Partie I: Bases de Java sans objets

(ING39)

V. Aponte

Cnam

4 septembre 2022

Contenu

1. Caractéristiques
2. Structure d'un programme sans objets
3. Sous-programmes,
4. Types primitifs, expressions, conversions ;
5. Déclarations, instructions simples, lecture et écriture sur la console ;
6. Tableaux
7. Références
8. Blocs, instructions conditionnelles, boucles
9. Exécution et passage de paramètres

1. Caractéristiques principales de Java

1. Caractéristiques principales

- compilé vers du code intermédiaire (code portable) ;
- interprété par une machine virtuelle adaptée à chaque plateforem matérielle ;
- langage fortément typé (typage rigoureux à la compilation) ;
- rammasse-miettes pour gérer la mémoire ;
- tout est classe + paradigme objet ;
- exceptions, paquetages (bas niveau), généricité limitée.

2. Structure d'un programme Java sans objets

2. Structure d'un programme sans objets

```
public class <nom-du-programme> {

    <déclaration-variables-et-sousprogrammes>

    public static void main (String[] args) {

        <déclarations-et-instructions>
    }
    <déclaration-variables-et-sousprogrammes>
}
```

Premier programme : Bonjour

```
/* Affiche 'Bonjour' */

public class Bonjour {
  public static void main (String[] args) {
      System.out.println("Bonjour !!");
  }
}
```

- Le texte entre `/*` et `*/` , et après `//` : des commentaires.
- Le nom du programme (et de la classe définie) : `Bonjour`.
- Le nom du fichier contenant : `Bonjour.java`
- la méthode (sous-programme) **main** est chargé d'orchestrer l'exécution.

Deux sortes de classes

- classe *principale* (une seulement)

  $\Rightarrow$ contient une méthode statique **main**,
  qui peut invoquer d'autres méthodes, utiliser des objets ;

- classes « patrons d'objets » (0 ou plus)

  $\Rightarrow$ pour créer et manipuler des objets ;

Organisation du code

application = classes « patrons d'objets » + 1 classe principale

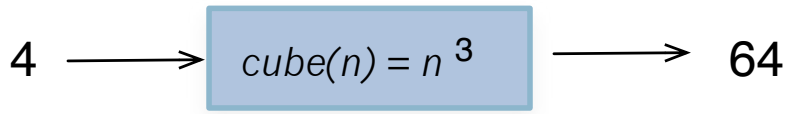
- Plusieurs classes « patrons d'objet » + une seule classe principale ;
- 1 classe par fichier
- plusieurs fichiers dans un paquetage ou *module*
- paquetage matérialisé par un répertoire contenant tous les fichiers du paquetage,
- chaque fichier du paquetage `toto` *doit* avoir `package toto;` en 1ère ligne.

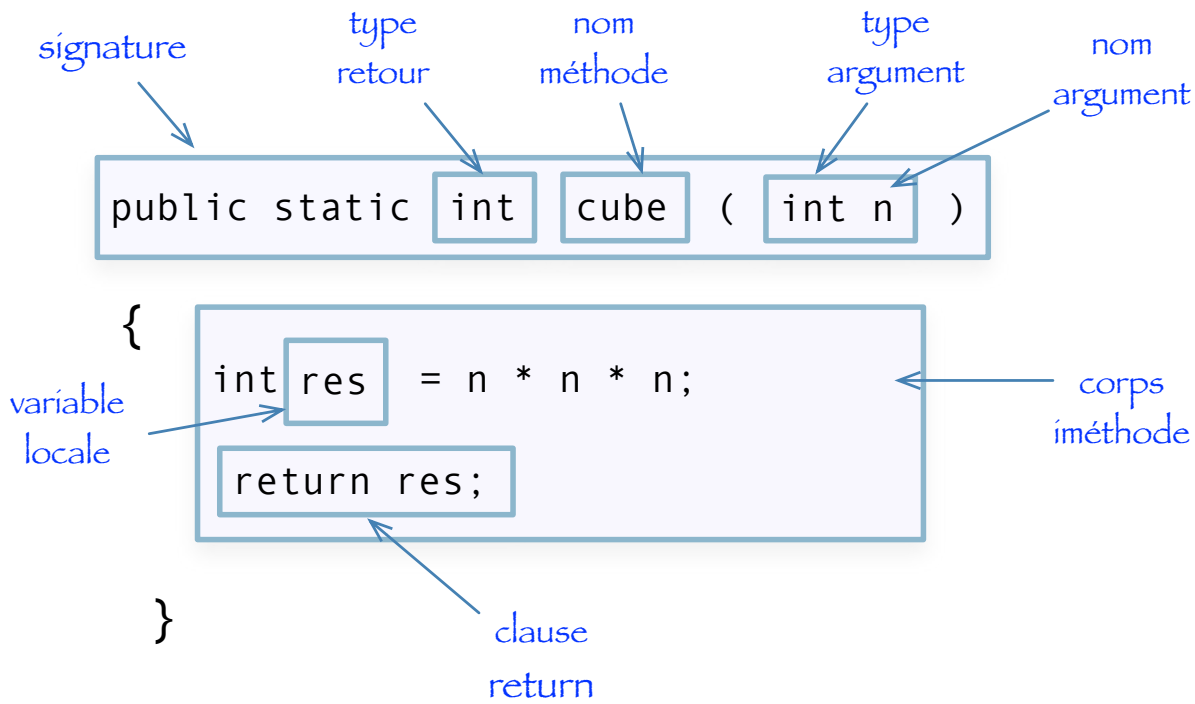
3. Les sous-programmes (méthodes)

Les sous-programmes

- appelés méthodes en Java ;
- applicables sur un objet, ou tels quels si déclarés `static` ;
- déclarés avec zéro ou plus d'arguments ;
- renvoient toujours un résultat, qui peut être vide (« void »), ou un « vrai » résultat (différent de « void ») ;
- invoquées avec tous les paramètres déclarés :
  - `()` (prononcé « void ») si la méthode ne prend pas d'argument
  - des valeurs séparées par des virgules pour chaque argument, le tout entouré de parenthèses.

Anatomie d'un sous-programme





Exemples de méthodes statiques

Méthode avec deux arguments (int) et un résultat (int) :

```
public static int somme (int x, int y) {
     return (x+y);
}
```

Méthode sans paramètre et sans résultat :

```
public static void deuxSautsLgn () {
   System.out.println(); System.out.println();
}
```

Méthodes statiques

- n'accèdent pas aux variables d'instance ;
- à la déclaration : précédées du mot-clé `static`
- à l'invocation : précédées du nom de la classe qui les contient.

Exemple

```
// retourne un boolean
static boolean estPair(int x){
   return (x%2 == 0);
}
static void maint(String [] args) {
  boolean b = estPair(3);  // appel
  System.out.println(estPair(4)); // appel
```

Procédures et fonctions

- Fonctions :
  - réalisent un calcul et retourne un résultat ;
  - ont un type de retour différent de `void` (vide) ;
  - possèdent une ou plusieurs caluse `return`
- Procédures :
  - ne retournent pas de résultat, mais réalisent des *effets* : lecture, écriture, modification de la mémoire ;
  - type de retour : `void`

Procédures et fonctions (exemples)

Fonction

```
public static int somme (int x, int y) {
      return (x+y);
}
```

Un appel produit une donnée (qu'on recupère) :
`int res = somme(2,7);` attention : pas d'affichage !

Procédure

```
public static void afficheSomme (int x, int y) {
   System.out.println("La somme est: "+ (x+y));
}
```

Rien à recupérer : `afficheSomme(2,7);`

Clause return et fonctions

Fonctions : doivent toujours finir exécution par un return

```
static int valeurAbsolue(int n){
    int res;
    if (n > 0) { res = n;
    } else if(n < 0){ return -n;
    } else { return 0;}
}
```

Si `n > 0`, pas d'instruction `return`.
⇒ erreur (compilation) :

```
> javac ValAbsFunc3.java
ValAbsFunc3.java:11: missing return statement
```

4. Types primitifs, expressions, conversions

3. Quelques types (primitifs) en Java

type	nature	constantes	opérateurs
`int`	entiers (32 bits)	1, 67, -2	`+`, `-`, `*`, `/`, `%` (modulo)
`double`	à virgule (64 bits)	0.56, 2.0 , -8.5	mêmes que pour int (sauf modulo)
`char`	caractères Unicode (16 bits)	`'a'` , `'\t'`	`(int)` conversion vers code Unicode
`boolean`	valeurs booléennes	`true`, `false`	`&&` (et), `!` (négation), `||` (ou).
`String`	chaînes de caractères (non modifiables)	`"Salut"`, `""` (chaîne vide)	`+` (concaténation)

Opérateurs de comparaison

Comparent deux *expressions* **de même type**.
Le résultat est un booléan.

==	egalité
<	plus petit
>	plus grand
>=	plus grand ou égal
<=	plus petit ou égal
!=	(différent)

Les expressions

C'est quoi ? :

- Calcul d'un résultat via l'application d'opérateurs sur des opérandes.
- opérandes : valeurs constantes, variables, ou appels de fonctions.
- valeur de l'expression : résultat d'appliquer les opérateurs sur les opérandes. On parle d'évaluation.

Expression	Valeur
7	7
7 + 6.5	13.5
"x" + "y"	"xy"
x + 3	valeur de x en mémoire + 3

Exemples d'expressions

Supposons qu'en mémoire : x=3, a=true, b =false, c='x'.

Expression	Valeur
7 > 4	true
7 >= 7	true
7 != 7	false
'a' < 'b'	true
c > 'c'	true
c == 'c'	false
!a	false
a && !b	true
x < 10	true
(x > 10) == ('b' == c)	true

Opérateurs d'incrément et décrement

Applicables sur tous les types numériques et sur `char`

```
i++;   équivaut à i=i+1;
i--;   équivaut à i=i-1;
```

Opérateurs d'affectation :

```
x -= y;   équivaut à   x=x-y;
x += y;   équivaut à   x=x+y;  (aussi avec String)
x *= y;   équivaut à   x=x*y;
x /= y;   équivaut à   x=x/y;
```

Opérateurs conditionnel

Permet d'obtenir une valeur résultat au moyen d'une conditionnelle.

```
<expr-bool> ? <expression1> : <expression2>;
```

- si <*expr-bool*> est vrai, renvoie <*expression1*>,
- sinon renvoie <*expression2*>

```
anneeSuivante = (mois == 12) ? (annee+1) : annee;
```

`anneeSuivante` est `(annee+1)` si on est en décembre,
égale à `annee` sinon.

Conversions implicites entre types

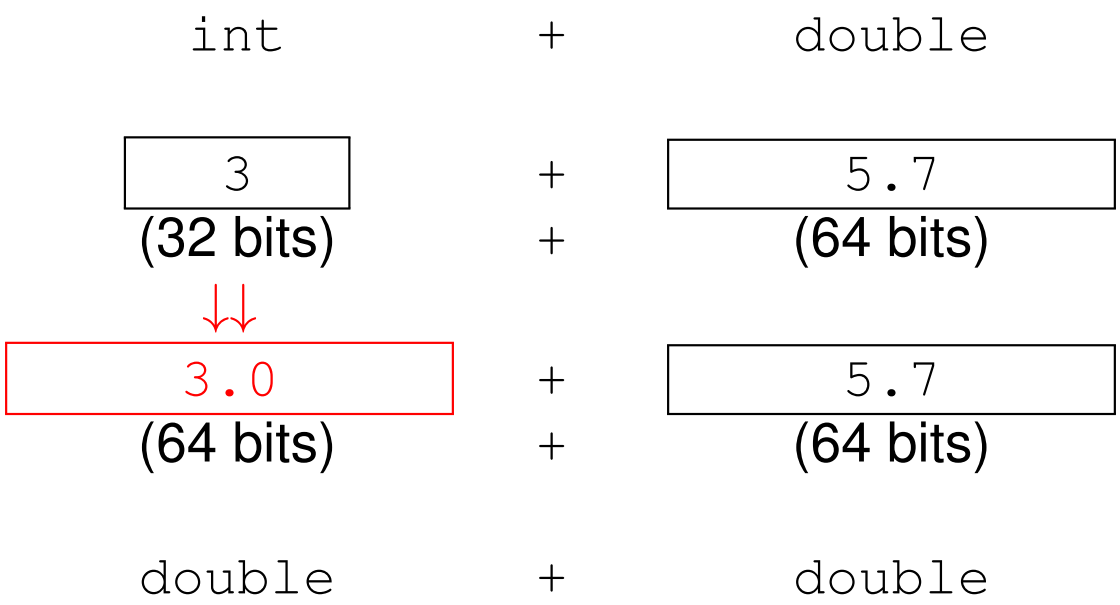
Certaines opérations requièrent un changement de *format de représentation des opérandes*.

- (3 + 4.3) $\Rightarrow$ 3 et 4.3 doivent avoir la même représentation en mémoire *avant* d'éffectuer l'addition.
- Java réalise une conversion implicite du type de 3, de `int` (32 bits) vers `double` (flottant sur 64 bits) ;
- cette conversion est *sure* (ne provoque pas d'erreur à l'exécution)
- elle est implicite : se fait sans intervention du programmeur ;

Conversions implicites sur opérations arithmétiques

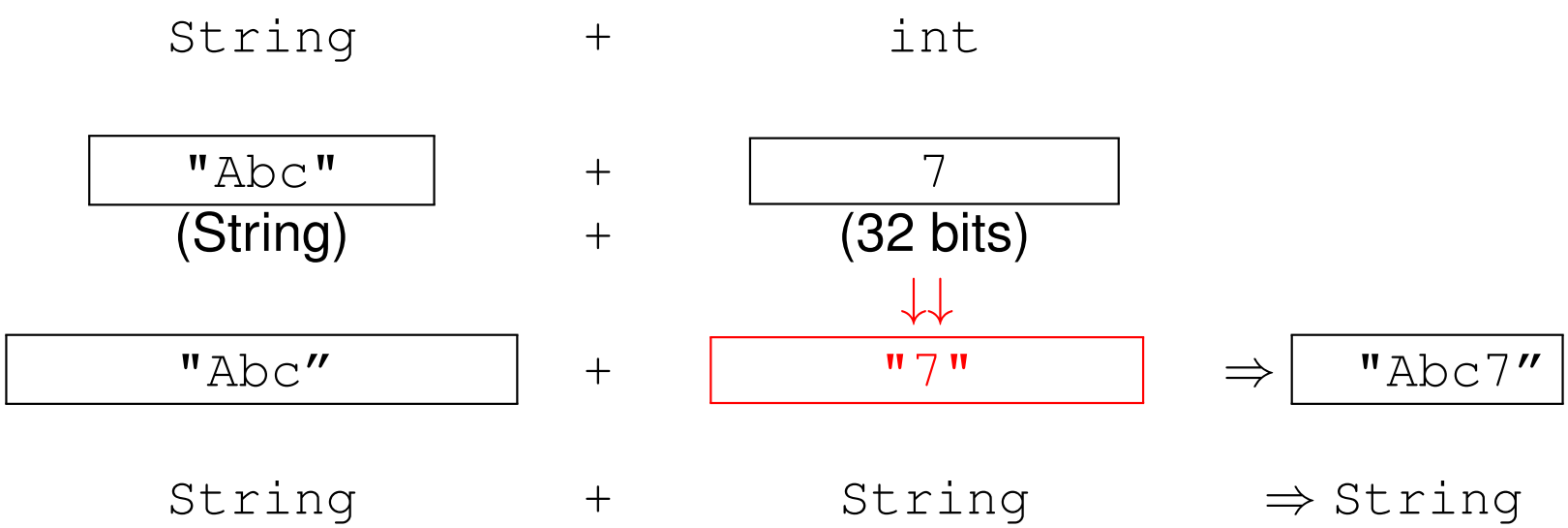
Les opérations arithmétiques en Java se font seulement si toutes les opérandes sont de même type.

Exemple :



Conversions implicites sur opérations concaténation

L'opération de concaténation entre chaînes peut provoquer la conversion d'opérandes vers String.



Cas de la division entière

La division de deux entiers donne en résultat un entier

```
5/2  donne en résultat  2
      et non pas  2.5
```

Pourquoi ?

Conversions explicites (*Cast*)

Exemple : convertir `3.67` vers un `int`

$$\texttt{(int) 3.67} \Rightarrow 3$$

produit une nouvelle valeur `3` de type `int`.

Syntaxe : `(type_cible) v`,
où `type_cible` est le type vers lequel on souhaite faire la conversion.

Attention : n'est possible que si la conversion a un sens vis-à-vis des types.
`(boolean) 5` est invalide car `boolean` n'est pas un type numérique.

5. Affectations, entrées/sorties standard

Instruction d'affectation

Syntaxe

```
<nom-variable> = <expression>;
```

Exemple :

```
int x = 3;      // declarations
int y = 7;
x = x + y;      // affectation
```

Affectations avec conversions

Affectation : peut récquerir de changer représentation de la valeur affectée.

(type variable)	=	(type valeur)	conversion nécessaire ?
`int`	=	`int`	non
`double`	=	`double`	non
`double`	=	`int`	implicite : `int` $\rightarrow$ `double`
`int`	=	`double`	explicite (cast)

```
double m = 6;   // conversion int → double
int x = m;      // erreur
```

Remarque : la conversion implicite ne change *jamais* le type des variables.

Méthodes d'entrées/sorties en Java

`System` : classe prédéfinie dans la bibliothèque Java avec méthodes d'entrées/sorties.

`Terminal` : classe écrite par les enseignants, qui regroupe les fonctions de saisie/affichage sur le clavier/écran (pas de lecture/écriture sur les fichiers), pour tous les types primitifs utilisés dans ce cours.

Le fichier source `Terminal.java`, doit se trouver dans le même répertoire que vos programmes.

Affichage avec System

- `System.out.print` : affiche une valeur primitive ou un message qui lui est passé en paramètre.
- pas de valeur résultat ⇒ l'appel est une instruction

```
System.out.print(5);                    5

System.out.print(bonjour);     contenu de bonjour

System.out.print("bonjour");         bonjour

System.out.print("bonjour "+ 5 ); bonjour 5

System.out.print(5 + 2);                7
```

Méthodes de saisie dans Terminal

- `Terminal.lireInt()`
- `Terminal.lireDouble()`
- `Terminal.lireBoolean()`
- `Terminal.lireChar()`
- `Terminal.lireString()`

```
int x;
double y;
char c;
x = Terminal.lireInt() + 4;
y = Terminal.lireDouble();
c = Terminal.lireChar();
Terminal.ecrireInt(Terminal.lireInt());
```

6. Blocs d'instructions, instruction conditionnelle

Block d'instructions : Suite d'instructions placées entre accolades `{, }`.

Utiles pour :

- donner le corps d'une méthode :

```
public static void main(){
    System.out.println ("Hello, world");
}
```

- regrouper des instructions derrière if, else, for, etc. :

```
 if (valeur < 0) {
     valeur=-valeur;
     System.out.println("Debit  : "+valeur);
 }else
     System.out.println("Credit : "+valeur);
```

Bloc : environnement local

- Un bloc est un environnement local de déclaration : toute variable n'est connue qu'à l'intérieur du bloc où elle est déclarée.
- La *vie* d'une variable déclarée dans un bloc se termine en franchissant l'accolade fermante de ce bloc.

```
for (int i=0; i<5; i=i+1){
    System.out.println(i);      // Correcte
}
System.out.println (i);         // Erreur
```

Instruction conditionnelle

Syntaxe

```
if (expre_bool) {
    suiteInstructions1
} else {
    suiteInstructions2
}
```

- on peut écrire une conditionnelle sans `else`
- ainsi qu'une conditionelles à multiples branches :

```
if (A) {
    I1
} else if (B){
    I2
} else if (C) ...
```

5. Boucles

Boucle `while`

Syntaxe

```
while (<expr-bool>)  {
    suiteInstructions
}
```

« Tant que expr-bool est vrai, faire suiteInstructions »

1. *expr-bool* est la **condition** testée avant chaque tour de boucle
2. le bloc d'instructions de la boucle est son **corps** ;

```
int i=1;
while (i<=4)  {
   System.out.println("****");
   i=i+1;
}
```

La boucle for

Syntaxe

```
for (decl-et-init; condition; pas-avancement)  {
    instructions
}
```

C'est un raccourci pour la boucle `while` suivante :

```
{
  decl-et-init
  while ( condition ) {
      instructions
      pas-avancement
  }
}
```

Déroulement d'une boucle for

```
for (int i=1;i<=4;i=i+1){
    System.out.println("****");
}
```

1. **déclaration et initialisation** (`int i=1;`) : exécuté la 1ère fois ; puis
2. test **condition** (`i<=4;`) :
   - si fausse, arrêt ;
   - si vraie, exécuter instructions du corps
3. exécuter **pas-avancement** (`i=i+1;`)
4. on teste une nouvelle fois **condition** et on recommence à (2)

Boucle `do-while`

Il s'agit d'une boucle `while` où les instructions du corps sont exécutées avant de tester la condition de la boucle.

```
do
  {
     suiteInstructions
  }
while (c);
```

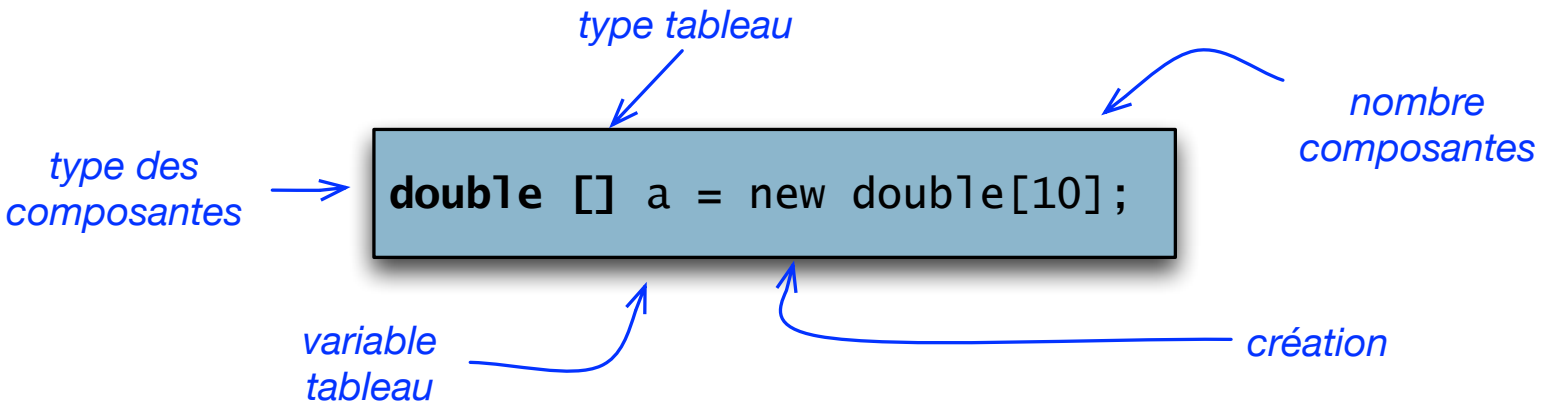
où *c* est une expression booléenne.
On lit : *"faire suiteInstructions, tant que c est vrai"*

1. *suiteInstructions* est exécuté,
2. *c* est évaluée à la fin de chaque itération : s'il est vrai, le contrôle revient à *suiteInstructions* (point 1).
3. Si *c* est faux, le contrôle du programme passe à l'instruction immédiatement après la boucle.

7. Tableaux

Les tableaux (`array`)

1. Déclaration :
   - `double [] t` déclare un tableau de double.
2. Création en mémoire :
   - opération `new` avec nombre + type de composantes à allouer en mémoire
   - `new double [10]` 10 composantes crées en mémoire
3. Initialiser les valeurs des composantes (par défaut ou explicitement).



En détail : création des composantes d'un tableau

Syntaxe :

```
new T[n];   // n (nbe composantes)
            // T (type composantes)
```

En mémoire `new T[n]` se traduit par :

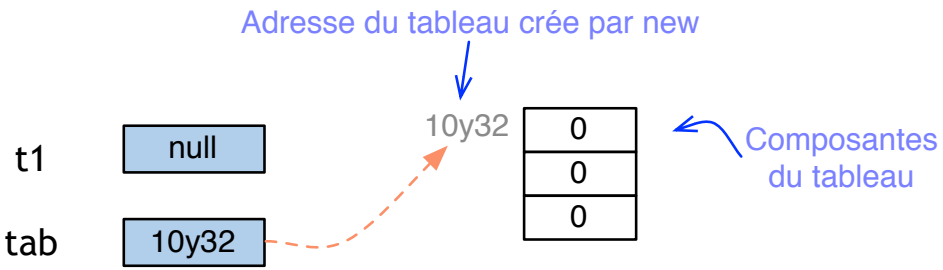
1. nouvel block mémoire reservé pour n composantes de type T.
2. initialisation par défaut de ces composantes (0 pour les types numériques, false pour bool, etc.)

Exemple de création en mémoire

```
int [] t1;
int [] tab;          // declaration
tab = new int[3]; // creation + affectation
```

Après l'affectation `tab = new int[3]`

- `tab` : contient l'adresse d'un block mémoire de 3 composantes `int` initialisées à 0.



Valeurs d'initialisation par défaut via `new`

Les valeurs par défaut données par `new` (selon le type des composantes) :

- composantes `boolean` ⇒ initialisées à `false`.
- composantes numériques ⇒ initialisées à 0.
- composantes `char` ⇒ initialisées au caractère nul (`'\0'`)
- composantes de type *référence* ⇒ initialisées à `null` (pointeur nul).

3 façons de création + initialisation

- initialiser avec valeurs par défaut, via `new`

```
int tab=new int [3];
```

- initialiser en donnant une liste de valeurs :

```
int [] tab = {1,9,2};
```

- ou, par création + affectation de chaque composante :

```
int [] tab = new int [3];
tab[0] = 1;
tab[1] = 9;
tab[2] = 2;
```

Mémoire : que contient une variable tableau ?

```
int []  t;        // declaration tableau de int
```

- si t n'est pas affecté :
  - t contient la valeur `null` $\Rightarrow$ ne possède aucune composante ;
  - tout accès `t[i]` $\Rightarrow$ erreur fatale (`NullPointerException`)
- si t est affectée par :
  - une valeur de type tableau (de int),
  - ou par une opération de création de composantes (`new`) :
    - tout accès `t[i]` (dans les bornes de t) réussit
    - t contient l'adresse mémoire du début de la zone mémoire où sont stockées ses composantes.

Taille d'un tableau

Taille du tableau `t`

C'est le nombre de composantes de `t`.

- donné par : `t.lenght`
- Indices de `t` : entre 0 et `t.length-1`.

Attention : la taille d'un tableau peut-être 0.

```
int [] t = new int[3];                  // taille 3
System.out.println(t.length);           // affiche 3
double [] m = new double[0];               // taille 0
System.out.println(m.length);           // affiche 0
```

Schéma typique de parcours (il y en d'autres !)

Pour travailler avec un tableau : utiliser des boucles !

Boucle de parcours du tableau t

Permet de « visiter » les composantes en faisant varier leur indice.
Faire varier une variable *i* qui servira d'indice :

- *i* varie dans l'intervalle [0..`t.lenght` − 1].
- traiter chaque composante `t[i]`

```
for (int i=0; i< t.length; i++){
      actions sur t[i]
}
```

Les boucles `for` sont en général bien adaptées.

Fonction de calcul moyenne d'un tableau

Problème : calculer la moyenne d'un tableau de notes
Solution : fonction de parcours+calcul
param : tableau de notes
retourne : moyenne calculée

```
  /* Calcule la moyenne de composantes dans t
   */
static double moyenneTab (double [] t) {
    double somme = 0;
    for (int i=0; i< t.length; i++) {
          somme = somme + t[i];
    }
    return somme/t.length;
}
```

Procédure affichage d'un tableau

Problème : afficher un tableau de notes
Solution : boucle parcours+affichage (pas de calcul)
param : tableau de notes
pas de valeur retour ⇒ procédure

```
 /* Affiche composantes d'un tableau de notes
  */
static void afficheTabNotes (double [] t) {
   for (int i=0; i<t.length; i++)  {
       System.out.println("note␣"+ (i+1)+":␣"+ t[i]);
   }
}
```

Remarque : une procédure/fonction ne lit pas ses entrées ⇒ les prend en paramètre ;

Fonction création + lecture tableau

Problème : créer un tableu de taille n et l'intialiser par lecture
Solution : création puis parcours+lecture
param : n (taille tableau)
valeur retour : tableau crée et intialisé

```
/* Creation et lecture d'un tableau taille n
 * Retourne: tableau avec composantes lu
 */
static double [] lireTab (int n) {
    double [] t = new double[n];
    for (int i=0; i<= t.length -1; i++)  {
         System.out.print("Une note? ");
         t[i] = Terminal.lireDouble();
    }
    return t;
}
```

3. Représentation des données en mémoire

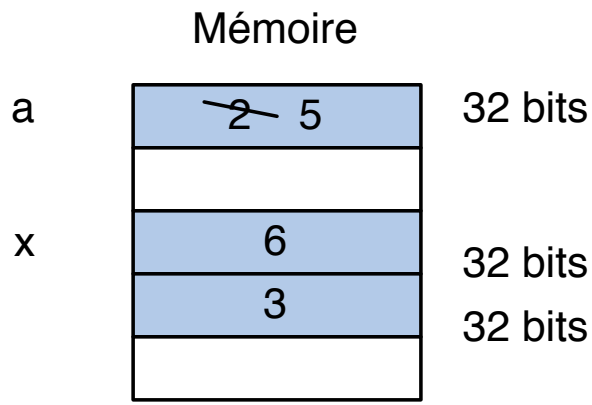
Deux catégories de données en Java

Représentées différemment en mémoire :

- données de type primitif : valeurs élémentaires
  - int, boolean, char, double, etc.
  - l'emplacement mémoire de la variable contient sa valeur
- données de type référence (pointeurs) : valeurs composites, formées (possiblement) de plusieurs données plus élémentaires
  - tableaux, String, objets.
  - l'emplacement mémoire de la variable ne contient pas sa valeur mais une adresse vers une autre zone mémoire où se trouve cette valeur.

Emplacement de stockage : types primitifs

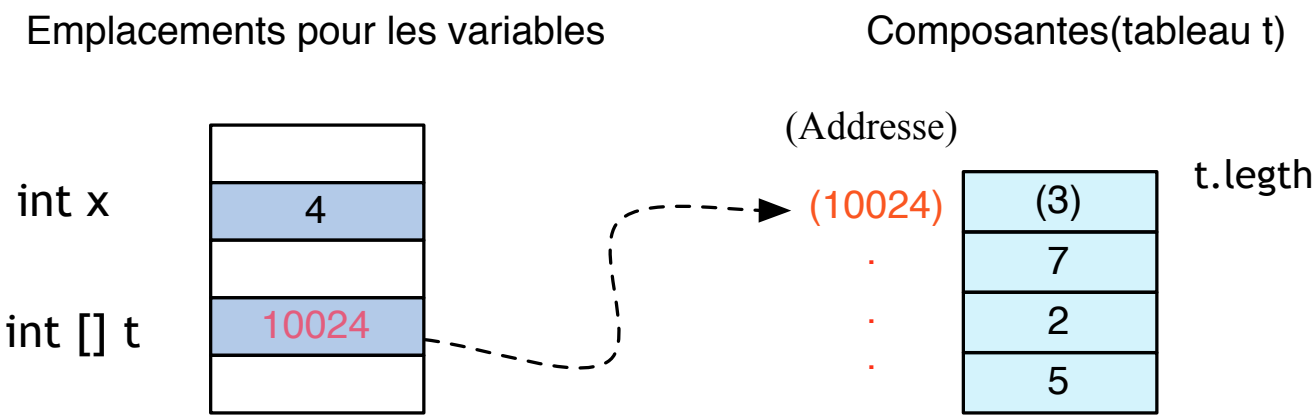
- taille : variable selon le type.
  - `int` ⇒ 32 bits
  - `double` ⇒ 64 bits
  - `char` ⇒ 16 bits
  - ...
- contenu stocké : la donnée *en place*, un entier, un double, etc.




```
public static void main(….) {
   int  a = 2;
   double x = 6.3;
   a = a+2;
   ….
}
```

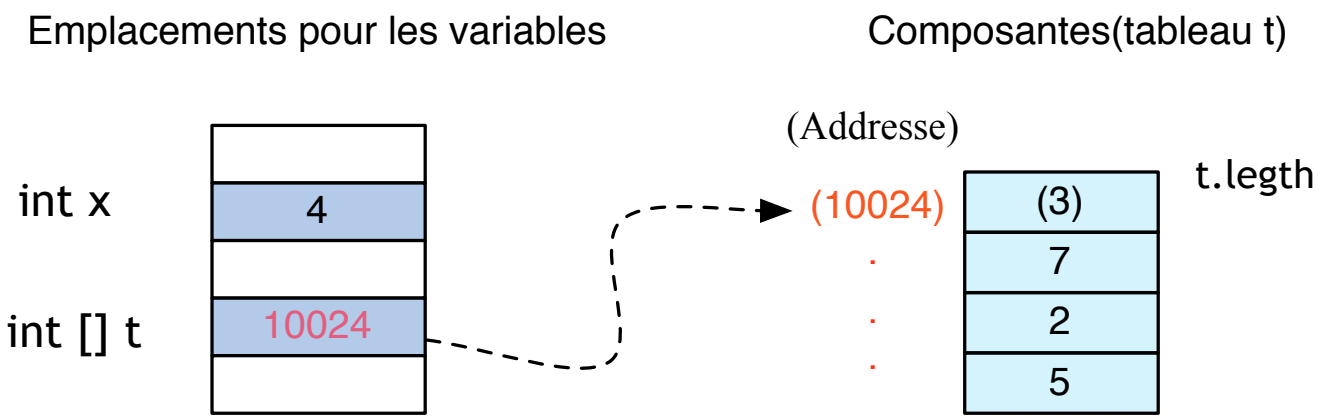
Emplacement de stockage : types référence (pointeur)

- donnée de type référence $\Rightarrow$ toujours composite (plusieurs) ;
- emplacement d'une variable de ce type $\Rightarrow$ ne contient pas ses données ;
- il contient :
  - adresse mémoire d'un espace ailleurs pour les données.



Exemple

```
int x = 4;
int[] t = {7,2,5};
```



- `x` est de type primitif : elle contient **directement** sa valeur.
- `t` est de type référence : elle ne contient pas le tableau, mais l'adresse où se trouvent ses composantes.
- `t` est un *pointeur ou référence*.

Exemples de données de types référence

- Une variable de type String, ne contient pas la chaîne elle-même, mais l'adresse mémoire où se trouve la chaîne.
- La variable `int [] t = {4, 6, 3}` ne contient pas le tableau, mais l'adresse où se trouvent ses composantes.
- Chacune de ces variables est un *pointeur ou référence*.

Affectation entre variables de type tableau

Ce code est-il correct ?

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;    // affectation
```

Affectation entre deux variables de type pointeur

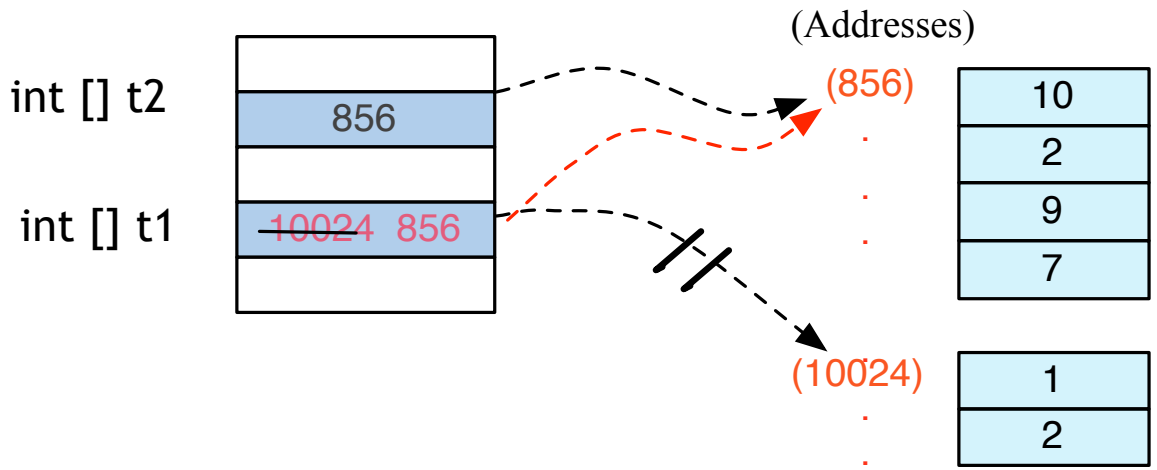
C'est possible, si leurs types sont compatibles. Son comportement :

- Copie du contenu d'une variable dans l'autre.
- Ce contenu est **une adresse**.

Vue de la mémoire pour affectation entre tableaux

```
int [] t1 = {1,2};
int [] t2 = {10,2, 9, 7};
t1 = t2;
```

Mémoire : contenu des variables *Mémoire : les composantes*



On recopie le contenu de t2 (l'adresse 856) dans t1.

Affectation : la taille des tableaux n'est pas importante

Les tableaux d'une affectation peuvent avoir des longueurs différentes

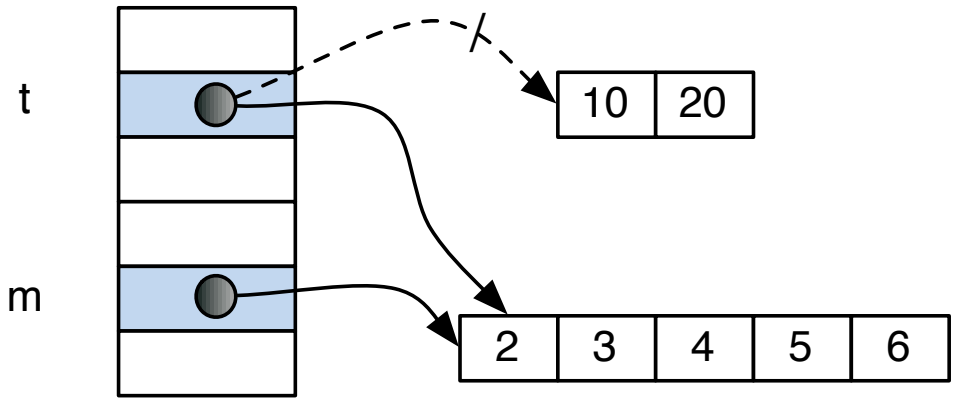
```
int [] t = {10, 20};    // taille 2
int [] m = {2,3,4,5,6};
System.out.println("Longueur de t = "+ t.length);
t = m;                  // taille 5
System.out.print("Nouvelle longueur t = "+ t.length);
```

```
Longueur de t = 2
Nouvelle longueur de t = 5
```

Affectation : la taille des tableaux n'est pas importante

Le tableau affecté « hérite » des caractéristiques du tableau à droite du « = ».

```
int [] t = {10, 20}; // taille 2
int [] m = {2,3,4,5,6};
t = m;                  // taille 5
```



Affectation entre pointeurs : partage de variables

Après affectation, t1 et t2 pointent vers le même emplacement mémoire :

```
int [] t1 = {1,2};
int [] t2 = {10,2, 9, 7};
t1 = t2;
t1[0] = 50;
Terminal.ecrireInt(t2[0]); // affiche??
```

- le changement d'une case de l'un modifie cette même case pour l'autre.

Partage, aliasing

On dit des variables t1 et t2 qu'elles **partagent** leurs composantes, ou qu'elles sont des **alias** pour celles-ci.

Comparaison de valeurs de type référence

L'opérateur == compare les bits contenus dans les variables.

S'il s'agit d'adresses, cela teste si les adresses sont égales, c.a.d. si les variables référencent le même objet en mémoire.

```
int [] t1 =  {1,2};
int [] t2 =  {10,2, 9, 7};
int [] t3 =  {1,2};
t2 = t1;
if (t1==t2){ System.out.println("t1==t2"); }
if (t1==t3){ System.out.println("t1==t3"); }
else {System.out.println("t1!=t3");  }
```

Affichages :

```
t1==t2
t1!=t3
```

Tableaux de plusieurs dimensions

En Java, un tableau de n dimensions et composantes de type `TyBase` est déclaré par :

```
TyBase [] []...[]  tab;    // n fois le symbole []
```

Chaque occurrence du symbole [] permet d'obtenir une dimension supplémentaire :

```
int [] t;            // 1 dimension
int [] [] m;         // 2 dimensions
char [] [] [] p;     // 3 dimensions
```

Création et initialisation avec `new`

- création avec `new`, en donnant la taille de chacune des dimensions,
- toutes les composantes sont initialisées avec des valeurs par défaut.

```
int [][] T=new int [3][4]; //creation avec 3 lignes
                           //et 4 colonnes
T[1][2]= 7;     // modification composante (1,2)
```

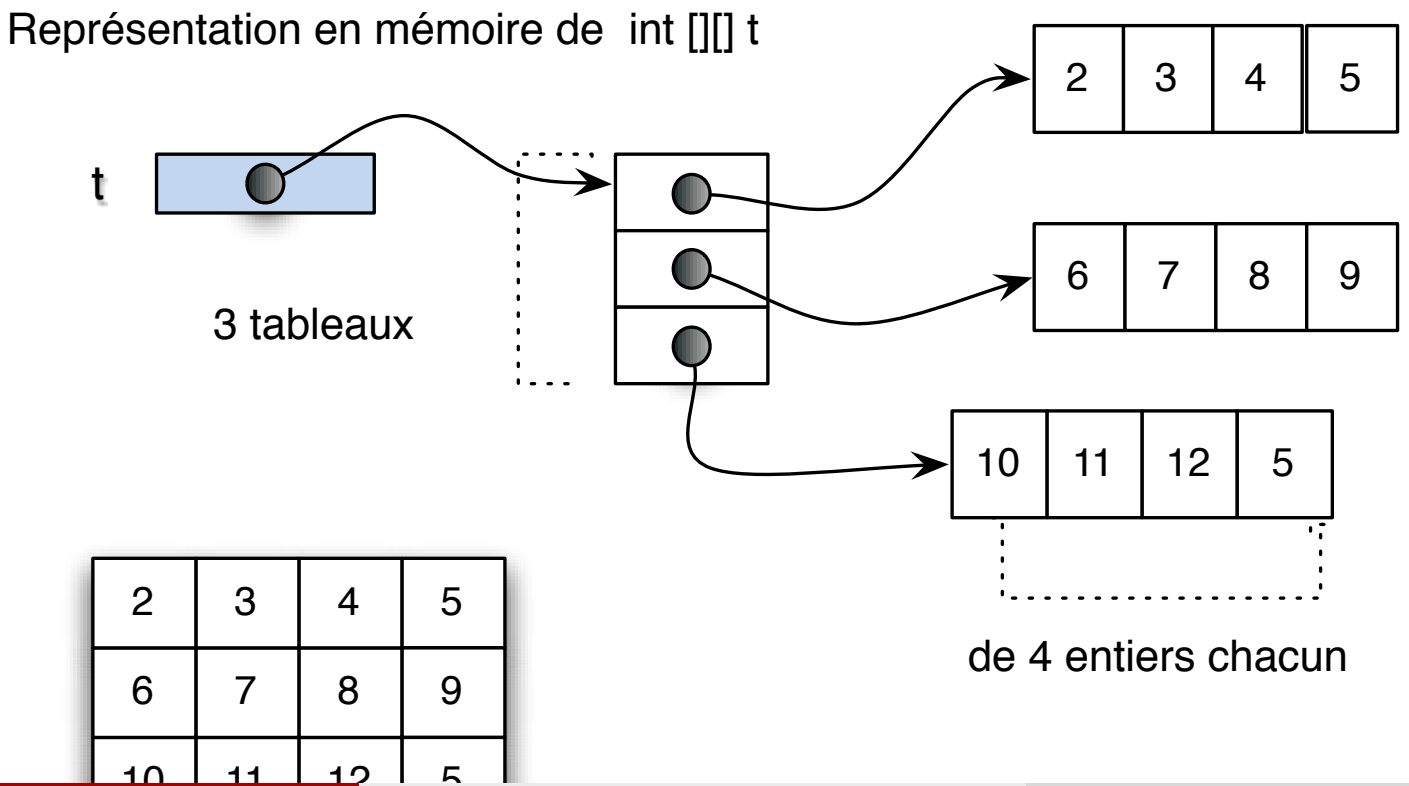
T =

	0	1	2	3
0	0	0	0	0
1	0	0	7	0
2	0	0	0	0

T[0][0]	T[0][1]	T[0][2]	T[0][3]
T[1][0]	T[1][1]	T[1][2]	T[1][3]
T[2][0]	T[2][1]	T[2][2]	T[2][3]

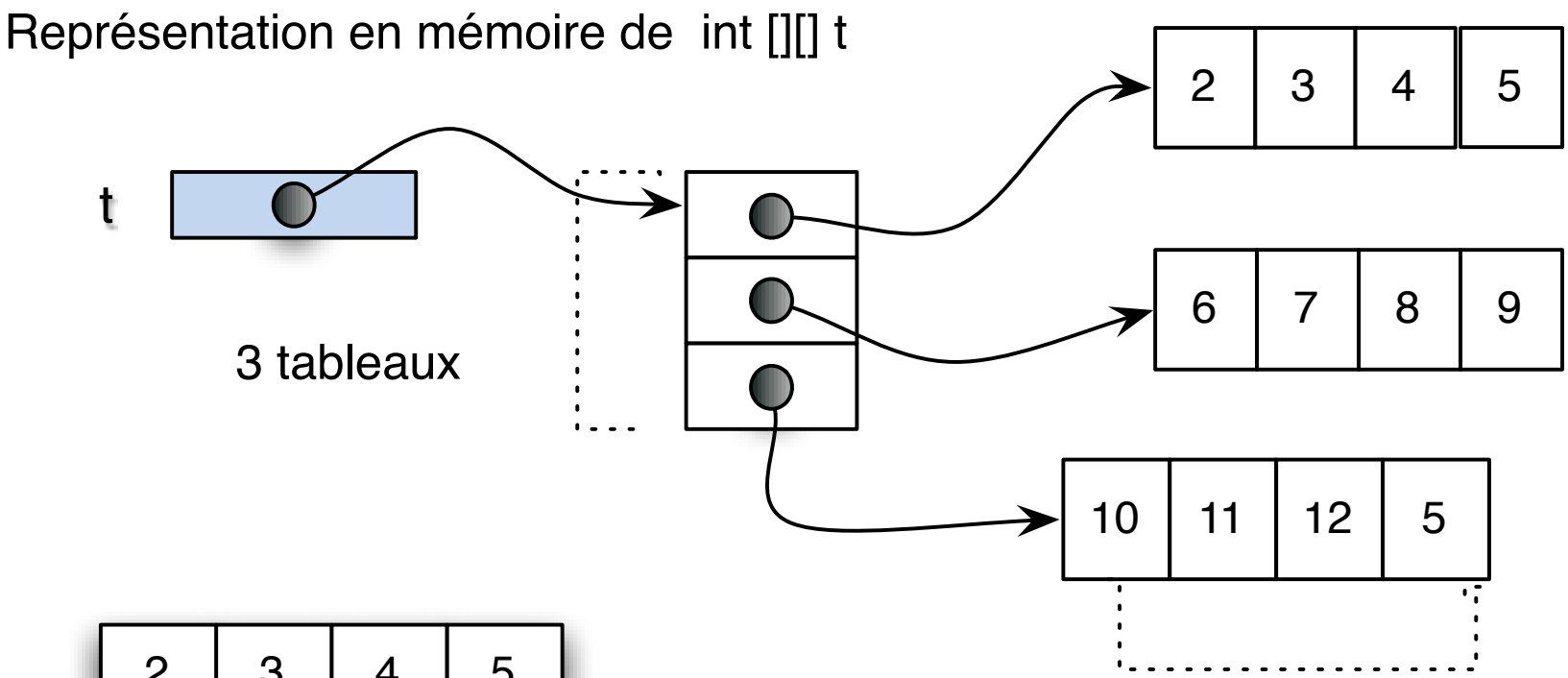
Représentation en mémoire des matrices

- En Java, une matrice est en réalité **un tableau de tableaux**.
- Exemple : `int[][] t = new int[3][4]` est formé de :
  - 3 tableaux de `int`,
  - où chacun de ces 3 tableaux a 4 composantes de type `int`.



Représentation en mémoire des matrices

Souvent il nous suffira de penser aux matrices comme des grilles.



2	3	4	5
6	7	8	9
10	11	12	5

.... que nous verrons comme cette grille

Longueur d'une dimension

Si `t` est une matrice :

- `t.length` : donne la longueur de la première dimension (nombre de lignes du tableau).
- `t[i].length` : donne la longueur de la ligne i de t, autrement dit, le nombre de colonnes de cette ligne.

```
int [][] t = new int [3][4];          //  3 lignes, 5 col
Terminal.ecrireIntln(t.length);       // affiche 3
Terminal.ecrireIntln(t[1].length);    // affiche 4
```

Exécution de sous-programmes et mémoire

Exécution $\Rightarrow$ *requiert mémoire (stockage variables)*.

Exécution méthode $\Rightarrow$ mémoire propre et locale :

- stocker variables de la méthode :
  environnement local : paramètres + décl. locales
- inaccessible en dehors du sous-programme ;
- chaque appel $\Rightarrow$ création nouvelle mémoire ;
- active pendant une exécution :
  - fin d'exécution $\Rightarrow$ mémoire méthode "disparaît".

Variables d'un sous-programme

Deux sortes :

- les paramètres,
- les variables déclarées localement,

Elles sont toutes locales au sous-programme :

- un sous-programme ne peut utiliser que ses variables locales ;
- toute autre variable mentionnée sera considérée inconnue.

Variables locales méthode m $\approx$ environnment local pour m

Retour sur exécution d'un appel

La méthode `main` appelle la méthode `plusUn` :

```
static int plusUn(int x){
  int res = x+1; ...
}
public static void main (String [] args){
  int x = 3;
  int y = plusUn(x*2); // <--- appel
  System.out.println("Resultat= "+y);
}
```

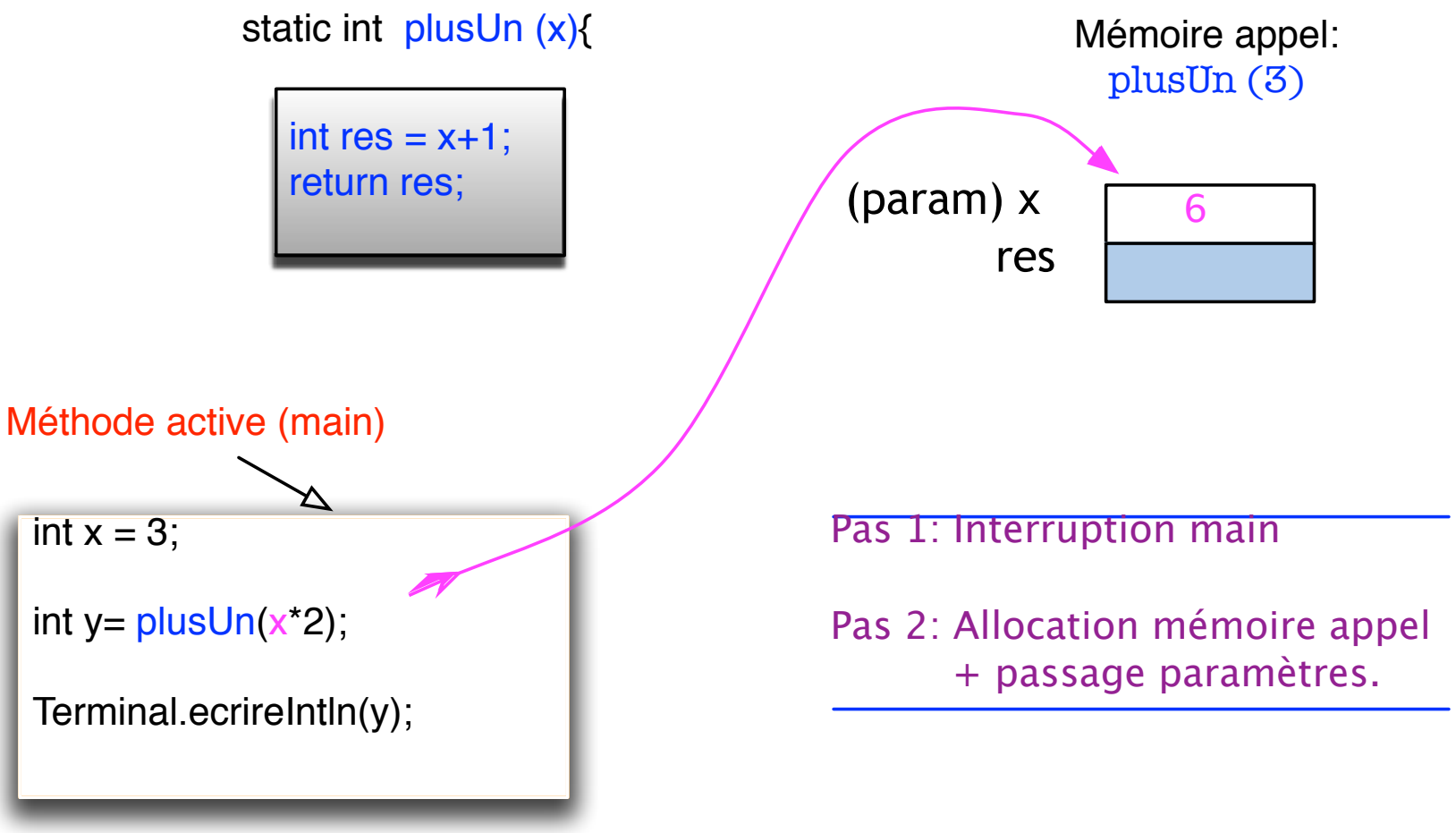
Avec quels arguments se fait l'appel ? ⇒ plusUn(6)

Retour sur exécution d'un appel (2)

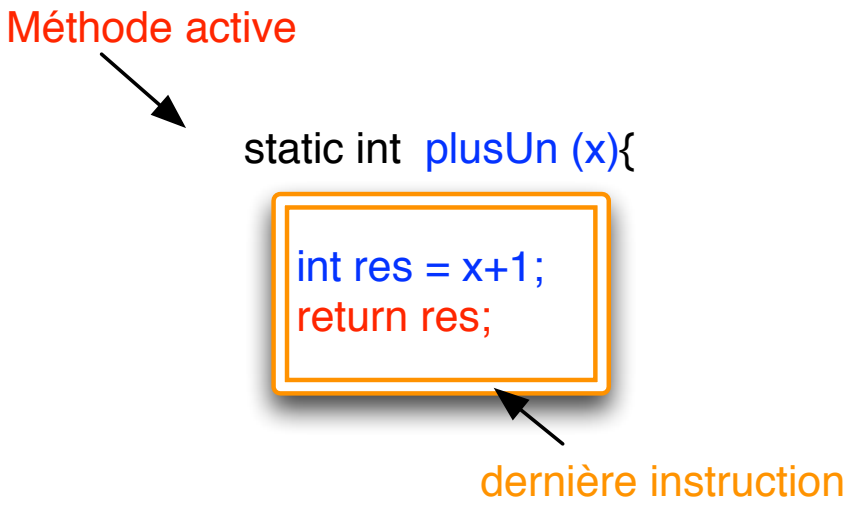
Appel plusUn(3) :

1. Interruption méthode appelante (main) ;
2. Allocation mémoire locale `plusUn` (2 variables) + passage des entrées (paramètres) :
   1. recopie valeurs paramètres : $3 \mapsto x$
3. Exécution `plusUn` ;
4. Fin exécution : des-activation mémoire + retour (avec résultat) vers la méthode appelante (main) ;
5. Reprise exécution méthode appelante (main)

Exécution appel plusUn



Exécution appel plusUn (2)



Mémoire appel:
plusUn (3)

(param) x	6
res	7

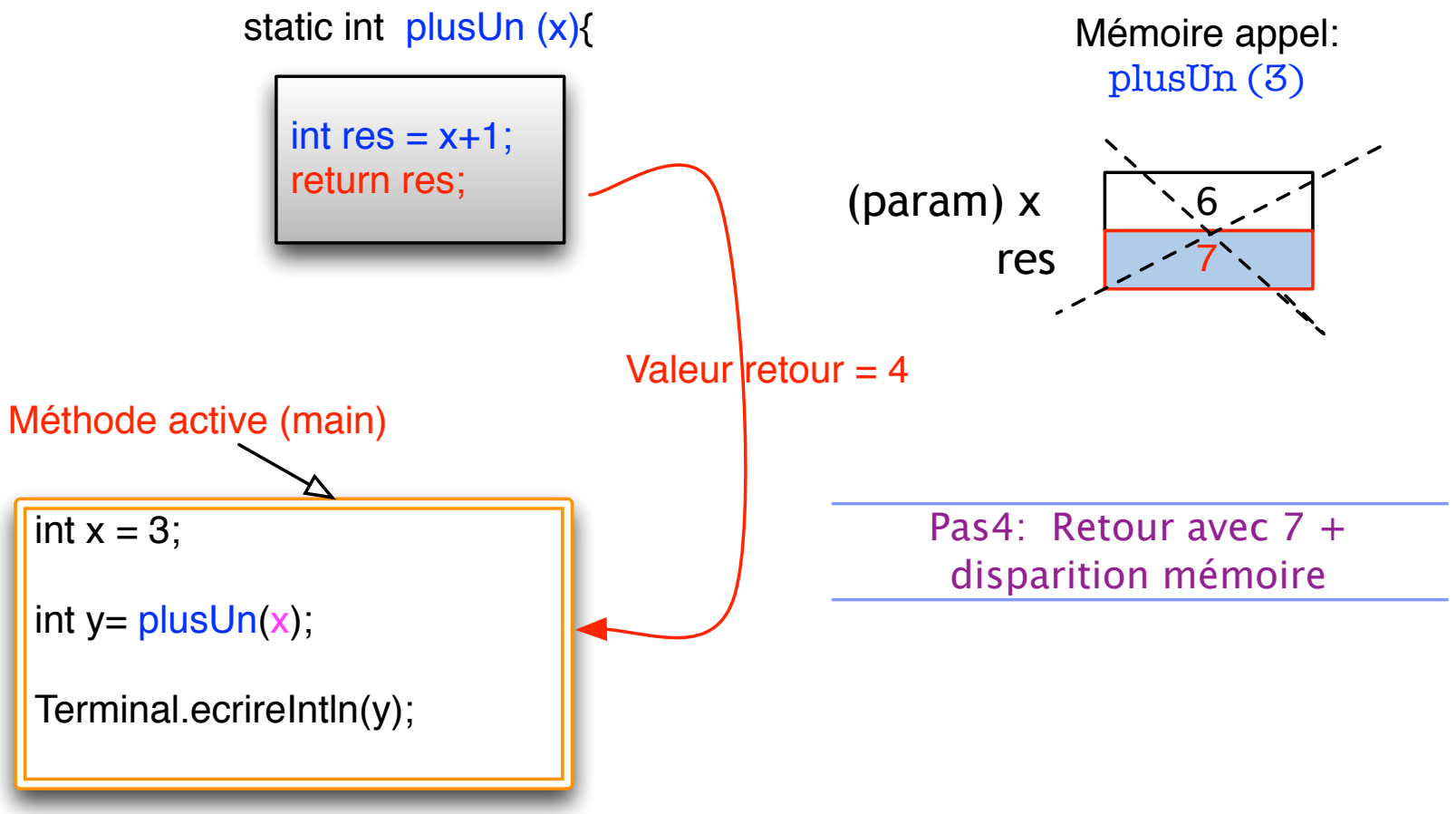
```
int x = 3;

int y= plusUn(x * 2);

Terminal.ecrireIntln(y);
```

Pas 3: Exécution plusUn

Exécution appel plusUn (3)



Retour sur exécution d'un appel (3)

La méthode `main` appelle la méthode `M` :

```
static int plusUn(int x){
  int res = ....
}
public static void main (String [] args){
  int x = 3;
  int y = plusUn(x); // <--- appel
  System.out.println("Resultat=␣"+y);
}
```

- la méthode main possède-t-elle une mémoire locale ?
- quand est-elle active ?
- où est elle passée pendant l'exécution de `plusUn` ?

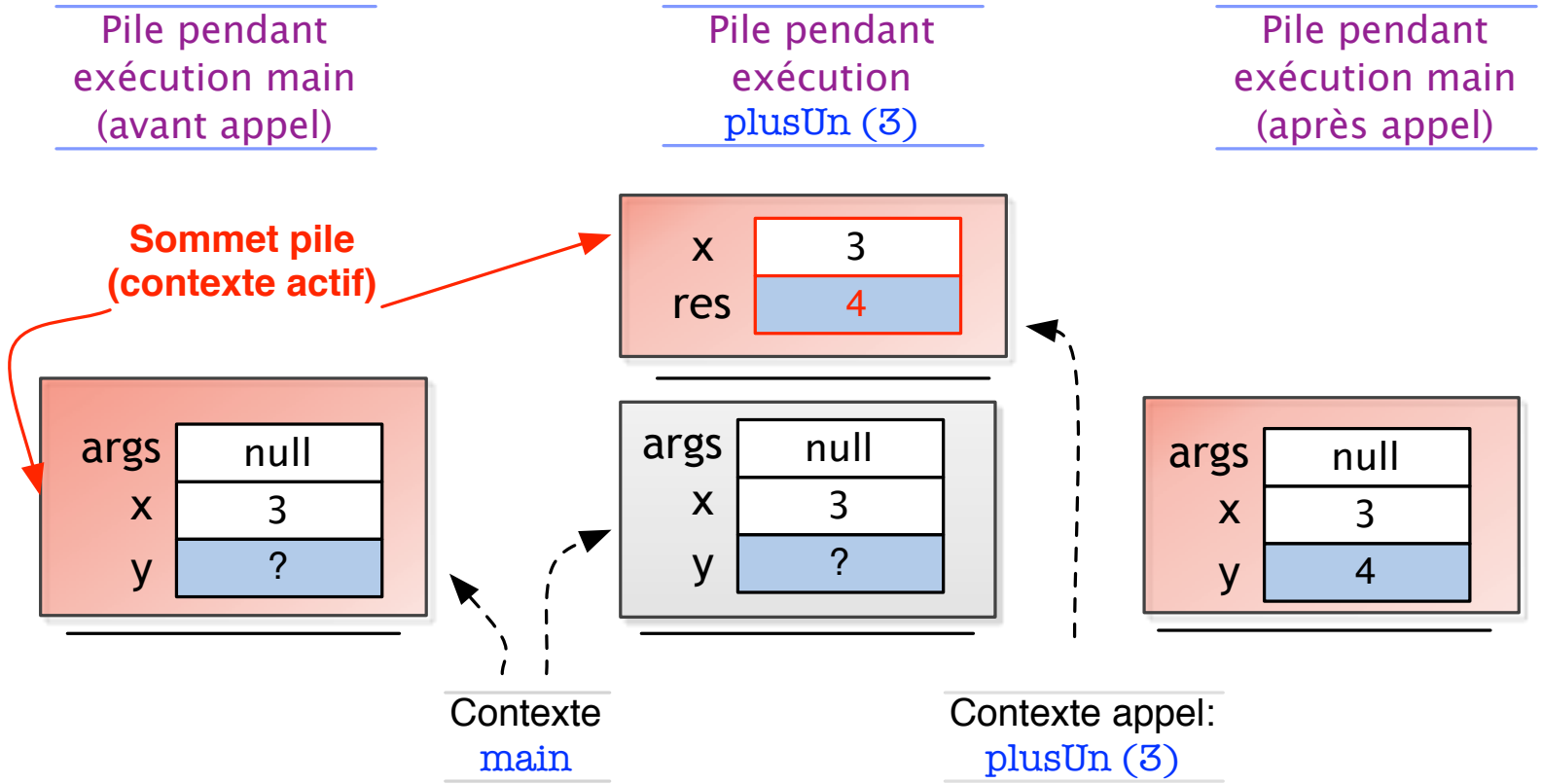
Pile d'exécution

Empilement de mémoires locales correspondant à tous les appels non encore terminés.

- Haut de la pile : mémoire de la méthode active (qui s'exécute) ;
- chaque nouvel appel vers une méthode m, met en place son environnement (mémoire locale) en haut de la pile d'exécution ;
- juste au dessous : mémoire de la méthode appellant m ;
- dès que m est terminée, sa mémoire sort de la pile.
- Se retrouve en haut de la pile $\Rightarrow$ mémoire méthode appelante, qui devient active.

Pile d'exécution (2)

Empilement de mémoires locales correspondant à tous les appels non encore terminés.



Procédure avec argument de type référence

Que se passe-t-il si :

- une procédure modifie explicitement son argument ?
- cela change la valeur de la variable passée par la méthode appelante ? (c.a.d., la mémoire de la méthode appelante ?)
- différence selon que l'argument est de type primitif ou référence ?

Retour sur le passage de paramètres

Le passage de paramètres en Java se fait « par valeur » :

⇒ lors d'un appel `m(x)`, on passe à m :
la valeur contenue dans la variable x.

- x de type primitif : on passe sa valeur, entier, booléan, etc.
- x de type référence : on passe sa valeur, qui est une adresse.

Que peut faire m de plus ou de moins selon le cas ?

Passage avec argument de type référence (tableau)

```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
   int [] x = {1,2,3};
   m(x);
   for (int i=0; i< x.length; i++){
      System.out.print(x[i] + "␣");
   }
}
```

Qu'affiche ce programme ?

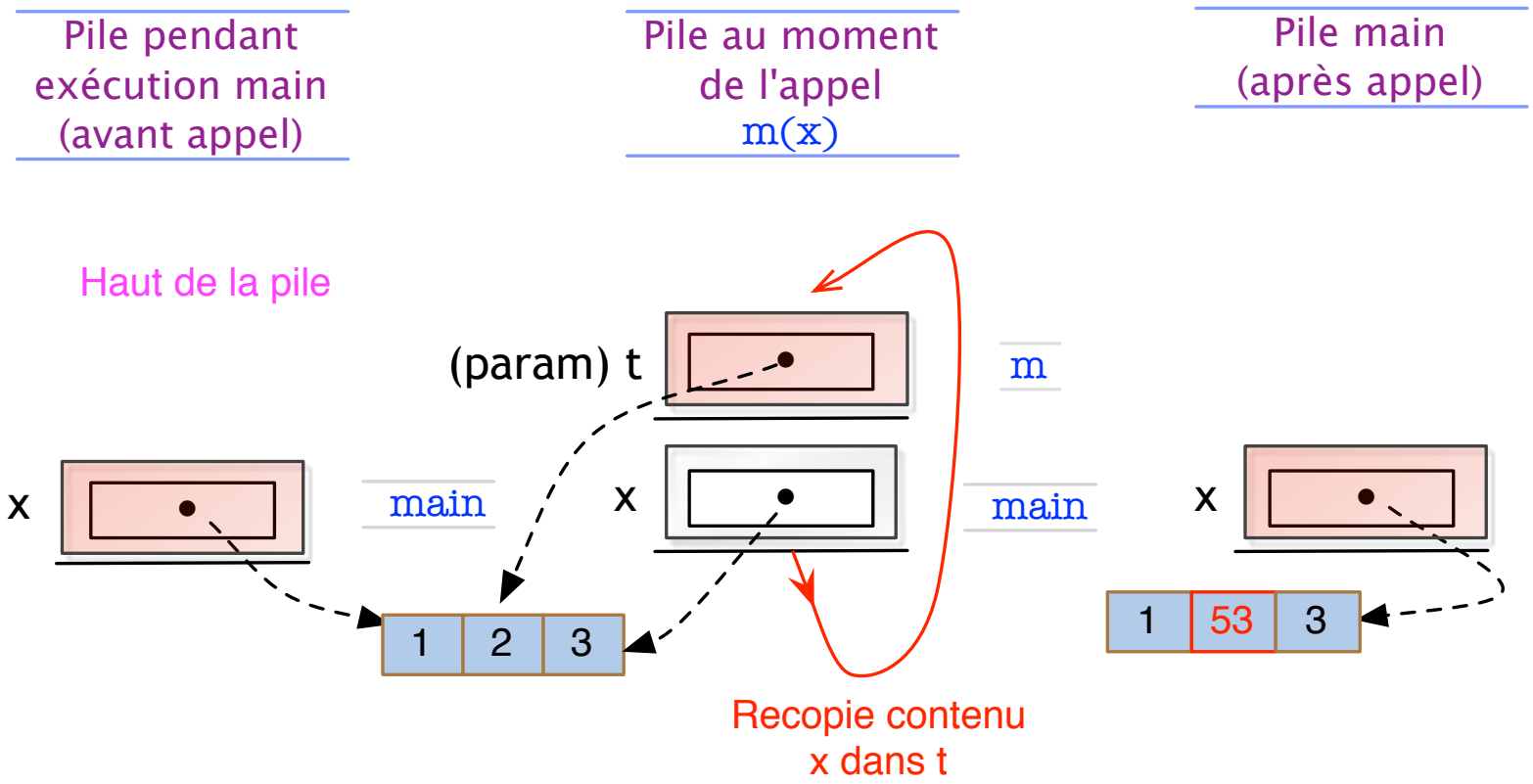
Passage avec type référence (2)

```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
   int [] x = {1,2,3};
   m(x);
   for (int i=0; i< x.length; i++){
      System.out.print(x[i] + "␣");
   }
}
```

- le paramètre de `m` est un tableau (type référence) ;
- `p` appelle `m(x)` ⇒
  - x est une variable locale à p,
  - au retour, m a changé la valeur de x ?

Pile d'exécution + paramètres type référence

Pendant l'exécution de m, x et t pointent sur le même tableau ⇒ m peut changer les composantes de x.



Règles de style en Java

- Java n'impose pas des règles de formattage des programmes.
- On peut écrire tout un programme sur une seule ligne : c'est un entrée valide pour le compilateur. Mais ce n'est pas très lisible pour les humains.
- Certaines conventions d'écriture sont devenues des «standards» : elles visent à améliorer la lisibilité des programmes par les programmeurs eux-mêmes.

Règles de style (2)

- Les noms des classes débutent par une majuscule (`Terminal`, `Conversion`).
- Les noms des variables et méthodes débutent par une minuscule (`main`, `x`, `euros`, `lireInt`).
- Les noms composés se font par adjonction de plusieurs mots, chaque mot débutant par une majuscule (`CompteBancaire`, `lireInt`).
- Chaque variable initialisée est déclarée (toute seule) sur une ligne.

Règles de style (3)

- Les variable essentielles au programme sont déclarées en début de la méthode main. Les variables auxiliaires, juste au moment où elles sont nécessaires.

- Chaque nouvelle structure est décalée de 2 ou 3 caractères à droite par rapport à la structure qui la contient. On parle **d'indentation**.

```
public class Conversion {
  public static void main (String[] args) {
```

Règles de style (4)

- Les instructions son indentées à droite par rapport au bloc qui les contient.

```
public static void main (String[] args) {
   double euros;
   double francs;
   System.out.println("Somme␣en␣euros?␣");
   euros = Terminal.lireDouble();
   ...
```

- Chaque instruction est écrite sur une ligne. Toutes les instructions d'un bloc sont alignées sur la même colonne.

Règles de style (5)

- L'accolade fermante d'un bloc est alignée sur la même colonne que le début de la structure qu'elle délimite.

```
public class Conversion {
  public static void main (String[] args) {
    ....
    francs = euros * 6.559;
    System.out.println("Conversion= "+ francs);
  }
}
```