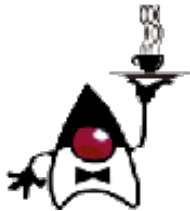


Université de Corse



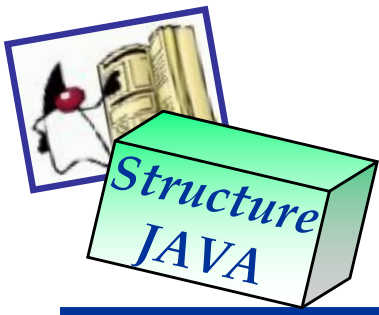
# Structure du langage Java

*(notions de base)*



Marie-Laure NIVET - Evelyne VITTORI  
[vittori@univ-corse.fr](mailto:vittori@univ-corse.fr)

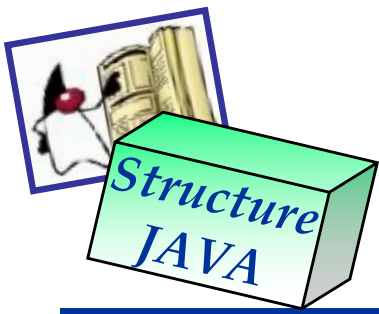




# Structure du langage Java

---

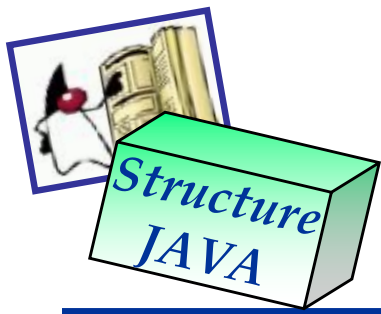
- La syntaxe du langage
  - Les commentaires
  - Les identificateurs, les variables
- Les types primitifs
- Les opérateurs
- Les expressions
- Les structures de contrôle
- La portée des variables



# Les commentaires

---

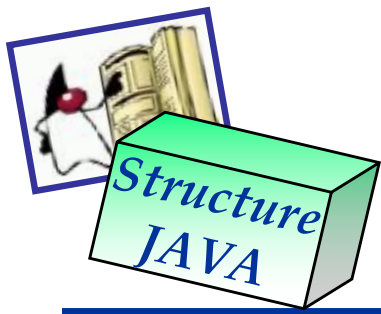
- **// commentaire jusqu'à la fin de la ligne**
  - `// ceci va jusqu'à la fin de la ligne`
  - `int i = 1; // celui ci aussi`
- **/\* ... \*/ blocs de commentaire**
  - `/* commentaire occupant  
deux lignes ou plus... */`
- **/\*\* ... \*/ commentaires de documentation**
  - `javadoc.exe`



# Les séparateurs

---

- Une instruction se termine par un point virgule `;`
- Elle peut tenir sur plus d'une ligne
  - `total=ix+iy+iz; //une instruction`
  - `total=ix+`  
`iy+iz; //2 lignes, 1 instruction`



# Les séparateurs

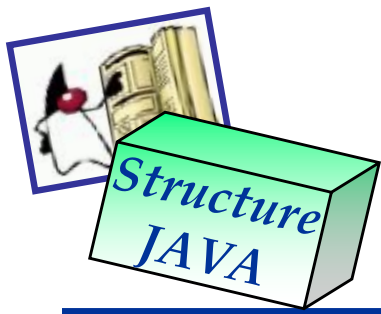
---

- Un bloc regroupe un ensemble d'instruction, il est délimité par des accolades '{ }'

- ```
{ // un bloc
    ix = iy+1; // une instruction
    iz = 25;   // la deuxième du bloc
}
```

- Ils peuvent s'emboîter les uns dans les autres

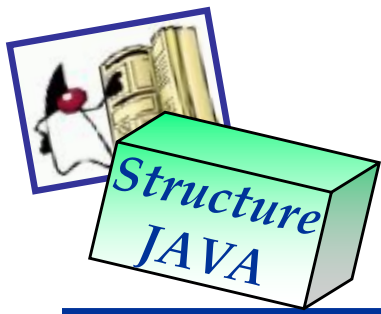
- ```
{
    int ix = iy+1;
    while (ik<MIN) {
        ik = ik*ix;
    }
}
```



# Les identificateurs, les noms de variables

---

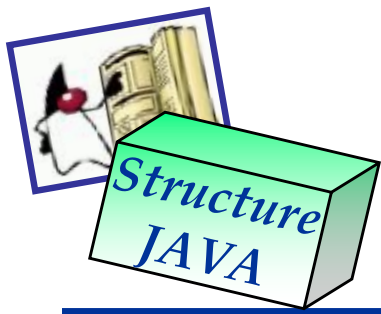
- Il commence par une lettre, un trait de soulignement '\_', ou un '\$'
- Les caractères suivants peuvent comporter des chiffres
- Ils respectent la casse : 'M' ≠ 'm'
- Pas de limitation de longueur
- Ne doivent pas être des mots clés du langage
  - ~~2var\_5~~      ■ identificateur      ■ \_var\_sys      ■ fenêtre
  - MaClasse      ■ \$picsou      ■ M5TLEA3245
  - MaClasseQuiDeriveDeMonAutreClasse



# Les identificateurs

---

- Un identificateur de variable sert à manipuler deux sortes de choses
  - Des variables représentant un élément d'un type primitif
  - Des variables représentant des références (adresses) à des instances (objets)

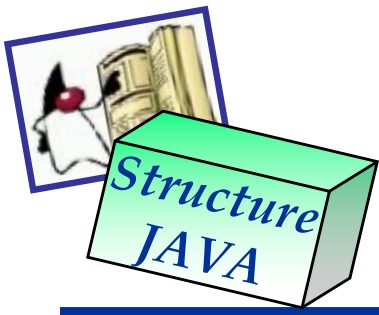


# Les variables

---

- Variable d'instances
  - Déclarées en dehors de toute méthode
  - Modélisent l'état d'un objet
  - Elles sont accessibles par toutes les méthodes de la classe
- Variables locales (à une méthode)
  - Déclarées à l'intérieur d'une méthode
  - Accessible dans le bloc ou elles ont été déclarées



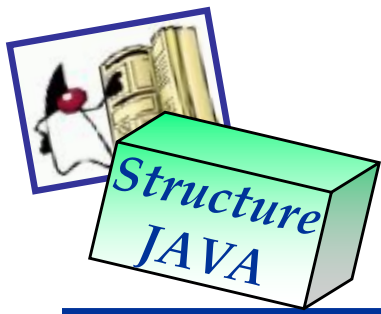


# Déclaration des variables

---

- Toute variable doit être déclarée avant d'être utilisée
- On lui attribue un nom : identificateur et un type
  - String nom;
  - int age;
  - double taille;

```
[modificateur] type nomVariable
```



# Types primitifs

## ■ Les entiers

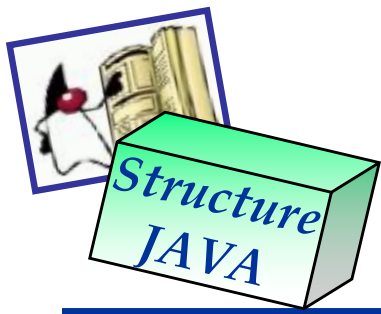
- byte : 8 bits, 1 octet (-128 à 127)
- short : 16 bits, 2 octets ( $-2^{15}$  à  $2^{15}-1$ )
- int : 32 bits, 4 octets ( $-2^{31}$ ,  $2^{31}-1$ )
- long : 64 bits, 8 octets ( $-2^{63}$ ,  $2^{63}-1$ )

## ■ Peuvent être exprimés en:

- Décimal, base 10
- Hexadécimal, base 16
- Octal, base 8

```
byte count;  
short v=32767;  
int i=73;  
int j=0xBEBE; //48830  
long var=077; //63  
//Dans une expression  
98L → décimal de type long  
077L → 63 sur un long  
0xBEBEL → 48830 sur un long
```

Tous les entiers sont par défaut considérés comme des int, pour les voir comme des long il faut suffixer la valeur d'un L ou l



# Types primitifs

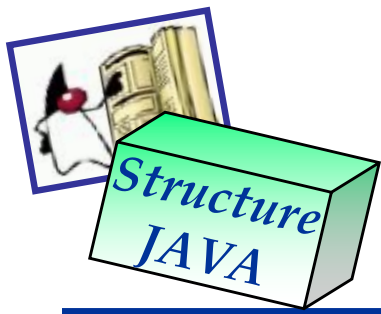
## ■ Les réels

- float : 32 bits, simple précision  
( $1.40239846 \times 10^{-45}$  à  $3.40282347 \times 10^{38}$ )
- double : 64 bits, double précision  
( $4.940656458411246544 \times 10^{-324}$   
à  $1.79769313486231570 \times 10^{308}$ )

## ■ Typage très strict, double par défaut, sinon suffixée par F

```
static final float PI=3.141594635 //erreur  
static final float PI=(float)3.141594635 //OK  
static final float PI=3.141594635F //OK
```

**Définition d'une constante**



# Types primitifs

---

- La logique, les booléens

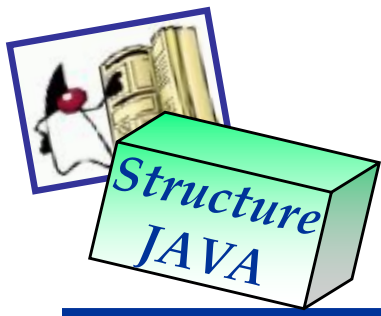
- boolean
- Deux valeurs true ou false

```
boolean isOK = true;  
boolean isnt = false;
```

- Les caractères

- char, codés sur 16 bits, Unicode (\u suivi du code hexadécimal à 4 chiffres du caractère)
- Entourés de « ` »

```
char c='a'; //le caractère a  
char tab='\t' //une tabulation (\n RC)  
char carLu='\u0240' //caractère unicode
```



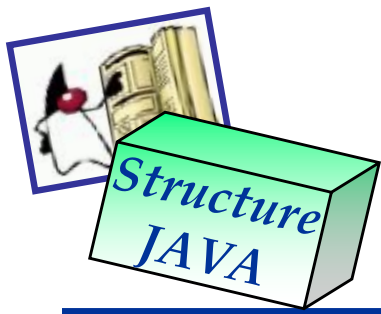
# Résumé : types primitifs

---

- byte, short, int, long
- float, double
- boolean
- char



Il n'existe pas de type primitif chaîne de caractère en java, la 'string' est représentée par un objet de la classe **String**



# Initialisation des variables

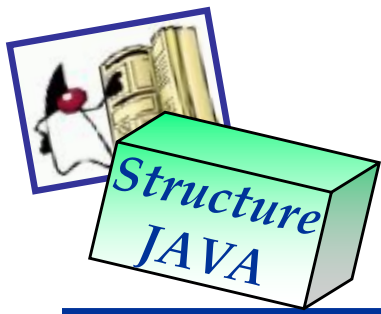
---

- Les variables locales doivent être explicitement initialisées avant d'être utilisées sinon il y a erreur à la compilation
- Les autres variables (d'instance et de classe) si elles n'ont pas explicitement été initialisées reçoivent la valeur par défaut associée à leur type

- **Types**

## Valeur par défaut

<code>byte, int, short</code>	<code>0</code>
<code>long</code>	<code>0L</code>
<code>float</code>	<code>0.0f</code>
<code>double</code>	<code>0.0d</code>
<code>char</code>	<code>'\u0000'</code> (Null en Unicode)
<code>boolean</code>	<code>false</code>



# Variables de type primitif et références

## ■ Distinction entre types primitifs et les autres

```
type nomVariable;  
nomVariable=valeur;
```

nomVariable

valeur

```
int i;
```

```
i = 6;
```

i

6

```
Objet varObjet;  
varObjet=new Objet (...);
```

varObjet

référence

```
String nom;
```

```
nom = new String("Dupond");
```

nom

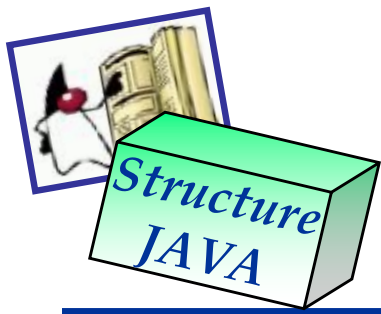
référence

Un objet ou  
un tableau

Dupond

Pour les types primitifs dans la case mémoire il y a directement la valeur

Pour les objets dans la case mémoire il y a seulement une référence



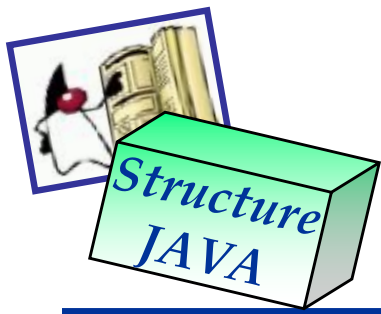
# Typage strict

---

- Attention le typage est très strict en Java.
- Le compilateur refuse de passer d'un type à un autre s'il y a un risque de perte d'information
- C'est pour cette raison qu'il considère par défaut qu'une constante réelle est un double

```
int i=258;    //Ok
long l=i;     //Ok car codé sur plus de bits
byte b=i; //error:explicit cast need to convert int to byte
byte b2=200; // idem
float f=253.62 // error : possible loss of precision, found
               double, required float...
```





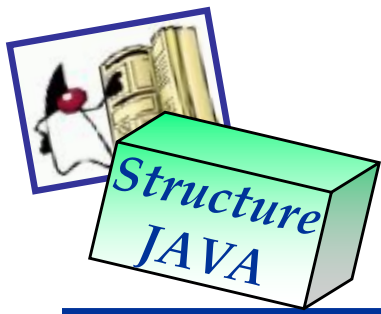
# Forçage de type

---

- Il peut être nécessaire de forcer le programme à considérer une expression comme étant d'un type différent de son type réel ou déclaré
- On effectue alors le *cast* de l'expression
- On force le type

`(typeforcé) expression`

```
int a,b;  
double division = (double)a / (double)b;  
// pour forcer la division réelle
```

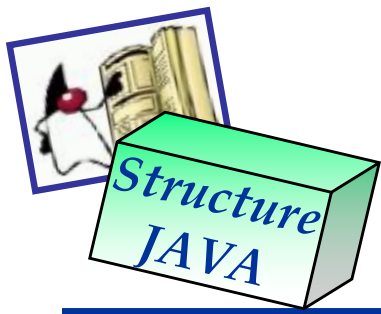


# Forçage de type : cast

---

- Attention : le cast peut être source de perte de précision voire de données
  - Perte de précision : C'est le cas quand on passe d'un long à un float (on perd des chiffres significatifs, mais on conserve l'ordre de grandeur, troncation)
  - Perte de données : Lorsqu'on passe par exemple d'un int (32) à un byte (8)

```
short s = 24;  
char c = s; //Caractère dont le code est 24  
int i = 130;  
byte b=(byte)i; // Ok mais b=-126
```



# Les opérateurs

---

## ■ Arithmétiques

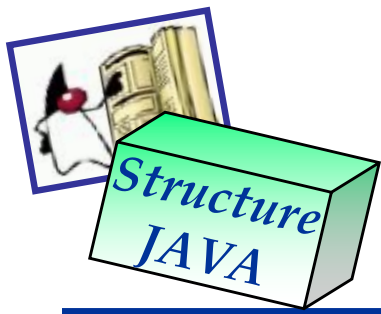
- $+$   $-$   $*$
- $/$  {division entière si les opérandes sont entiers}
- $\%$  {modulo}
- $++$   $--$

## ■ Relationnels

- $==$ ,  $!=$ ,  $>$ ,  $<$ ,  $>=$ ,  $<=$  (résultat booléen)

## ■ Logiques

- $\&\&$   $||$  l'opérande 2 est évalué si besoin
- $\&$   $|$   $!$   $\wedge$  (ou excl.) les 2 opérandes sont évalués



# Les opérateurs

## ■ Opérateurs sur les bits

- Décalage de bit :  $>>$ ,  $<<$ ,  $>>>$
- Opération bit à bit :  $\&$  |  $\wedge$   $\sim$ (complément à 1)

## ■ Référence

- .

## ■ Tableaux

- []

## ■ Groupement

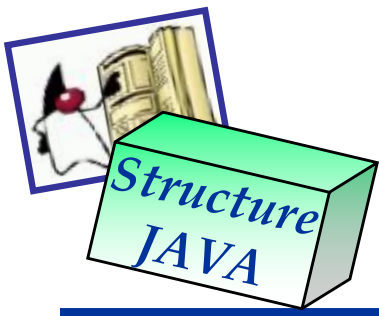
- ( ) { } ;

## ■ Affectation

- =
- += -= \*= /= %=
- ^= ~= |= &= <<= >>= >>>=

## ■ Autre

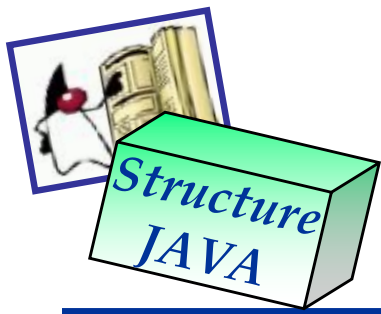
- ,
- ? : (raccourci pour if-else)
- new
- instanceof



# Les opérateurs

---

- Trois types d'opérateurs
  - Unaire : `!a`
  - Binaire : `a && b`
  - Ternaire : `(a > b) ? a : b`
- Les expressions sont évaluées de la gauche vers la droite
- N'hésitez pas à mettre des parenthèses pour « forcer » le calcul en fonction des priorités

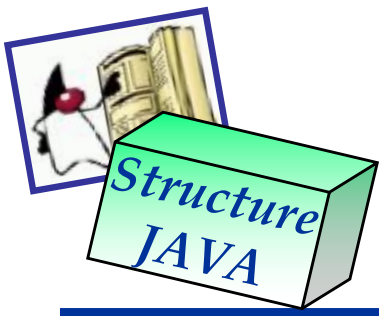


# Les expressions

---

- Le type d'une expression arithmétique dépend des opérateurs et des types des éléments de l'expression
  - Si les éléments ont tous le même type, pas de problème...
  - Sinon, le type final correspond au type le plus précis (le plus d'octets)

```
float * double = double
```



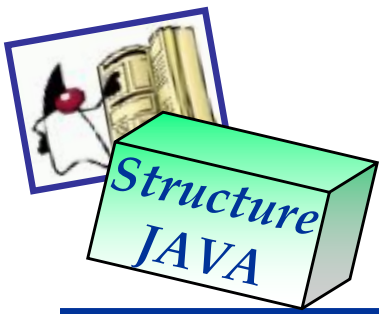
# Les structures de contrôles

If-else, switch

While, do..while, for

Break, continue

return



# Conditionnelle : if...else

- Si condition vrai alors instructions1,  
Sinon instructions2

**Obligatoires si  
plusieurs instructions**

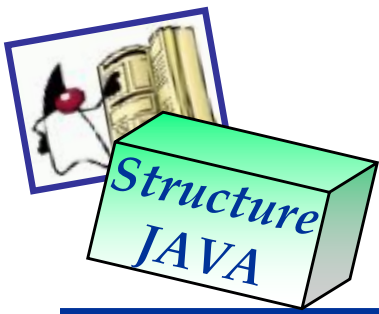
**Optionnel**

```
if (condition) {  
    instructions1  
} else {  
    instructions2  
}
```

```
(condition) ? inst1 : inst2
```

```
if (i<0) i++;  
if (end) { // end doit valoir true  
    fin = true;  
    i=0;  
} else i--;
```



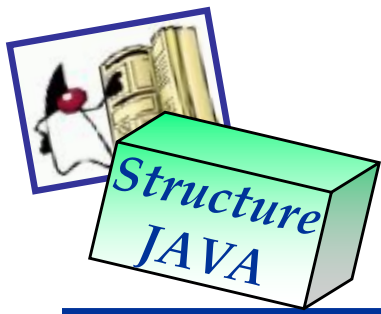


# Conditionnelle : switch

- L'expression doit retourner un type énumérable :  
int, byte, short, boolean, char
- Si break, on invalide les tests suivants
- Sinon on continue sur le case suivant

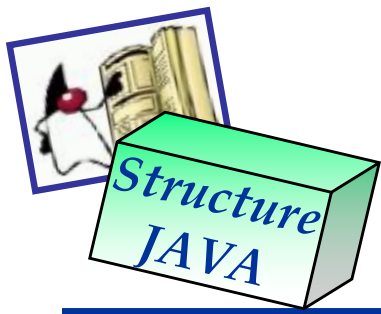
Expression peut aussi être de type String depuis java 7

```
switch (expression) {  
    case exp1 :  
        instruct°;  
        [break;]  
    case exp2 :  
        instruct°2;  
        [break;]  
    ...  
    default :  
        instruct°;  
}
```



# Conditionnelle : switch

```
switch (month) {  
    case 1: System.out.println("January"); break;  
    case 2: System.out.println("February"); break;  
    case 3: System.out.println("March"); break;  
    case 4: System.out.println("April"); break;  
    case 5: System.out.println("May"); break;  
    case 6: System.out.println("June"); break;  
    case 7: System.out.println("July"); break;  
    case 8: System.out.println("August"); break;  
    case 9: System.out.println("September"); break;  
    case 10: System.out.println("October"); break;  
    case 11: System.out.println("November"); break;  
    case 12: System.out.println("December"); break;  
    default: System.out.println("Hey, that's not a valid month!"); break;  
}
```



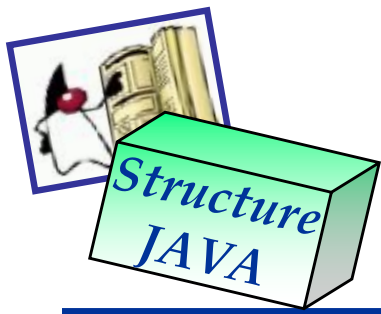
# Itérative : for

- Boucle à nombre d'itération connu

- ① Initialise la variable compteur
- ② Condition de poursuite
- ③ Évolution du compteur

```
for (init; test; incrément) {  
    instructions  
}
```

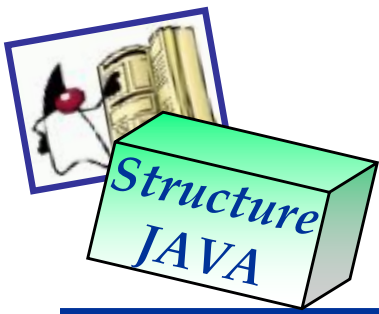
- **break**, sortie de boucle
  - L'exécution reprend après le bloc for
- **continue**, force l'exécution à l'itération suivante



# Itérative : for

## ■ Exemple

```
for (int i=0 ;i<10;i++) {  
    ...  
    if (i==0) continue; //on saute à l'itération i=1  
                        //sans exécuter les instructions  
                        //suivantes  
    if (bonneValI(i)) break; // on sort de l'itération  
}  
int i=0;  
for (;i<100;i++) {...}
```



# Itérative : while

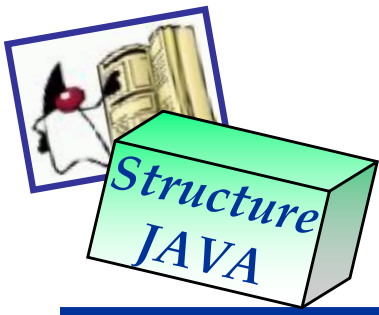
## ■ Boucle à nombre d'itération inconnu

Condition qui doit être  
Vérifiée pour que les  
instructions de la boucle  
soient exécutées

```
while (expression) {  
    instructions;  
}
```

Les instructions de la boucle  
ne sont exécutées que si la  
condition est vérifiée au  
moins une fois

```
int i=0;  
while (i<C.length) {  
    C[i]=i*i;  
    i++;  
}
```



# Itérative : do...while

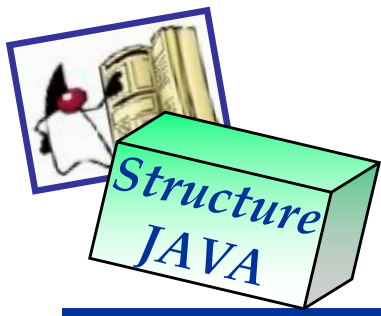
- Boucle à nombre d'itération inconnu, exécutée au moins une fois

```
do{  
    instructions;  
} while (expression);
```

Condition qui doit être vérifiée pour que les instructions de la boucle soient ré-exécutées

Les instructions de la boucle sont exécutées au moins une fois

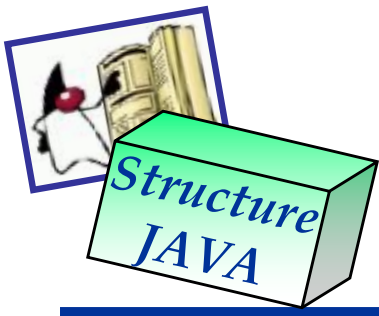
```
int i=0;  
do{  
    C[i]=i*i;  
    i++;  
} while (i<C.length);
```



# Exemple: lecture d'une phrase caractère par caractère

---

```
public static String lirePhraseClavier() throws IOException{  
    String ligne = " ";  
    char c;  
    do{  
        c = (char)System.in.read();//lecture d'un  
                                   //caractère au clavier  
        ligne += c; //reconstruction de la chaîne  
    } while (c != '\n' && c != '\r');  
    return ligne.substring(0,ligne.length()-1);  
}
```

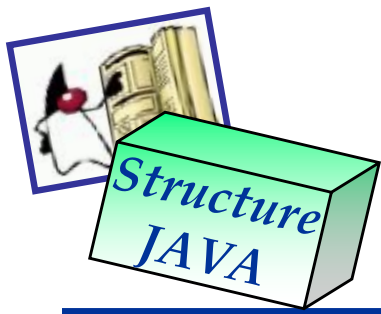


# Itérative : les boucles infinies

---

- for
  - for ( ; ; ) {...}
- while
  - while (true) {...}
- do...while :
  - do {...} while (true)

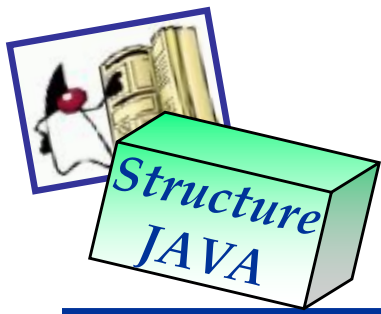




# Break et continue

---

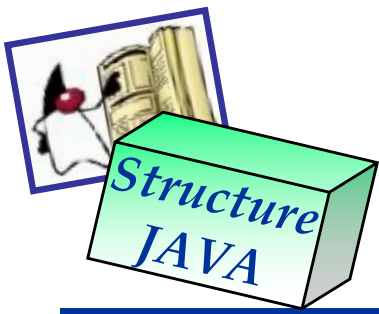
- Les instructions continue et break, sont utilisables dans tous les types de boucles : for, while, do...while
  - continue : passe à l'itération suivante (incrémentement du compteur) sans exécuter les instruction d'après dans la boucle
  - break : sort de la boucle et exécute la première instruction qui suit celle-ci



# Les blocs labellisés

```
un : while (cond1){  
    deux : for (...){  
        trois : while (...){  
            if (...) continue un;  
            if (...) break deux;  
            continue;  
        }  
    }  
}
```

L'étiquette d'un « continue » doit se trouver sur une boucle englobante pour indiquer que l'exécution doit reprendre avec cette boucle et non avec celle qui est directement englobante comme c'est le cas par défaut. C'est la même chose pour le break



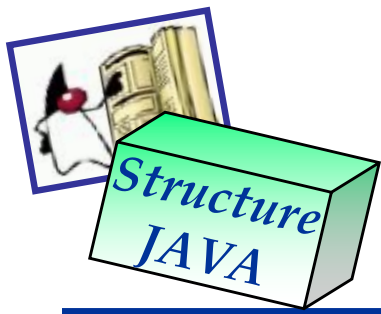
# La portée des variables

---

## ■ Variables locales

- Définies par bloc { }
- Doivent être initialisées explicitement

```
{  
    int ix=0; int iz=0;  
    // portée de la déclaration...  
    { // ici aussi  
    }  
}  
// fin de la portée pour ix et iz  
for(int i=0;i>t;i++){  
    //portée pour i  
}
```



# Les conventions

---

- Conventions de nommage
  - classe : `MaClasse`, `LaTienne`, `Rectangle`,...
  - variables et méthodes : `nombreClients`,  
`calculer`, `calculExact`, `dessiner`,...
  - constante : `TAILLE_MAX`, `MAX_INT`, `PI`, `JAUNE`
  - package : `fr.asso.gourousjava.util`
- Organisation de fichier
  - Une seule classe public par fichier, un seul main