# Le langage Java - Syntaxe



MCours.com

# Les structures de contrôles

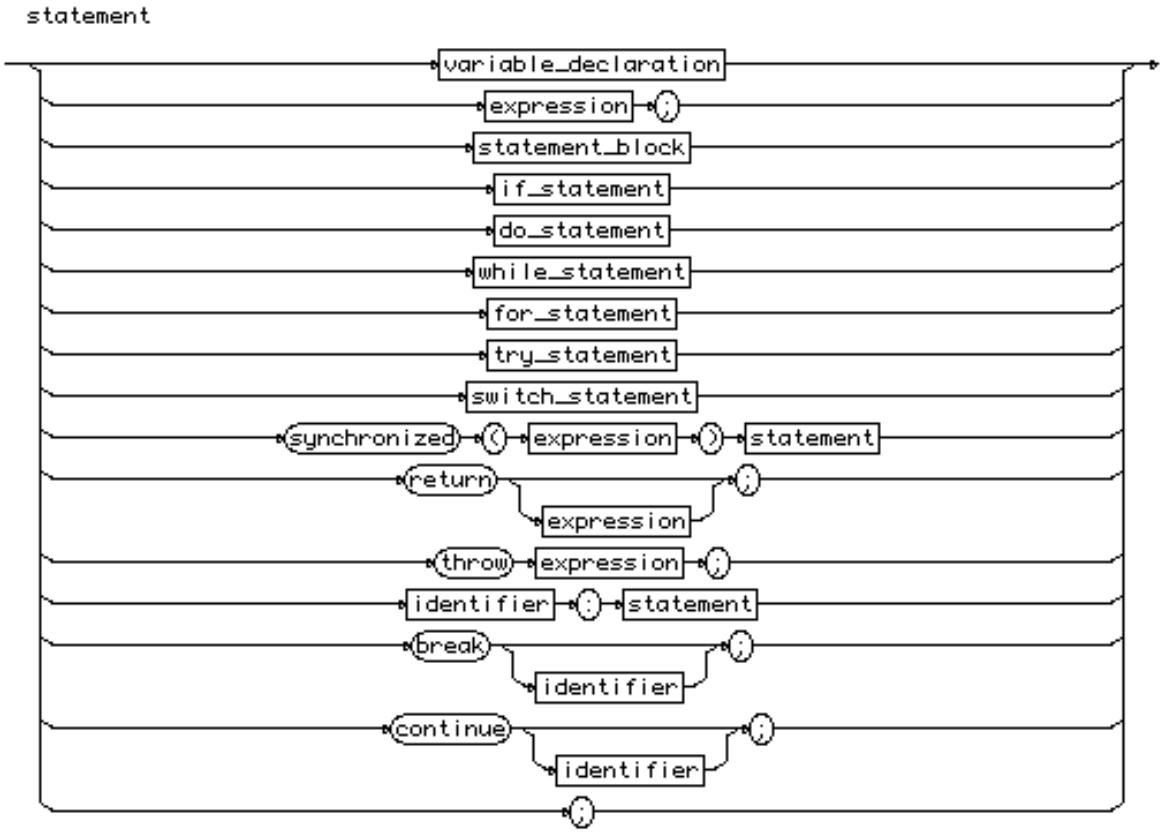
les différentes stuctures de contrôle:

le si-alors,

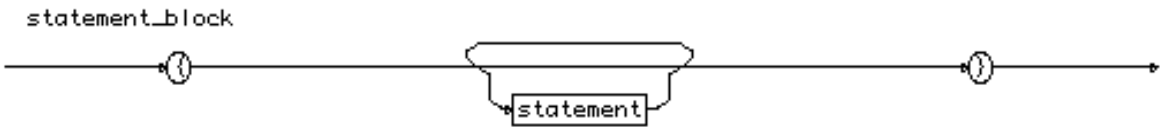
le faire-tantque,

le tantque-faire,

dans-le-cas



# Le bloc d’instructions

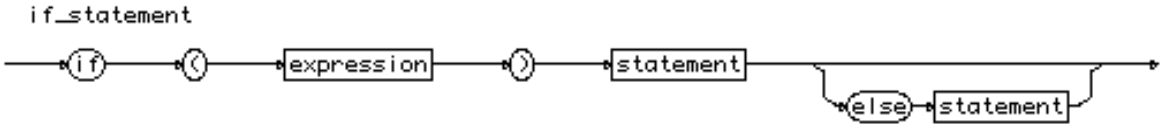



- contenu entre {}.
- définit la visibilité des variables.
- instruction = bloc d’instructions

# Exécution conditionelle

Il existe deux formes d’exécution conditionnel. le *si_alors_sinon* et le *dans_le_cas.*

# Si ... alors ... sinon




structure de base de la programmation

L'expression de test booléene (vraie ou fausse)

**if (e) S1;**

```
if (i==1) s=" i est égal à 1 ";
```

**if (e) S1 else S2;**

```
if (i==1) s=" i est égal à 1 ";
else s=" i est différent de 1 ";
```

De ces deux formes, nous pouvons dériver:

**if (e) {B1};**

```
if (i==1) {
   s=" i est égal à 1 ";
   i=j;
}
```

**if (e) {B1} else {B2};**

```
if (i==1) {
   s=" i est égal à 1 ";
   i=j;
}
else {
   s=" i est différent de 1 ";
   i=1515;
}
```

# cascader les conditions

**if (e1) S1 else if (e2) S2  ... else Sn;**

```
if (i==1)
    s=" i est égal à 1 ";
else if (i==2)
    s=" i est égal à 2 ";
else if (i==3)
    s=" i est égal à 3 ";
else s=" i est différent de 1,2 et 3 ";
```

L’utilisation de {} est nécessaire

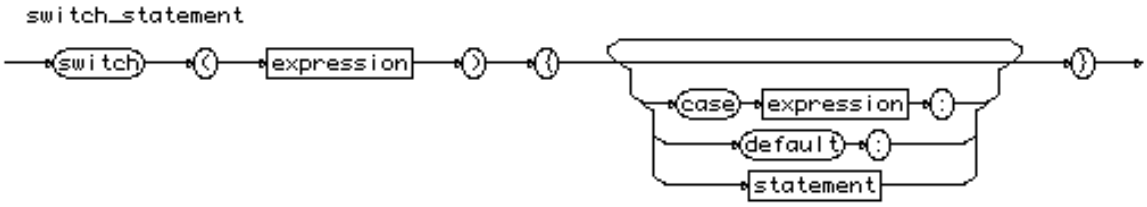
marquer le début est la fin des instructions;

```
i=0; j=3;
if (i==0)
   if (j==2)  s=" i est égal à 2 ";
else i=j;
System.out.println(i);
```

Que va imprimer le programme suivant 0 ou 3 ?

Il va imprimer 3! car i=0 et j!=2 alors on exécute i=j. Le *else* se rapporte en effet au *if* le plus imbriqué.

**au cas où**




MCours.com

*switch* = un aiguillage. limité *à char, byte, short ,int.*

- évalue l'expression qui lui est liée,
- compare aux *case*.
- commence à exécuter le code du switch OK
- exécute donc tous les instructions des cas suivants.

isoler chaque cas, il faut le terminer avec un *break*.

Si aucun cas , clause *default*

*case* et *defautl* sont considérer comme des étiquettes.

```
switch (e) {
case c1: S1;
case c2: S2;
...
case cn: Sn;
default: Sd
};
```

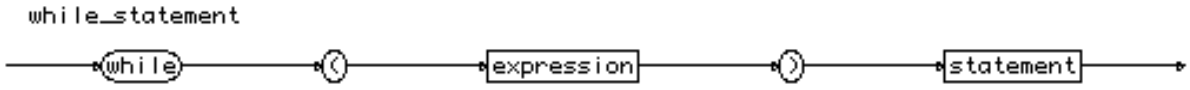
Examinons la forme générale de switch avec break;

```
switch (e) {
case c1: S1 break;
case c2: S2 break;
...
case cn: Sn break;
default: Sd
};
```

# au cas où ...

```
class TestSwitch{
    public static void main (String args[]) {
        int i=3;
        switch (i) {
          case 1: System.out.println("I"); break;
          case 2: System.out.println("II"); break;
          case 3: System.out.println("III"); break;
          case 4: System.out.println("IV"); break;
          case 5: System.out.println("V"); break;
          default: System.out.println(
                        "pas de traduction");
          }
    }
}
```

# tant que ... faire ...



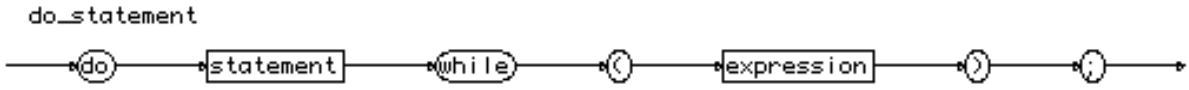

L'expression *e* doit être du type booléen.

**while (e) S1;**.

**while (e) {B1};**

```
class TestWhile{
    public static void main (String args[]) {
        int i=100, somme_i=0, j=0;
        // boucle 1: expression sans effet de bord
        while (j<=i) {
           somme_i+=j;
           ++j;
          }
        System.out.println("boucle  1:"+somme_i);
        // boucle 2: expression avec  effet de bord
        somme_i=0; j=0;
        while (++j<=i ) somme_i+=j;
        System.out.println("boucle  2:"+somme_i);
     }
}
```

# faire ... tant que ...



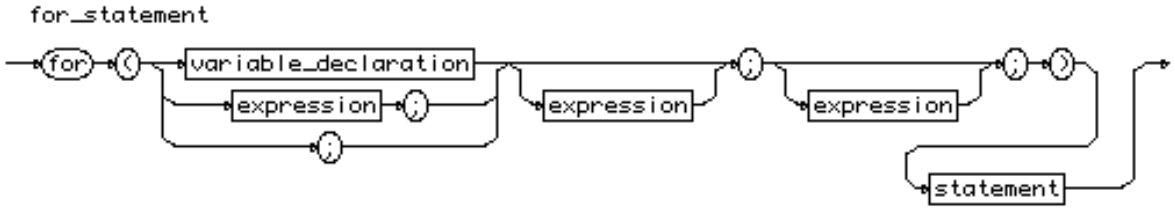

L'expression *e* doit être du type booléen.

**do S1 while(e);**

**do {B1} while(e);**

```
class TestDo{
    /* le résultat n'est pas erroné si l'on execute
       une addition de trop si i= 0 ! */
    public static void main (String args[]) {
        int i=100, somme_i=0, j=0;
        // boucle 1: expression sans effet de bord
        do {
           somme_i+=j;
           ++j;
          } while (j<=i);
        System.out.println("boucle  1:"+somme_i);
        // boucle 2: expression avec  effet de bord
        somme_i=0; j=0;
        do somme_i+=j; while (++j<=i );
          System.out.println("boucle  2:"+somme_i);
     }
}
```

# pour ... faire ...




basé sur un itérateur qui est contrôlé dans l'instruction

- expression initialise l'itérateur + le déclare
- expression teste si la condition d'achèvement
- expression modifie l'itérateur.

**for (e1;e2;e3) S1;**.

**for (e1;e2;e3) B1;**

Les expressions *e2* et *e3* sont optionnelles.

Le bloc B1doit

- modifier l'itérateur
- un *break* pour quitter la boucle.

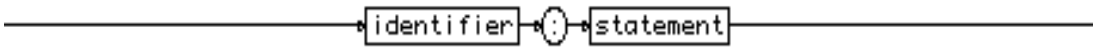
*for* est équivalent au *while*

**e1; while(e2) {S1; e3};**

# pour ... faire ...

```
class TestFor{
   public static void main (String args[]) {
        int n[] = new int[100];
      // boucle 1: calculer la somme des entiers pour
        // indice du vecteur
        for (int i=1; i<100; i++) n[i]=n[i-1]+i;
        // boucle 2: imprimer le résultat  par
        // ordre décroissant
        for (int i=99; i>=0;) {
            // modification de i dans le bloc d'in-
  structions
           System.out.println("somme("+i+")="+n[i]);
            i--;
        }
    }
}
```

# identifier une instruction

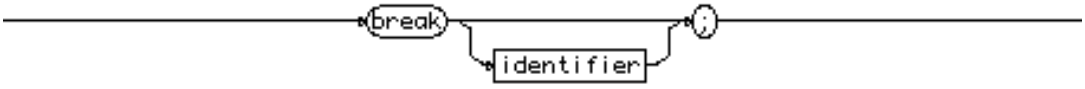



identifier une instruction ou un bloc d'instructions

- instructions de rupture
- continuation d'itération.

```
class LabelInst{
  public static void main (String args[]) {
     boucle1:
          for (int i=0;i<10;i++)
               for (int j=0; j<10;j++){
                    System.out.println(i+"/"+j);
               }
  }
}
```

# rupture




utilisé dans l'instruction *switch*, *while*, *do* et *for*

- quitter la boucle dans laquelle elle est comprise
- remonter de plusieurs niveaux

```
class TestBreak{
  public static void main (String args[]) {
     boucle1:
          for (int i=0;i<10;i++)
               for (int j=0; j<10;j++){
                    System.out.println(i+"/"+j);
                    // quitte la boucle locale
                    if (j==1) break;
                    // quitte la boucle for i ...
                    if (i==2) break boucle1;
               }
  }
}
```

exécution du programme TestBreak:

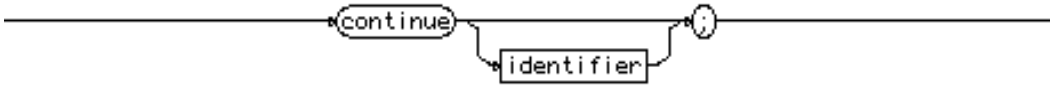
```
0/0
0/1
1/0
1/1
2/0
```

# rupture ...

Le break quitte le bloc complet d'instruction libellé.
les clauses *finally* sont exécutées.

```
class TestBreak2{
  public static void main (String args[]) {
     boucle1:{
          for (int i=0;i<10;i++)
               if (i==2) break boucle1;
          System.out.println("ne sera pas écrit");
     }
  }
}
```

# continuation




*continue* est utilisée dans *while*, *do* et *for*.

- interruption du déroulement normal de la boucle
- repris après la dernière instruction de la

d'associer une étiquette

```
class TestContinue{
  public static void main (String args[]) {
     boucle1:
          for (int i=0;i<10;i++)
               for (int j=0; j<10;j++){
                    // quitte la boucle locale
                    if (j>1) continue;
                    // quitte la boucle for i ...
                    if (i>2) continue boucle1;
                    System.out.println(i+"/"+j);
               }
      System.out.println("en dehors des boucles");
  }
}

exécution du programme TestContinue:
0/0
0/1
1/0
1/1
2/0
2/1
en dehors des boucles
```

MCours.com

# TestPremier2

Dans le programme TestPremier2, dans la boucle1, nous cherchons tous les nombres premiers jusqu'à 1100. Nous conservons ces nombres dans un vecteur p.

On remarquera :

- `i+=2;` permet d'incrémenter par deux.
- `j<i/j;` la condition d'arrêt évolue avec l'indice.
- `continue boucle1;` interrompt les tests dès qu'un diviseur de i a été trouvé.
- seuls sont conservés les nombres n'ayant pas quittés la boucle prématurément (donc ayant subit avec succés le test)

Dans la boucle2, nous cherchons les intervalles les plus grands sans nombre premier (entre 7 et 11, nous avons un *trou* de 4).

On remarquera :

- la boucle portant sur l'indice j test la primalité d'un nombre en utilisant uniquement les nombres stockés dans le vecteur p (crible d'Eratosthème).
- `p[j]<i/p[j];` la condition d'arrêt pour plus directement sur l'indice j, mais sur les valeurs référencées.
- `continue boucle2;` On quitte la boucle de test si l'on trouve un diviseur.
- quand on trouve un nombre premier, on test sa distance avec le précédent. Si elle est plus grande que la dernière trouvée, alors on l'imprime.

# TestPremier2 ...

```
class TestPremier2{
  public static void main (String args[]) {

  int p[]=new int[200];
  int dernierP=1;

     p[0]=2;
     p[1]=3;
     boucle1:
          for (int i=5;i<1100;i+=2){
               for (int j=3; j<i/j;j+=2)
                    if ((i%j)==0) continue boucle1;
    dernierP++;
       p[dernierP]=i;
          }
     System.out.println("P entre 2 et 1100: "+derni-
  erP);

    int dp=11, trou=4;
     boucle2:
    for (int i=13;i<1000000;i+=2){
               for (int j=1; p[j]<i/p[j];j++)
                    if ((i%p[j])==0) continue boucle2;
     if (i-dp>trou) {
       trou=i-dp;
       System.out.println(dp+".."+i+" = "+trou);
     }
     dp=i;
    }
 System.out.println("fin!");
  }
}
```

# TestPremier2 ...

L’exécution du programme TestPremier2:

```
P entre 2 et 1100: 193
31..37 = 6
89..97 = 8
139..149 = 10
199..211 = 12
293..307 = 14
887..907 = 20
1129..1151 = 22
1327..1361 = 34
9551..9587 = 36
15683..15727 = 44
19609..19661 = 52
31397..31469 = 72
155921..156007 = 86
360653..360749 = 96
370261..370373 = 112
492113..492227 = 114
fin!
```

# TestAmicaux

permet de trouver les nombres amicaux (un nombre dont la somme des diviseurs est égale à un autre nombre et vice-versa).

Exemple: 220 et 284 sont amicaux

somme_div(220)=1+2+4+5+10+11+20+22+44+55+110=284

somme_div(284)=1+2+4+71+142=220

Dans la boucle1, on calcule pour les nombres jusqu'à 10000, la valeur de la somme de leurs diviseurs que l'on mémorise dans un vecteur p. Le code est très semblable à celui de la recherche des nombres premiers sauf qu'ici nous voulons conserver la valeur des diviseurs.

Dans la boucle2, on cherche les nombres amicaux.

On remarquera :

- `((p[i]<10000)&&(p[p[i]]==i));` la condition du test avec abandon, ceci permet de test si la somme des diviseurs est plus grande que 9999, alors on effectue pas le test d'amitié entre les nombres (dans le cas contraire, il ne faut pas effectuer ce test car on sort des valeurs autoriseés pour le vecteur p.

# TestAmicaux ...

```
class TestAmicaux{
  public static void main (String args[]) {
  int p[]=new int[10000];
  int dernierP=1;

    p[1]=1;  p[2]=1;
    p[3]=1;  p[4]=3;
    p[5]=1;  p[6]=6;

        // boucle 1
        for (int i=7;i<10000;i++)
            for (int j=1; j<=i/2;j++)
                if ((i%j)==0) p[i]+=j;

        // boucle 2
        for (int i=2;i<10000;i++)
                if ((p[i]<10000)&&(p[p[i]]==i))
      System.out.println(i+" "+p[i]);

   System.out.println("fin!");
  }
}
```

# TestAmicaux

L’exécution du programme TestAmicaux:

```
6 6
28 28
220 284
284 220
496 496
1184 1210
1210 1184
2620 2924
2924 2620
5020 5564
5564 5020
6232 6368
6368 6232
8128 8128
fin!
```

Les nombres qui sont amicaux avec eux-mêmes sont des nombres parfaits! (comme 6, 28, ...).

## mais où sont les objets?

MCours.com