# Chapitre 4 : exécution des programmes♪

Notion de variable informatique

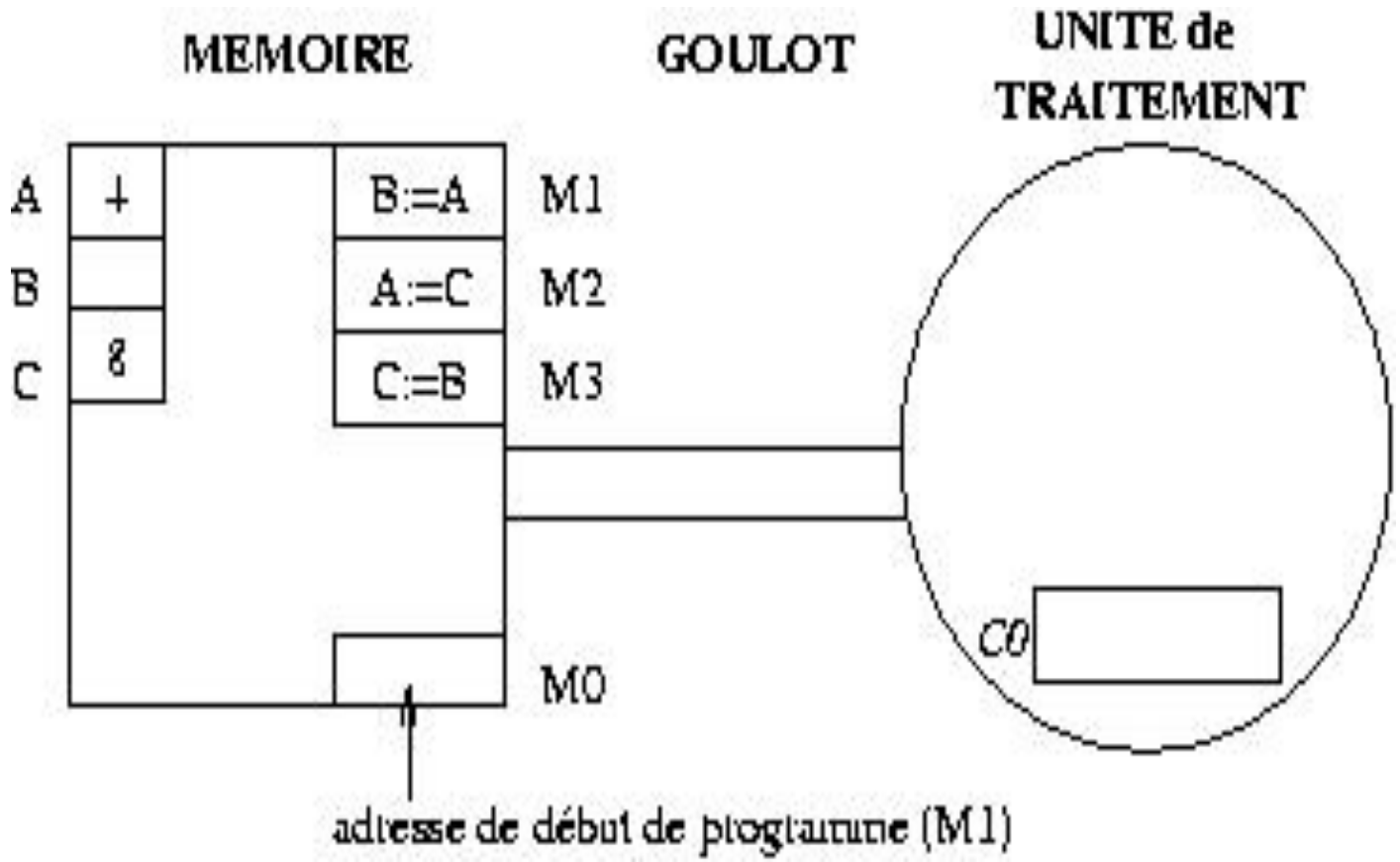
Affectation et séquence

Etat d'un programme et déclarations

Structures de contrôle

# le modèle Von Neumann

- Les machines actuelles, dans leur forme la plus rudimentaire, s'apparentent à un modèle qui date de la fin des années 40, le modèle Von Neumann.
- Selon ce modèle, un ordinateur est constitué de 3 parties :
  - une unité centrale de traitement (CPU)
  - une mémoire
  - un tube qui connecte CPU et mémoire et permet de transmettre des mots de l'une vers l'autre.




Programme de permutation des 2 valeurs 4 et 8 dont les noms en mémoire sont, à l'instant initial A et C :

```
B=A;
A=C;
C=B;
```

| actions | passages | commentaires |
|---|---|---|
| M0->C0 | 1 | adresse de début → dans le compteur ordinal |
| C0->M1 | 1 | transfert de l'adresse de la 1ère instruction |
| B=A->UAL | 1 | transfert de l'instruction dans l'UAL |
| UAL->A | 1 | transfert de l'adresse A |
| 4->UAL | 1 | transfert de la valeur 4 dans l'UAL |
| UAL->B | 1 | transfert de la valeur 4 dans B |
| C0->M2 | 1 | transfert de l'adresse de la 2ème instruction |
| A=C->UAL | 1 | transfert de l'instruction dans l'UAL |
| UAL->C | 1 | transfert de l'adresse C |
| 8->UAL | 1 | transfert de la valeur 8 dans l'UAL |
| UAL->A | 1 | transfert de la valeur 8 dans A |
| C0->M3 | 1 | transfert de l'adresse de la 3ème instruction |

| | | |
|---|---|---|
| C=B->UAL | 1 | transfert de l'instruction |
| UAL->B | 1 | transfert de l'adresse B |
| val(B)->UAL | 1 | transfert de la valeur de B dans l'UAL |
| UAL->C | 1 | transfert de 4 dans C |
| | =16 | |

# Variable informatique

- Les cellules mémoires, référencées par leurs adresses (nommées A, B, C, M0, M1, etc...), sont modélisées dans les langages de programmation par des *variables informatiques*. Une variable informatique prend donc des valeurs différentes au cours de l'exécution d'un programme.

- On notera la différence entre une variable informatique et une variable au sens mathématique qui n'est qu'une entité abstraite ne pouvant désigner qu'une unique valeur.

# Notion d'affectation

- Le moyen d'"imiter" les actions d'"aller chercher dans la mémoire" et de "mettre en mémoire" est donné par

  **l'instruction d'affectation** notée = en Java (comme dans l'exemple précédent).

- C'est l'instruction qui sert essentiellement à modifier la valeur d'une variable.

# Exemple

```
X=Y+2;
```

On va chercher la valeur de `Y` en mémoire, on lui ajoute 2 dans l'UAL, puis on transfert cette valeur à l'adresse notée symboliquement `X`

```
X=X/4;
```

On va chercher la valeur de `X` en mémoire, on la divise par 4 dans l'UAL, puis on transfert cette valeur à l'adresse notée symboliquement `X`

# Affectation

- La valeur de la partie droite est enregistrée à l'adresse de la variable en partie gauche.

- L'affectation sépare la programmation en 2 mondes :
  - Le premier, concerné par la partie droite de l'affectation, est un monde d'expressions avec leurs propriétés algébriques, un monde dans lequel ont lieu les calculs.
  - Le second, concerné par la partie gauche est un monde d'adresses permettant de localiser des données (valeurs ou instructions) dans la mémoire.

# Séquence

- L'idée imposée par le modèle Von Neumann de penser un programme en terme de trafic à travers le goulot amène à organiser séquentiellement la suite d'affectations.

- En Java, une séquence d'instructions est obtenue en séparant les instructions successives par un `;` .

# Calcul et valeurs

- Un calcul exprime une solution à un problème. Il est formé d'expressions construites à l'aide de valeurs (arithmétiques ou autres) et d'opérateurs.
- Tout programme manipule, lit, écrit, crée des valeurs.

- Les valeurs élémentaires sont :
  - les caractères
  - les entiers
  - les réels
  - les booléens
  - les chaînes de caractères

| représentation Java | | | |
|---|---|---|---|
| 3 | 3. | '3' | "3" |
| TRUE | "TRUE" | | |

# Représentation des valeurs littérales en Java

- Un entier est représenté classiquement :

```
3      56      987
```

- Un réel est représenté avec la notation pointée :

```
3.1415
```

- Un caractère est représenté entre quotes pour le distinguer de tout autre identificateur :

```
'A'   '7'     '$'
```

- Une chaîne de caractères est représentée entre guillemets pour la distinguer de tout autre identificateur ou mot clé :

```
"Hello World "
```

- Un booléen est représenté par les symboles :

```
true, false
```

# Expressions

- ✓ Les valeurs constituent des ensembles.
- ✓ L'environnement initial (qu'il est inutile d'importer explicitement) comporte tous les opérateurs sur les valeurs de ces ensembles de base.
- ✓ Une expression est formée à partir des valeurs et opérateurs associées à cet ensemble.
- ✓ Une expression crée ainsi une nouvelle valeur.

**Exemple**

```
(15+30)/2
true && false
```

# Exécution et contrôle

Un programme met en oeuvre un calcul sur un **exécutant** informatique.

Un programme contient l'expression d'un calcul et les éléments qui permettent son exécution, les mécanismes de contrôle (boucles, séquence, conditionnelle, exceptions, appels de fonctions et de procédures)

Les mécanismes de contrôle permettent l'ordonnancement des instructions d'un programme.

Pour ordonnancer les instructions, il faut définir comment on continue un calcul..

On répond ainsi à la question : quelle instruction exécuter ensuite ?

# Exécutant

Il est défini par :

- un modèle d'exécution implanté : le mécanisme de gestion de la continuation
- un mécanisme d'exécution des instructions

# Etat d'un programme

- L'état d'un programme correspond à la description d'une étape de son exécution. On parle d'état courant pour spécifier un état à un instant donné de l'exécution.

- L'état d'un programme est constitué des objets informatiques manipulés par le programme à une étape donnée de l'exécution.

- Plus formellement, l'état d'un programme est représenté par le couple formé de l'environnement (Env) et de la mémoire (Mem).

# Environnement

- L'environnement rassemble l'ensemble des identificateurs d'objets, utilisables au cours d'une exécution, associés à leur type et leur valeur.
- C'est une liste de triplets (identificateur, type, valeur).

| **Identificateur** | **Valeur** |
|---|---|
| variable | adresse |
| constante | valeur |
| exception | valeur d'exception |
| sous-programme | fermeture |
| paquetage | environnement |
| type | valeur de type |

# Mémoire

- De manière simplifiée, la mémoire contient des valeurs référencées par des adresses. La mémoire est vue comme une liste de couples (adresse, valeur référencée).

- Tout identificateur appartenant à l'environnement référence une valeur de la mémoire. Toute valeur de la mémoire non associée à un identificateur devient inaccessible.

- Un processus ramasse-miettes ("garbage collector") récupère les zones mémoire allouées à des objets devenus inaccessibles.

# Généralités sur les déclarations

- Une déclaration d'objet permet :
  - ✓ d'introduire son identificateur dans l'environnement.
  - ✓ de préciser les conditions de son utilisation (son type)
  - ✓ de définir une valeur initiale (facultativement)

# Variables informatiques

- Une variable informatique est un objet informatique dont la valeur peut changer au cours de l'exécution du programme.

- A une adresse donnée correspond un ensemble de valeurs référencées possibles (notion de type) au cours de l'exécution d'un même programme .

- Une variable peut être :
  - anonyme (création dynamique)
  - nommée (liée à un identificateur)

# Déclaration de variables

- **syntaxe**

```
<declaration_variable>::=
    <type><identificateur> [= <expression>];
   |<type><identificateur>{,<identificateur>}
                                    [= <expression>];
```

```
double euros,€;
char X = 'X';
```

- **sémantique**

  - allocation d'un espace mémoire (de la taille spécifiée par le type) destiné à mémoriser les valeurs qui lui seront successivement affectées

  - association du nom de la variable à cet espace

- **initialisation**

  la valeur de la partie droite est, si elle existe, enregistrée dans cet espace

# Déclaration d'une variable : sémantique (1/3)

- Soit l'état courant

```
Etat0 = (Env0, Mem0)
```

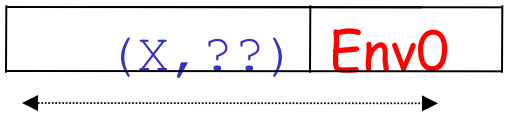
- et la déclaration

```
int X = 12;
```

✓ Extension de l'environnement `Env0` avec le couple :

```
(X,??)
```

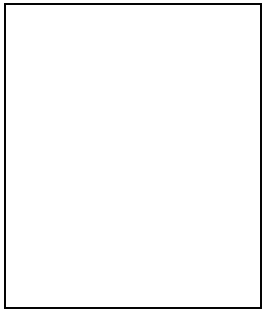
d'où




Env1

et

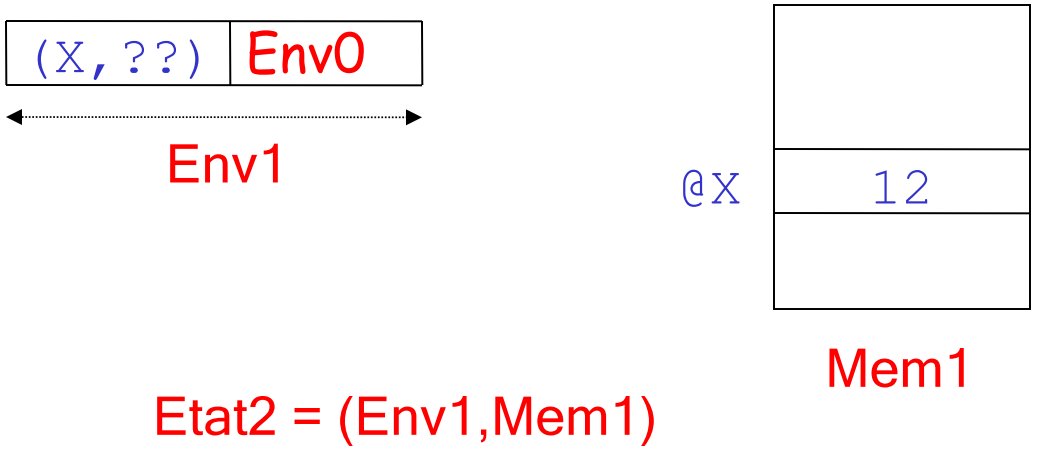
```
Etat1 = (Env1,Mem0)
```



Mem0

# Déclaration d'une variable : sémantique (2/3)

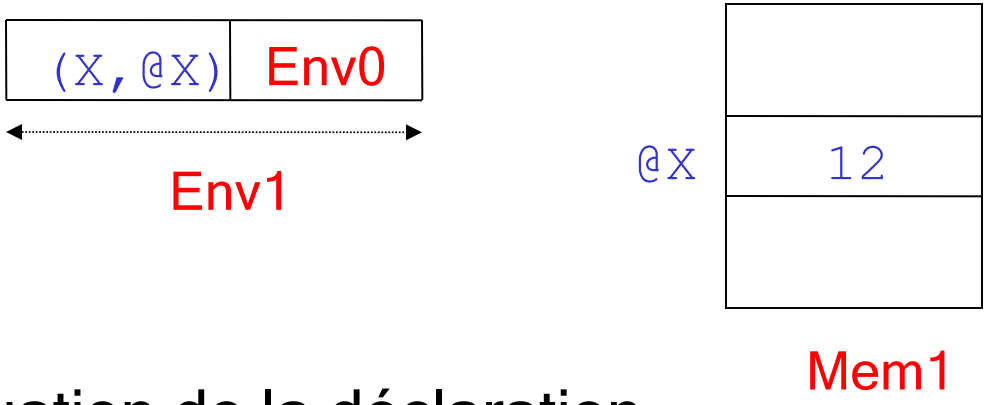
- ✓ Détermination du type et des contraintes
- ✓ Evaluation de l'expression initiale dans ce nouvel état (Etat1) : l'expression vaut `12`
- ✓ Adjonction du couple `(@X,12)` dans Mem0 qui devient Mem1




Etat2 = (Env1,Mem1)

# Déclaration d'une variable : sémantique (3/3)

- ✓ Modification de Env1




- ✓ Après évaluation de la déclaration,

Etat3 = (Env1,Mem1)

# Déclaration d'une variable : exemple 1 (1/2)

Etant donné l'état :

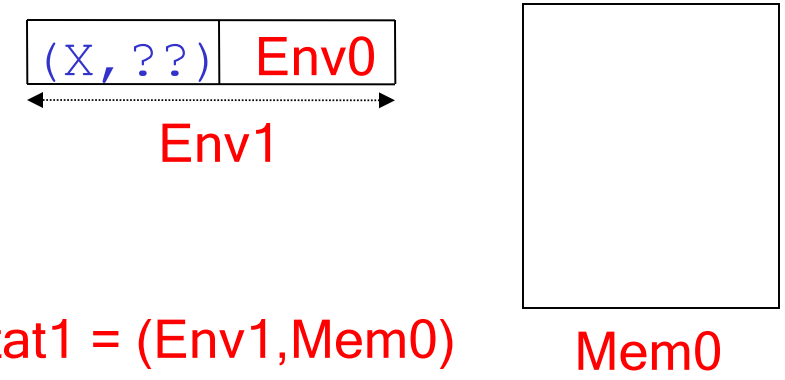
```
Etat0=(Env0,Mem0)
```

et la déclaration

```
int X = X+1;
```

# Déclaration d'une variable : exemple 1 (2/2)

Extension de l'environnement Env0 avec le couple : `(X, ??)`

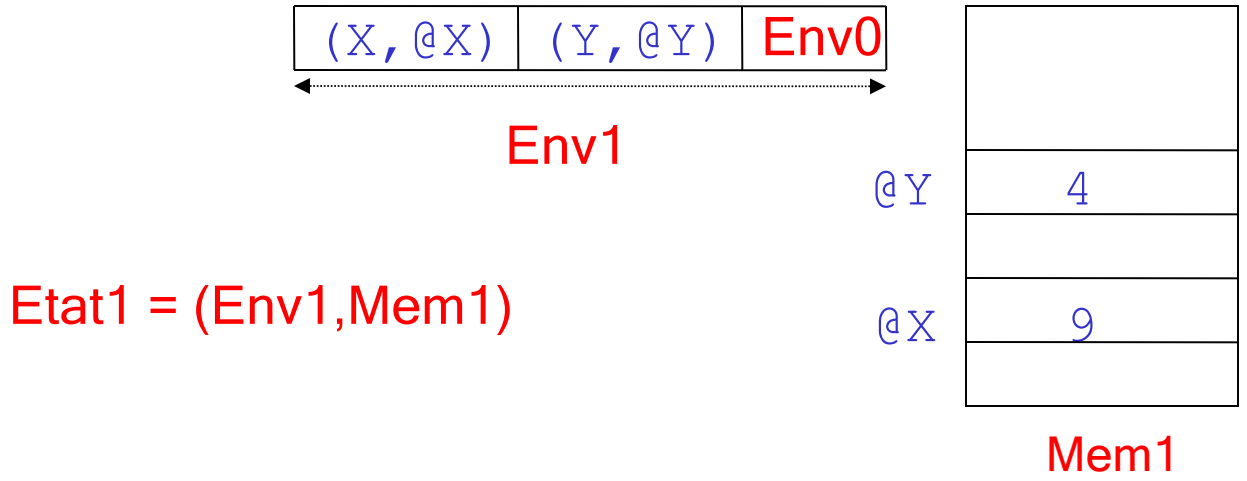



- ✓ Détermination du type et des contraintes
- ✓ Evaluation de l'expression `X+1` dans Etat1

L'identificateur `X` n'étant lié à aucune valeur, l'évaluation est impossible. Une telle initialisation est donc interdite.

# Déclaration d'une variable : exemple 2 (1/4)

- Etant donné l'état :

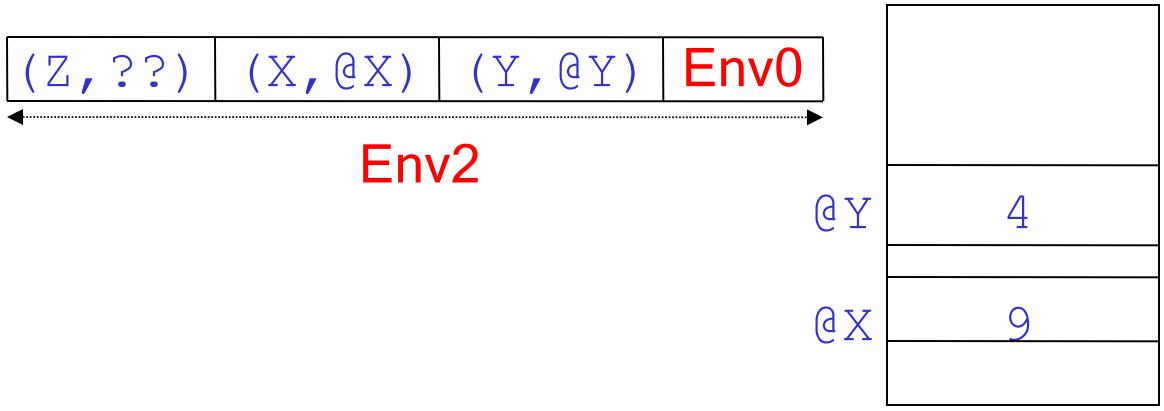



- et la déclaration :

```
int Z = Y-X;
```

# Déclaration d'une variable : exemple 2 (2/4)

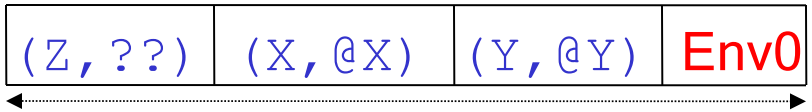
Extension de l'environnement Env1 avec le couple (Z,??) :




Etat2 = (Env2,Mem1)

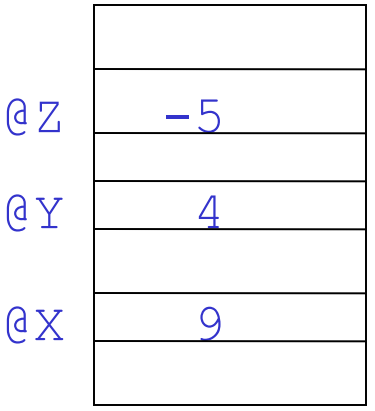
# Déclaration d'une variable : exemple 2 (3/4)

- ✓ Détermination du type et des contraintes
- ✓ Evaluation de l'expression dans ce nouvel état Etat2 : l'expression `Y-X` vaut `-5`
- ✓ Adjonction du couple `(@Z,-5)` dans Mem1 qui devient Mem2. Etat3 est le nouvel état




Env2

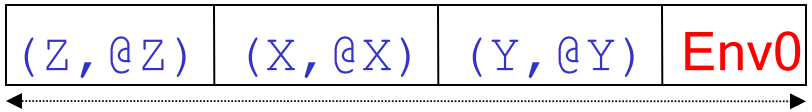
Etat3 = (Env2,Mem2)

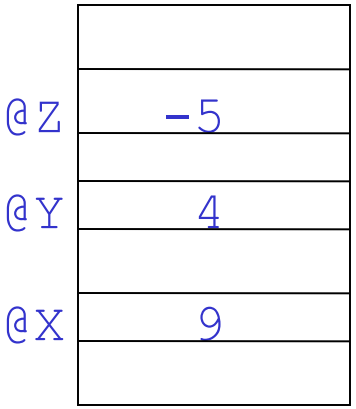



Mem2

# Déclaration d'une variable : exemple 2 (4/4)

Modification de Env2




Après évaluation de la déclaration,

Etat4 = (Env2,Mem2)

# Constantes

Une constante est un objet informatique dont la valeur ne change pas durant l'exécution d'un programme.

Contrairement aux variables, les constantes représentent des données permanentes

## Syntaxe

```
<declaration_constante>::=
    final <type><identificateur> = <expression>;
```

## Exemple

```
public class Surface{
 public static void main(String[] args){
    final double PI = 3.14159;
    double rayon = 20;
    double surface = rayon*rayon*PI;
    System.out.print("la surface du cercle de rayon:");
    System.out.println(rayon+" est "+surface);
 }
}
```

# Déclaration d'une constante : sémantique

La déclaration d'une constante permet de nommer un objet et de l'ajouter à l'environnement courant :

- ✓ Ajout de l'identificateur, associé à une valeur indéfinie, à l'environnement
- ✓ Evaluation de l'expression constante
- ✓ Remplacement de la valeur indéfinie par la valeur évaluée dans l'environnement courant

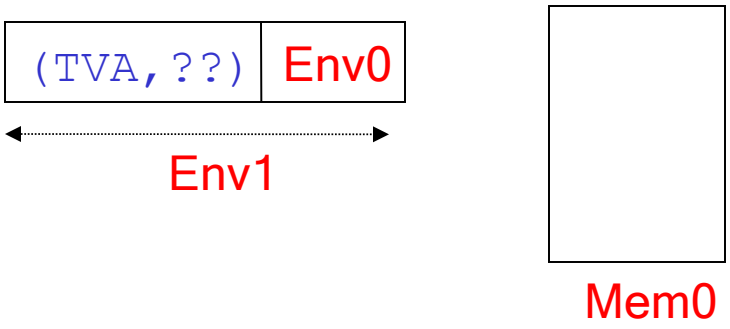
# Exemple (1/2)

Soit, dans l'état (Env0,Mem0) la déclaration :

```
final float TVA = 19.6f;
```

Elaboration en 3 étapes :

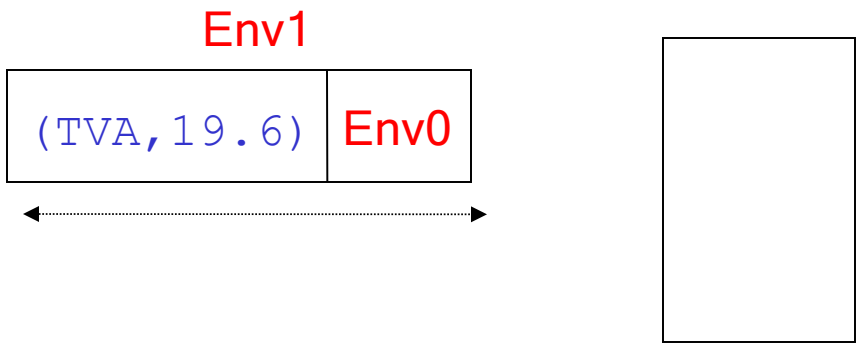
1. Ajout de l'identificateur `TVA` associé à une valeur indéfinie dans Env0. On obtient le nouvel environnement Env1 :



# Exemple (2/2)

1. Evaluation de l'expression constante ---> `19.6`

2. Remplacement de la valeur indéfinie ( ?? ) par la valeur (évaluée dans Env1)



# Affectation d'une valeur à une variable

- **Syntaxe**

```
<affectation>::=
      <expression_gauche>=<expression droite>;
```

où

`<expression_gauche>` dénote une adresse

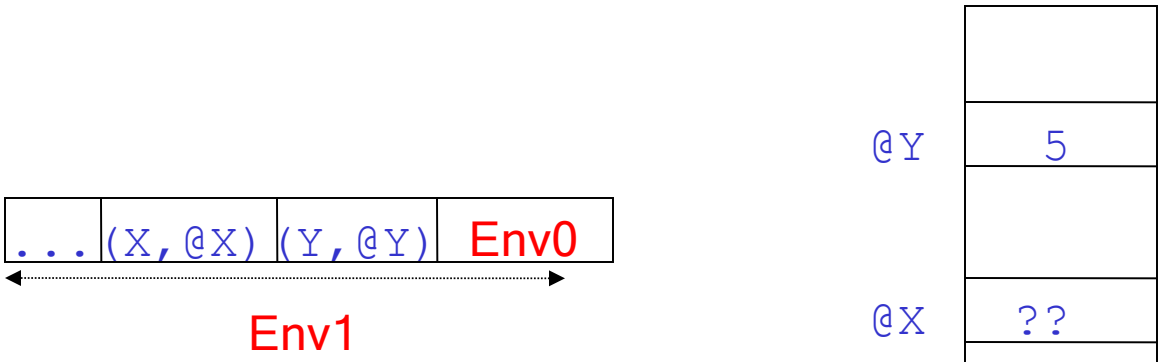
`<expression droite>` dénote une valeur

- **Exemples** : `X='X'; euros=56.8*7.896;`
- **Sémantique**
  - calcul de la valeur de l'expression droite
  - la valeur est ensuite enregistrée dans l'emplacement mémoire désigné par le nom de la variable
- **Expressions**
  - construites à partir d'opérateurs :

    `+,-,/,*,%,&&,||,!,<,>,<=,>=,==!=`

# Affectation (1/3)

Evaluation de l'affectation : `X=Y+2;` dans (Env1,Mem1)



# Affectation (2/3)

L'évaluation de `<expression_gauche>` (c'est à dire `X`) dans cet état retourne l'adresse de `X` : `@X`

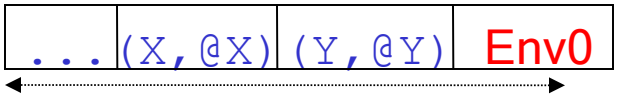
L'évaluation de `<expression_droite>` (c'est à dire `Y +2`) dans cet état nécessite d'abord l'évaluation des 3 symboles `Y, +, 2` :

- La valeur de `Y` est : `@Y`, la valeur de la variable est donc `5`
- `2` est une constante littérale, elle s'évalue en elle-même: `2`
- La valeur de `+` est sa fermeture, l'environnement d'exécution peut donc être constitué et son code binaire exécuté
- Le résultat de l'évaluation de `Y+2` est : `7`

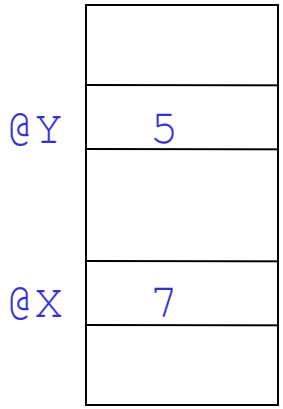
# Affectation (3/3)

La valeur obtenue est enregistrée à l'adresse de X

D'où le nouvel état : (Env1,Mem2)




Env1




Mem2

# Structures de contrôle

conditionnelles

itératives

notion de bloc

# Structures de contrôle

- La seule instruction qui a des effets sur le contenu de la mémoire est l'affectation.
- Les autres instructions ne sont là que pour permettre d'organiser de manière à la fois plus concise et plus lisible la suite des affectations.
- On appelle ces instructions des structures de contrôle.
- On distingue les structures de contrôle conditionnelles et itératives.

# Conditionnelle

## Syntaxe

```
<conditionnelle>::=
      if(<expression_booléenne>)
         <suite_instructions>
      [ else <suite_instructions> ]

<suite_instructions>::=
      <instruction_simple>|{<suite_instructions>}
```

## Sémantique

- Les parties entre crochets sont optionnelles
- Si la condition est vraie, la suite d'instructions associée est exécutée et l'instruction conditionnelle est terminée
- Si la condition est fausse, la partie **`else`** est exécutée et l'instruction conditionnelle est terminée

# Exemple : calcul du prix TTC(1/2)

Algorithme

```
début
  prixHT ← lire le prix HT;
  t ← lire le type de taux TVA (réduit ou normal)
  si t=type taux TVA réduit
  alors
     prixTTC ← prixHT+(prixHT*0.005);
  sinon
     prixTTC ← prixHT+(prixHT*0.196);
  fin si;
  afficher le prix TTC;
fin.
```

# Exemple : calcul du prix TTC(2/2)

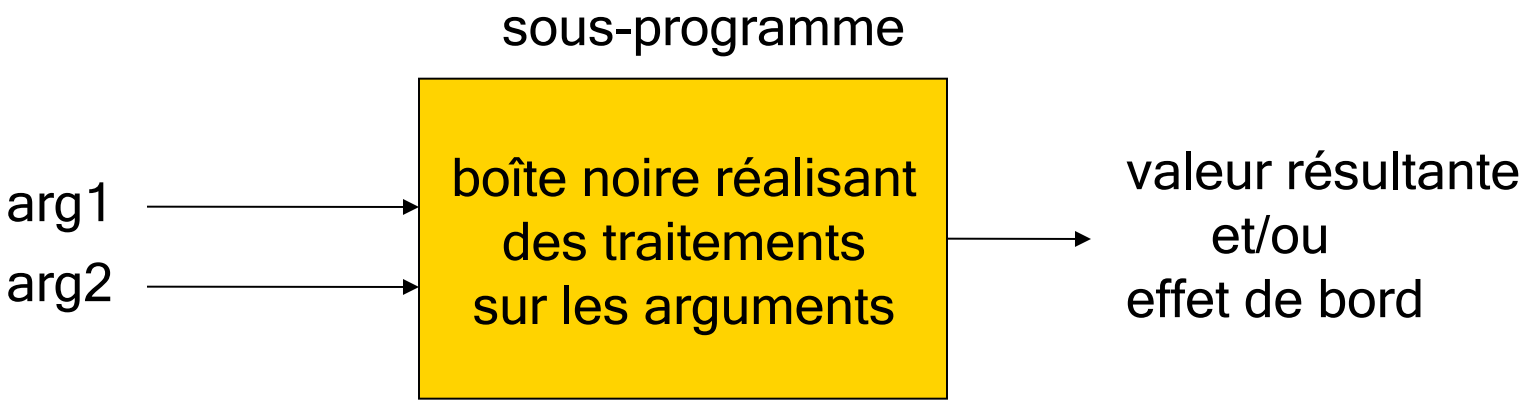
```
import java.util.Scanner;
public class PrixTTC{
  public static void main(String[] args){
    final int REDUIT = 0;
    final int NORMAL = 1;
    Scanner in = new Scanner(System.in);
    double prixHT,prixTTC;
    System.out.print("entrer le prix HT : ");
    prixHT = in.nextDouble();
    System.out.print
           ("prix réduit: taper 0, prix normal: taper 1 => ");
    int t = in.nextInt();
    if( t == REDUIT )
       prixTTC = prixHT + (prixHT*0.05);
    else
       prixTTC = prixHT + (prixHT*0.196);
    System.out.print( "le prix TTC est : "+ prixTTC );
  }
}
```

```
entrer le prix HT : 65,95
prix réduit: taper 0,
prix normal: taper 1 => 0
le prix TTC est : 69.2475
```

# Appel de méthode

Syntaxe

```
<ident_sous_programme>(<arg>,<arg>,...,<arg>)
```

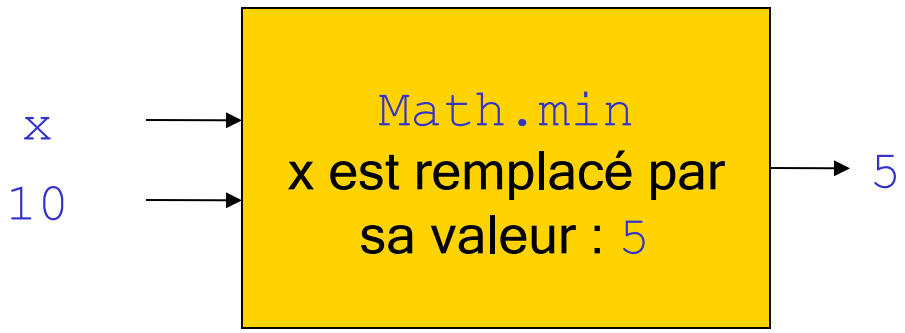



• Le sous-programme est invoqué en lui passant les données (arguments) à traiter.
• Les arguments peuvent changer à chaque appel
• Comme résultat, on obtient une valeur ou bien une modification de l'état de la machine appelée effet de bord ou bien encore les deux.

# Exemple

```
int x = 5;
x = x + Math.min(x,10) + 4;
```

appel de la méthode `Math.min`




1-x←5

2- évaluation de `x +Math.min(x,10)+4`

3- suspension du calcul de l'expression et évaluation de l'appel de la fonction `Math.min(x,10)`

- transmission des arguments
- le contrôle est passé aux instructions de la fonction qui calcule `5`
- le contrôle revient au calcul de l'expression en retournant la valeur `5`

4- l'évaluation de l'expression devient : `5+5+4 =>14`

5- `x←14`

# Exemple : racines d'équation du second degré (1/2)

Algorithme : résolution de $a*x^2+b*x+c=0$

```
début
  si a=0 alors x←-b/c;
  sinon
      delta←b²-4*a*c;
      si delta>0
      alors
       x1←(-b-sqrt(delta))/2*a;
       x2←(-b+sqrt(delta))/2*a;
   sinon
       si delta=0
       alors x←-b/2*a;
       sinon afficher "cas des racines complexes"
       fin si;
   fin si;
  fin si;
fin.
```

# Exemple : racines d'équation du second degré(2/2)

```
public class Degre2{
 public static void main( String[] args ){
    // calcul des racines de l'équation ax2+bx+c=0
    double a=3,b=5,c=1,x,x1,x2,delta;
    if(a==0.0){ x=-b/c; // calcul du cas linéaire
              System.out.println(" cas linéaire : "+x); }
    else{ // calcul du discriminant
      delta = java.lang.Math.pow(b,2.0)-4.0*a*c;
      if (delta>0.0){ //calcul des racines réelles
         x1=(-b-java.lang.Math.sqrt(delta))/2*a;
         x2=(-b+java.lang.Math.sqrt(delta))/2*a;
 System.out.println(" cas des racines réelles : "+x1+", "+x2);
      }
      else
        if( delta==0 ) x=-b/2*a;
        else System.out.println(" cas des racines complexes");
   }}}
```

# Expressions conditionnelles

```
<expression_conditionnelle>::=
  <expression_booléeenne> ? <expression> : <expression>
```

Exemples :

```
int y = (x>0) ? 1 : -1
// équivalent à
int y;
if (x>0) y=1; else y=-1;

System.out.println((num%2==0) ? "num pair":"num impair");
// ? est le seul opérateur ternaire en Java
```

# Choix multiple : Syntaxe

```
<choix>::=
switch(<expression>){
  case <valeur>:<instructions>;break;
  case <valeur>:<instructions>;break;
  case <valeur>:<instructions>;break;
  [default : <instructions>;]
}
```

`<expression>` est obligatoirement de type `char,byte,short` ou `int`

`<valeur>` est une constante de même type que `<expression>`

`<instructions>` est une suite d'instructions simples

# Choix multiple : Sémantique

- `<expression>` est évaluée. Sa valeur appartient à un type entier ou `char`
- Les valeurs utilisées en tant que filtre sont comparées de haut en bas à la valeur de `<expression>`
- Dès qu'il y a égalité entre la valeur de l'expression et celle du filtre, la suite d'instructions correspondante est exécutée. L'instruction **`break`** termine immédiatement l'instruction **`switch`**
- Bonne pratique : l'ensemble des filtres d'une instruction à choix multiple devrait représenter exhaustivement l'ensemble des valeurs du type de l'expression.
- Le filtre **`default`** capte les valeurs qui n'ont pas été citées en tant que filtre. Il est optionnel.

# Exemple : la calculette(1/2)

Algorithme

```
début
    opérateur ← lire opérateur;
    X ← lire premier opérande;
    Y ← lire seconde opérande;
    selon opérateur faire
        cas + : R←X+Y;
        cas - : R←X-Y;
        cas * : R←X*Y;
        cas / : R←X/Y;
        sinon : afficher "opérateur inconnu";
    fin selon;
    afficher la valeur de R;
fin.
```

# Exemple : la calculette(2/2)

```
import java.util.Scanner;
public class Calculette{
    public static void main( String[] args ){
      char operation;int X,Y,R=0;
      Scanner in = new Scanner( System.in );
      System.out.print("opérateur : ");
      operation=in.next().charAt(0);
      System.out.print("X : ");X=in.nextInt();
      System.out.print("Y : ");Y=in.nextInt();
      switch( operation ){
          case '+' : R=X+Y;break;
          case '-' : R=X-Y;break;
          case '*' : R=X*Y;break;
          case '/' : R=X/Y;break;
          default: System.out.println("opérateur inconnu");
  }
 System.out.println("R = "+R);
}}
```

# Structure itérative: boucle `while`

Les instructions itératives permettent de répéter un traitement décrit par une suite d'instructions tant qu'une décision d'interruption de l'itération n'est pas rencontrée.

### Syntaxe

```
<boucle>::=
        while(<condition>){
            <suite_instructions>;
        }
```

### Sémantique

`<condition>` est une expression booléenne évaluée avant chaque itération

- si `<condition>` est vrai, `<suite_instructions>` est exécutée, puis de nouveau la condition est évaluée
- si `<condition>` est faux, le contrôle passe à l'instruction qui suit la boucle

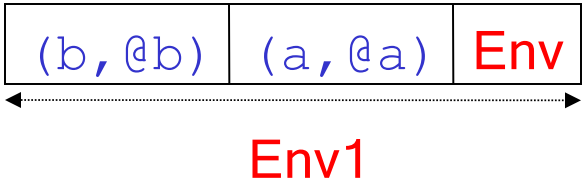
# Boucle tant que : sémantique (1/4)

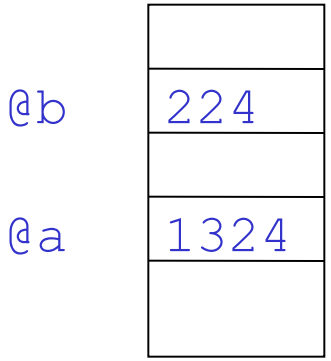
**Exemple :** calcul du PGCD de deux entiers naturels a et b par divisions successives

La boucle va permettre de construire une suite de valeurs pour les variables x et y respectivement initialisées à a et b avec (a>b).

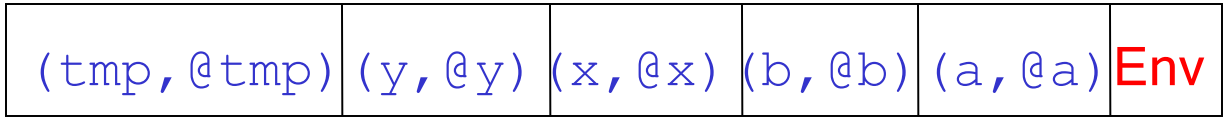
chaque terme courant $(x_i, y_i)$ est déterminé par le terme précédent : $(y_{i-1}, x_{i-1} \text{ modulo } y_{i-1})$

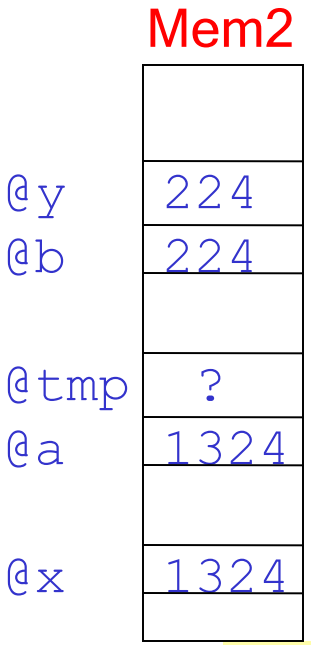
Le calcul de la suite s'arrête lorsque $x_k \text{ modulo } y_k = 0$




```
int x = a;
int y = b;
int tmp;
```



# Boucle tant que : sémantique (3/4)

```
// on suppose que x>=y
while (!(x%y==0)){   // x->x_{i-1}, y->y_{i-1}
      tmp = x;       // x->x_{i-1}, tmp->x_{i-1}
      x = y;         // x->x_i, y->y_{i-1}
      y = tmp%y;     // y->y_i
  };
System.out.println
                  ("PGCD("+a+ ","+b+ ")="+y);
```

# Boucle tant que : sémantique (4/4)

L'exécution d'un pas d'itération ne modifie que la mémoire.

Au terme du premier passage,

$x_0=1324,\ x_1=224,\ x_2=204,\ x_3=20$

$y_0=224,\ y_1=204,\ y_2=20,\ y_3=4$

la condition d'arrêt : `x mod y = 0` est donc satisfaite et le résultat est $y=4$

Résultat

```
a=1324
b=224
PGCD(1324,224)=4
```

# Exemple : somme d'un nombre quelconque d'entiers(1/2)

Algorithme

```
// on suppose que le programme s'arrête lorsque
// l'entier lu est égal à 0
début
   somme ← 0;
   N ← lire un entier;
   tant que N≠0 faire
    somme ← somme+N;
    N ← lire un entier;
   fin tant que;
   afficher la somme;
fin.
```

# Exemple : somme d'un nombre quelconque d'entiers(1/2)

Algorithme

```
// on suppose que le programme s'arrête lorsque
// l'entier lu est égal à 0
début
   somme ← 0;
   N ← lire un entier;
   tant que N≠0 faire
    somme ← somme+N;
    N ← lire un entier;
   fin tant que;
   afficher la somme;
fin.
```

# Exemple : somme d'un nombre quelconque d'entiers(2/2)

```
import java.util.Scanner;
public class SommeEntiers{
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
        int somme=0;
        System.out.print("taper un entier : ");
        int N=in.nextInt();
        // lecture d'une nouvelle donnée jusqu'à ce que N=0
        while( N!=0 ){
            somme=somme+N;
            System.out.print("taper un entier : ");
            N=in.nextInt();
        }
        System.out.print( "la somme est : "+ somme );
    }
}
```

```
taper un entier : 5
taper un entier : 8
taper un entier : 97
taper un entier : 2
taper un entier : 0
la somme est : 112
```

# Exemple : racine carrée entière (1/2)

Algorithme : calcul de la racine carrée entière de N, entier positif

```
début
  X ← 0;
  Y ← 1;
  N ← lire un entier;
  // X<=N && Y=(X+1)*(X+1)
  tant que N>=Y faire
   X ← X+1;
   Y ← X²+2*X+1; // ≡(X_{i-1}+1)²+2*(X_{i-1}+1)+1≡X_i²
  fin tant que;
  afficher X, racine carrée entière de N;
fin.
```

Construire la suite des $X_i$ :
$X_0 = 0$
$X_i = X_{i-1} + 1$
Jusqu'à : $N < X_i^2$

# Exemple : racine carrée entière (2/2)

Spécification formelle : $N >= 0 \Rightarrow \exists x,\ x^2 <= N < (x+1)^2$

```
import java.util.Scanner;
public class RacineCarreeEntiere{
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
      int X=0,Y=1,N;
      // X<=N && Y=(X+1)*(X+1)
      System.out.print("taper un entier : ");
      N=in.nextInt();
      while( N>=Y){
         X=X+1;
         Y=Y+2*X+1; // Y=(Xi-1+1)^2
      }
      System.out.println
        ( "la racine carrée entière de "+N+" est : "+ X );
    }
}
```

taper un entier : 678
la racine carrée entière de 678 est : 26

# Structure itérative: boucle do...while

**Syntaxe**

```
<boucle>::=
        do{
            <suite_instructions>
        }while(<condition>);
```

**Sémantique**

1. `<suite_instructions>` est exécutée
2. `<condition>` est une expression booléenne évaluée à la fin de chaque itération
   - si `<condition>` est vrai, le contrôle revient à `<suite_instructions>`
   - si `<condition>` est faux, le contrôle passe à l'instruction qui suit la boucle

# Exemple : saisie d'entiers(1/2)

Algorithme : saisie et affichage d'un nombre quelconque d'entiers

```
début
  faire
   X ← lire un entier;
   afficher X;
  tant que X>0;
  afficher "saisie terminée";
fin.
```

# Exemple : saisie d'entiers(2/2)

```
import java.util.Scanner;
public class Saisie{
  public static void main(String[] args){
    int X;
    Scanner in = new Scanner(System.in);
    System.out.println( "saisie d'entiers positifs" );
    do{
       X = in.nextInt();
       System.out.print( "la valeur saisie est : " );
       System.out.println(X);
    }while(X>0);
    System.out.println( "saisie terminée" );
  }
}
```

# Boucle `for` : syntaxe

```
<boucle_for>::=
   for( <init_inst >;<expr_bool>;<modif_inst>){
      <suite_instructions>
   }
```

où

`<init_inst>` sont les instructions d'initialisation de la boucle séparées par des virgules

`<expr_bool>` est une expression booléenne, condition de continuation de l'itération

`<modif_inst>` sont les instructions de mise à jour des variables de la boucle séparées par des virgules

# Boucle `for` : sémantique

- les instructions d'initialisation `<init_inst>` sont évaluées une seule fois avant le début des itérations
- `<expr_bool>` est évaluée avant le début de chaque itération

Si elle est vraie, on exécute :

1. les instructions `<suite_instructions>` du corps de la boucle
2. les mises à jour des variables `<modif_inst>`
3. puis de nouveau le test `<expr_bool>`

Si elle est fausse, la boucle termine

```
for(<init_inst >;<expr_bool>;<modif_inst>){
      <suite_instructions>
}
```

est équivalent à

```
<init_inst>;
while(<expr_bool>){
    <suite_instructions>;
    <modif_inst>;
}
```

# Exemple : calcul de la somme des n premiers entiers(1/2)

La boucle **for** est utilisée lorsque le nombre d'itérations est connu à l'avance (ne dépend pas d'une condition)

Algorithme de calcul de : 1+2+3+4+...+n

```
début
   n ← lire un entier;
   somme ← 0;
   pour i variant de 1 à n par pas de 1 faire
        somme ← somme+i;
   fin tant que;
   afficher somme;
fin.
```

# Exemple : calcul de la somme des n premiers entiers(2/2)

```
import java.util.Scanner;
public class nPremiersEntiers{
    public static void main(String[] args){
        System.out.print("taper un entier positif : ");
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int somme = 0;
        for(int i=1;i<=n;i++)
            somme = somme +i;
        System.out.print("somme = "+somme);
    }
}
```

i n'est utilisée que dans la boucle => variable locale

# Exemple: boucle for généralisée

```
public class IPlusJ{
 public static void main(String[] args){
  for( int i=0,j=0;(i+j<10);System.out.print(i+j+" "),i++,j++);
 }
}
```

résultat affiché

0 2 4 6 8

# Exemple : boucle infinie

```
for(;;){
   <suite_instructions>
}
```

=> boucle infinie

équivalent à

```
while(true){
   <suite_instructions>
}
```

# Rupture de contrôle : `break`

```
public class TestBreak{
    public static void main(String[] args){
    int n = 1;
    for(;;){
          System.out.println("A");
          if ( n>=1 ){
              System.out.println("B");
              break;
          }
          else
              System.out.println("C");
          System.out.println("D");
    }
   System.out.println("F");
  }
}
```

L'instruction **`break`** permet une sortie immédiate de la boucle la plus interne
et non d'un bloc quelconque

résultat affiché

```
A
B
F
```

# Cas d'utilisation

```
public class TestBreak2{
  public static void main(String[] args){
    int somme = 0;
    int n = 0;
    while( true ){
      n++;
      somme = somme+n;
      if( somme>100 ) break;
    }
    System.out.println("n="+n);
    System.out.println("somme="+somme);
  }
}
```

n=14
somme=105

# Instruction `continue`

```
public class TestContinue{
  public static void main(String[] args){
    int somme = 0;
    int n = 0;
    while( n<30 ){
        n++;
        if( n==10||n==12 ) continue;
        somme = somme+n;
    }
    System.out.println("n="+n);
    System.out.println("somme="+somme);
  }
}
```

L'instruction **`continue`** termine le pas d'itération en cours pour commencer le suivant.
Dans l'exemple, si n=10 ou 12, l'instruction `somme = somme +n;` n'est pas exécutée.
L'itération n'en est pas pour autant terminée.

```
n=30
somme=443
```

# Exemple : tirage du loto(1/2)

Algorithme : tirage aléatoire de 6 nombres compris entre 1 et 50.
Doublons possibles

```
début
  pour i variant de 1 à 6 par pas de 1 faire
   n ← valeur aléatoire entre 1 et 50;
   afficher n;
  fin pour;
fin.
```

# Exemple : tirage du loto(2/2)

```
import java.util.Random;
public class Loto{
  public static void main(String[] args){
       int n;
       System.out.println( "les numéros de la semaine !" );
       for(int i=1;i<=6;i++){
           Random alea = new Random();
           n = alea.nextInt(49)+1;
        System.out.print(n+" ");
    }
  }
}
```

un résultat affiché (des doublons sont possibles)

```
les numéros de la semaine !
31 49 46 29 17 30
```

# Affichage des table de multiplication

```
.........
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
 table de : 6
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
 table de : 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
.........
```

affichage partiel du résultat

# exemple : boucles imbriquées (1/2)

Algorithme : affichage des tables de multiplication de 2 à 9

```
début
  pour i variant de 2 à 9 par pas de 1 faire
    afficher "table de" i;
    afficher saut de ligne;
    pour j variant de 1 à 10 par pas de 1 faire
        afficher i "*" j "=" i*j;
        afficher saut de ligne;
        fin pour;
    afficher saut de ligne;
  fin pour;
fin.
```

# exemple : boucles imbriquées (2/2)

```
public class TablesMult{
  public static void main(String[] args){
    for( int i=2;i<=9;i++){
       System.out.println(" table de : "+ i);
       for( int j=1;j<=10;j++)
          System.out.println( i+ " * "+j+" = "+i*j);
    }
  }
}
```

# Notion de bloc (1/3)

- Un bloc est une suite d'instructions comprise entre accolades
- Le corps d'une boucle est donc un **bloc**
- Les instructions entre les accolades d'un **`if`**, d'un **`else`** sont des blocs
- Un bloc constitue un environnement local de déclarations de variables.

Les variables n'ont d'existence que dans le bloc dans lequel elles sont déclarées

```
public static void main(String[] args){
    int a=5;
    if (a==0){
        int b=3+2*a;
        System.out.println("b="+b);
    }
    else{
        int c=3+a;
        System.out.println("c="+c);
    }
    System.out.println("a="+a);
}
```

b est inconnu

b et c sont inconnus

# Notion de bloc (2/3)

Une variable déclarée dans l'en-tête d'une boucle **`for`** a pour portée le bloc qui suit

```
for( int i=0; i<10;i++){
   ...
   int i=7;
   ...
   ...
}
```

erreur : double déclaration de i dans le même bloc

# Notion de bloc (3/3)

Une variable peut-être déclarée plusieurs fois dans des blocs non imbriqués

```
public static void main( String[] args){
  int a=1;
  int b=5;
  for( int i =0;i<10;i++ ) a=a+b;
  for( int i =0;i<10;i++ ) b=b+a;
}
```

OK

Une variable ne peut-être déclarée qu'une seule fois dans des blocs imbriqués

```
public static void main( String[] args){
  int i=1;
  int b=0;
  for( int i =0;i<10;i++ ) b=b+i;
}
```

erreur