
 */
public class AvecTableau
{
    private double[] aNotes;

    /**
     * procedure principale (pour tester)
     */
    public static void test()
    {
        AvecTableau vTab = new AvecTableau( 8 );
        vTab.saisie();
        vTab.affiche();
        System.out.println( "Moyenne = " + vTab.moyenne() );
    } // test()

    /**
     * Constructeur
     * @param pTaille le nombre de cases souhaite pour ce tableau
     */
    public AvecTableau( final int pTaille)
    {
        int vTaille;
        if ( pTaille < 1 )  vTaille = 1;  else vTaille = pTaille;
        this.aNotes = new double[vTaille];
    } // AvecTableau()

    /**
     * procedure de saisie de tous les nombres au clavier
     */
    public void saisie()
    {
        for ( int vI=0; vI < this.aNotes.length; vI=vI+1 ) {
            this.aNotes[vI] = litNombreAuClavier( "Saisir nombre numero "+(vI+1)+" : "
);
        } // for
    } // saisie()

    /**
     * procedure d'affichage des nombres du tableau
     */
    public void affiche()
    {
        for ( int vI=0; vI < this.aNotes.length; vI=vI+1 ) {
            System.out.println( this.aNotes[vI] );
        } // for
    } // affiche()
```

```java
    /**
     * fonction qui calcule la moyenne des nombres du tableau
     */
    public double moyenne()
    {
        double vSomme = 0.0;
        for ( int vI=0; vI < this.aNotes.length; vI=vI+1 ) {
            vSomme = vSomme + this.aNotes[vI];
        } // for
        return vSomme/this.aNotes.length;
    } // moyenne()

    /**
     * fonction de saisie d'un nombre au clavier (avec message)
     * @param pMes message a afficher avant la saisie
     * @return le nombre saisi au clavier
     * @throws NumberFormatException si ce n'est pas un nombre
     */
    private double litNombreAuClavier( final String pMes )
    {
        return Double.parseDouble( JOptionPane.showInputDialog( pMes ) );
    } // litNombreAuClavier(.)
} // AvecTableau
```

Program Style Guide

Version 2.0.2

1. Naming

1.1 Use meaningful names.

Use descriptive names for all identifiers (names of classes, variables and methods). Avoid ambiguity. Avoid abbreviations. Simple mutator methods should be named set*Something*(...). Simple accessor methods should be named get*Something*(...). Accessor methods with boolean return values are often called is*Something*(...), for example, isEmpty().

1.2 Class names start with a capital letter.

1.3 Class names are singular nouns.

1.4 Method and variable names start with lowercase letters.

All three - class, method and variable names - use capital letters in the middle to increase readability of compound identifiers, e.g. numberOfItems.

1.5 Constants are written in UPPERCASE.

Constants occasionally use underscores to indicate compound identifiers: MAXIMUM_SIZE

1.6 **s**tatic member names (fields and methods) start with a lowercase **s**, except for constants.

1.7 Other variable names start with a specific letter describing their role :

- **a**ttributes (fields) start with a lowercase **a**
- **p**arameters start with a lowercase **p**
- local **v**ariables start with a lowercase **v**

2. Layout

2.1 One level of indentation is four spaces.

2.2 All statements within a block are indented one level.

2.3 Braces for classes and methods are alone on one line.

The braces for class and method blocks are on separate lines and are at the same indentation level, for example:

```
public int getAge()
{
    statements
}
```

2.4 For all other blocks, braces open at the end of a line.

All other blocks open with braces at the end of the line that contains the keyword defining the block. The closing brace is on a separate line, aligned under the keyword that defines the block. For example:

```
while (condition) {
    statements
}

if (condition) {
    statements
}
else {
    statements
}
```

2.5 Always use braces in control structures.

Braces are used in if-statements and loops even if the body is only a single

statement.

2.6 Use a space after a keyword, and before the opening brace of a control structure's block.

2.7 Use a space around operators.

2.8 Use a blank line between methods (and constructors).

Use blank lines to separate logical blocks of code. This means at least between methods, but also between logical parts within a method.

3. Documentation

3.1 Every class has a class comment at the top.

The class comment contains at least
- a general description of the class
- the author's name(s)
- a version number

Every person who has contributed to the class has to be named as an author or has to be otherwise appropriately credited.

A version number can be a simple number, a date, or other formats. The important thing is that a reader must be able to recognise if two version are not the same, and be able to determine which one is newer.

3.2 Every method has a method comment.

3.3 Comments are Javadoc-readable.

Class and method comments must be recognised by Javadoc. In other words: they should start with the comment symbol /**.

And javadoc comments `@param`, `@return`, and `@throws` should be used (if appropriate) in each method comment.

3.4 Code comments (only) where necessary.

Comments in the code should be included where the code is not obvious or difficult to understand (while preference should be given to make the code obvious or easy to understand where possible), and where it helps understanding of a method. Do not comment obvious statements - assume your reader understands Java!

4. Language use restrictions

4.1 Order of declarations: fields, constructors, methods.

The elements of a class definition appear (if present) in the following order: package statement; import statements; class comment; class header; field definitions; constructors; methods.

4.2 Fields may not be public (except for final fields).

4.3 Always use an access modifier.

Specify all fields and methods as either private, public, or protected. Never use default (package private) access.

4.4 Import classes separately.

Import statements explicitly naming every class are preferred over importing whole packages. E.g.

```
import java.util.ArrayList;
import java.util.HashSet;
```

is better than

```
import java.util.*;
```

and classes from the same package must be listed in alphabetical order.

4.5 Always include a constructor (even if the body is empty).

4.6 Always include superclass constructor call.

In constructors of subclasses, do not rely on automatic insertion of a superclass call. Include the super(...) call explicitly, even if it would work without it.

4.7 Initialise all fields in the constructor.

4.8 Always use *this* when possible.

For example :

```
this.aField = 0;
this.setField(0);
```

is better than

```
aField = 0;
setField(0);
```

5. Code idioms

5.1 Use iterators with collections.

To iterate over a collection, use a for-each loop. When the collection must be changed during iteration use an Iterator, not an integer index.

5.2 Use if/else instead of two ifs.

For example :

```
if (a==2) statement1; else statement2;
```

is better than

```
if (a==2) statement1;
if (a!=2) statement2;
```

Copyright notice

Copyright (c) Michael Kölling and David Barnes, completed by Denis Bureau (ESIEE).

This style guide was written for the book Objects First With Java - A Practical Introduction Using BlueJ*.*

Permission is granted to everyone to copy, modify and distribute this document or derived documents in any way. We would appreciate a reference to the original source in derived documents.