



## TD Boucles

### Résumé

Voyons ici comment incorporer des boucles, les structures répétitives, dans nos codes et comment les utiliser à bon escient.

<b>1</b>	<b>Les boucles</b>	<b>2</b>
1.1	tant que . . . . .	2
1.2	while . . . . .	4
1.3	faire - tant que . . . . .	6
1.4	do...while . . . . .	8
1.5	pour . . . . .	9
1.6	for . . . . .	12
1.7	Quel type de boucle choisir ? . . . . .	13
1.8	Sentinelle . . . . .	13
1.9	suite de nombres . . . . .	14
1.10	3 pas en avant, 2 pas en arrière . . . . .	15
<b>2</b>	<b>La gestion des erreurs : try - catch</b>	<b>16</b>
2.1	L'instruction try catch . . . . .	16
<b>3</b>	<b>Exercices</b>	<b>17</b>
3.1	Compréhension d'algorithme . . . . .	17
3.2	Compréhension de codes Java . . . . .	19
3.3	À vous de jouer... . . . . .	24



# 1 Les boucles

Si on veut faire effectuer un travail répétitif, il faut indiquer deux choses :

1. Le travail à répéter
2. La manière de continuer la répétition ou de l'arrêter.

## 1.1 tant que

Le «**tant que** » est une structure qui demande à l'exécutant de répéter une tâche (une ou plusieurs instructions) tant qu'une condition donnée est vraie.

En pseudo-code :

```
tant que condition faire
    séquence d'instructions à exécuter
fin tant que
```

La **condition** est une expression délivrant un résultat **booléen** (vrai ou faux).

Il faut qu'il y ait dans la séquence d'instructions comprise entre **tant que** et **fin tant que** au moins **une instruction qui modifie** une des composantes de la **condition** de telle manière qu'elle puisse **devenir fausse** à un moment donné. Dans le cas contraire, la condition reste **indéfiniment vraie** et la boucle va tourner sans fin, c'est ce qu'on appelle une **boucle infinie**.

Si la condition est fausse dès le début, la tâche n'est jamais exécutée.

Par exemple :

### Afficher les nombres plus petits que 10

On affiche uniquement les nombres inférieurs (pas strictement) à 10 .

```
// Affiche les nombres de 1 à 10.
algorithme compterJusque10 () // version avec tant que
    nb : entier
    nb ← 1 // c'est le premier nombre à afficher
    tant que nb ≤ 10 faire // pour aller jusqu'à 10
        afficher nb // on affiche la valeur de la variable nb
        nb ← nb + 1 // on passe au nombre suivant
```

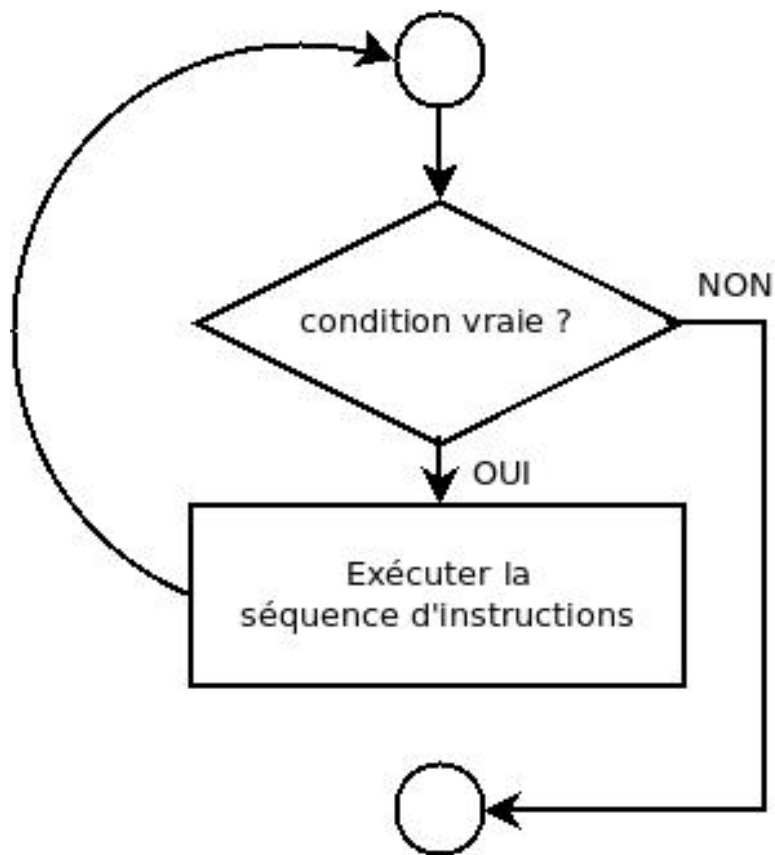


FIGURE 1 – boucleTq.jpg

```

    fin tant que
fin algorithme

```

### Somme de nombres

Si l'utilisateur encode une valeur négative l'algorithme arrête de demander une valeur .

```

// Lit des valeurs entières et retourne la somme des valeurs lues.
algorithme sommeNombres() → entier
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
    somme ← 0
    demander valeur
    tant que valeur ≥ 0 faire

```

```

    somme ← somme + valeur
    demander valeur // remarquer l'endroit où on lit une valeur.
fin tant que
retourner somme
fin algorithme

```

## 1.2 while

L'équivalent Java de la boucle

```

tant que condition faire
    séquence d'instructions à exécuter
fin tant que

```

est

```

while ( condition ) {
    séquence d'instructions à exécuter
}

```

### Les puissances de 2

```

public class Util {
    public static void puissance2() {
        int puissance = 1;
        while ( puissance < 100 ) {
            System.out. print(puissance + " ");
            puissance = 2 * puissance ;
        }
    }
}

```

affichera 1 2 4 8 16 32 64

### Somme d'entiers positifs

```

import java. util .Scanner;
public class Exemple {
    /**
     * Affiche la somme d entiers positifs entres au clavier .
     * S arrete des qu une valeur nulle ou negative est donnee.
     * @param args non utilise
     */
}

```

```

public static void main(String [] args) {
    Scanner clavier = new Scanner(System.in);
    int nb;
    int somme = 0;
    nb = clavier . nextInt ();
    while ( nb > 0 ) {
        somme = somme + nb;
        nb = clavier . nextInt ();
    }
    System.out. println (somme);
}

```

affichera 0 si le premier nb lu vaut -5.

### Un exemple complet

```

import java. util .Scanner;
public class Lecture {
    /**
     * Lit un entier au clavier .
     * Les valeurs non entieres sont passees .
     * @return l entier lu .
     */
    public static int demanderEntier () {
        Scanner clavier = new Scanner(System.in);
        int nb;
        // Tant que ce n est pas un entier au clavier
        while ( ! clavier .hasNextInt() ) {
            clavier .next (); // le demander , le passer
        }
        nb = clavier . nextInt ();
        return nb;
    }

    /**
     * Lit un entier au clavier .
     * Les valeurs non entieres , nulles ou negatives sont passees .
     * @return l entier lu .
     */
    public static int demanderPositif () {
        int nb;
        nb = demanderEntier ();
        while (nb<=0) {
            nb = demanderEntier ();
        }
        return nb;
    }

    /**
     * Un exemple de main.
     */
    public static void main(String [] args){
        int nombreLu;

        System.out. print ( "Entrez_un_entier_positif:_");
        nombreLu = demanderPositif ();
    }
}

```

```
}  
}
```

### 1.3 faire - tant que

```
faire  
    séquence d'instructions à exécuter  
tant que condition
```

Cette structure est très proche du «**tant que**» à une différence près :

- Le **test** est fait **à la fin** et pas au début. La tâche est donc toujours **exécutée au moins une fois**.

En pseudo-code :

```
faire  
    séquence d'instructions à exécuter  
tant que condition
```

La **condition** est une expression délivrant un résultat **booléen** (vrai ou faux).

Il faut que la séquence d'instructions comprise entre **faire** et **tant que** contienne au moins **une instruction qui modifie la condition** de telle manière qu'elle puisse **devenir fausse** à un moment donné pour arrêter l'itération.

La tâche est toujours **exécutée au moins une fois**.

Par exemple :

#### Somme de nombres

Après chaque nombre, on demande à l'utilisateur s'il y a encore un nombre à additionner.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.  
algorithme sommeNombres() → entier  
    encore : booléen // est-ce qu'il reste encore une valeur à additionner ?  
    valeur : entier // un des termes de l'addition  
    somme : entier // la somme
```

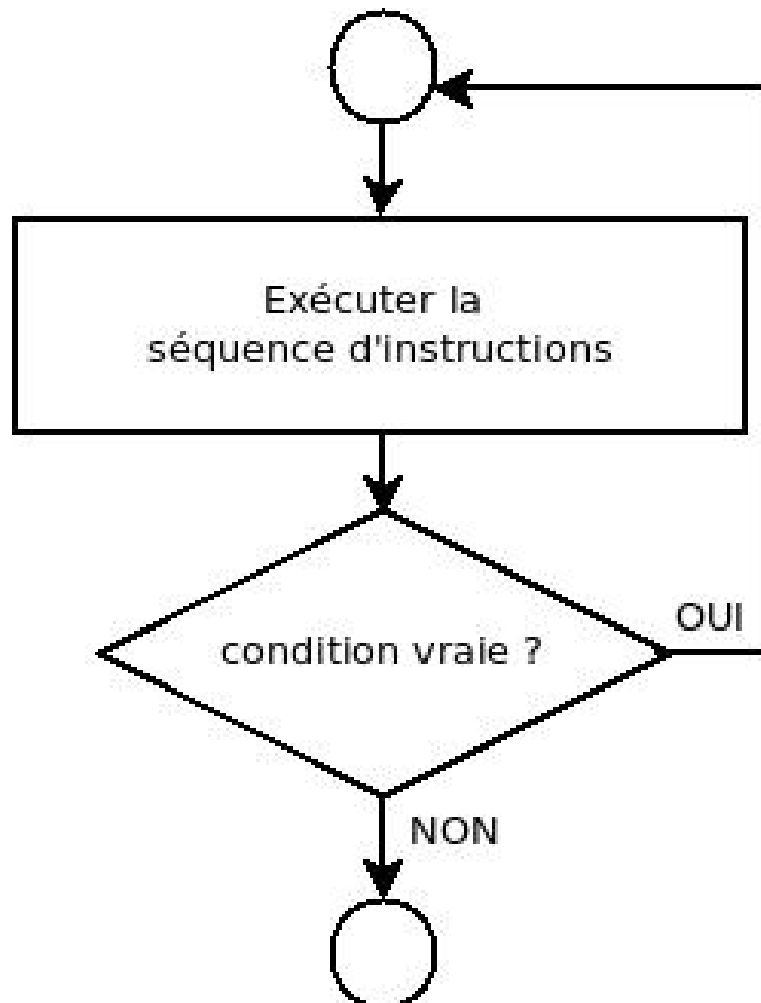


FIGURE 2 – boucleFaire.jpg

```
somme ← 0
faire
    demander valeur
    somme ← somme + valeur
    demander encore
tant que encore
retourner somme
fin algorithme
```

Avec cette solution, on additionne au moins une valeur.

## 1.4 do...while

L'équivalent Java de la boucle

```
faire
    séquence d'instructions à exécuter
tant que conditionPourResterDansLaBoucle

est

do {
    séquence d'instructions à exécuter
} while( conditionPourResterDansLaBoucle );
```

### Les puissances de 2

```
public class Util {

public static void puissance2() {
    int puissance = 1;
    do {
        System.out. print(puissance + " ");
        puissance = 2 * puissance ;
    } while ( puissance < 100 );
}
```

affichera 1 2 4 8 16 32 64

### Somme d'entiers positifs

```
import java. util .Scanner;
public class Exemple {
    /**
     * Affiche la somme d entiers positifs entres au clavier .
     * S arrete des qu une valeur nulle ou negative est donnee.
     * @param args non utilise
     */
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb;
        int somme = 0;
        do{
            nb = clavier . nextInt ();
            somme = somme + nb;
        } while ( nb > 0 );
        System.out. println (somme);
    }
}
```

affichera -5 si le premier nb lu vaut -5.



## 1.5 pour

On va indiquer combien de fois la tâche doit être répétée. Cela se fait au travers d'une **variable de contrôle** dont la valeur va évoluer **à partir d'une valeur de départ jusqu'à une valeur finale**.

En pseudo-code :

```
pour variable de début à fin [par pas] faire
    séquence d'instructions à exécuter
fin pour
```

est équivalent à

```
{
variable : entier
variable ← début
tant que variable ≤ fin faire
    séquence d'instructions à exécuter
    variable ← variable + pas // ou variable ← variable + 1 si le pas est omis.
fin tant que
}
```

Dans ce type de structure, **début**, **fin** et **pas** peuvent être des constantes, des variables ou des expressions (le plus souvent à valeurs entières mais on admettra parfois des réels).

Le **pas** est facultatif, et généralement omis (dans ce cas, sa valeur par défaut est 1).

Ce **pas** est parfois négatif, dans le cas d'un compte à rebours, par exemple pour **n** de 10 à 1 par -1.

1. Quand le **pas** est positif, la boucle s'arrête lorsque la **variable** dépasse la valeur de **fin**.
2. Par contre, avec un **pas** négatif, la boucle s'arrête lorsque la **variable** prend une valeur plus petite que la valeur de **fin**.

On considérera qu'au cas (à éviter) où

1. **début** est strictement supérieur à **fin** et le **pas** est positif, la séquence d'instructions n'est jamais exécutée (mais ce n'est pas le cas dans tous les langages de programmation!).

2. Idem si **début** est strictement inférieur à **fin** mais avec un **pas** négatif.

Attention de **ne pas modifier** dans la séquence d'instructions une des variables de contrôle **début**, **fin** ou **pas** !

Il est aussi fortement **déconseillé de modifier «manuellement»** la variable au sein de la boucle **pour**. Il ne faut pas l'initialiser en début de boucle, et ne pas s'occuper de sa modification, l'instruction **variable**  $\leftarrow$  **variable** + **pas** étant automatique et implicite à chaque étape de la boucle.

Il est aussi déconseillé d'utiliser **variable** à la sortie de la structure **pour** sans lui affecter une nouvelle valeur.

Par exemple :

```
// Affiche les nombres de 1 à 10.
algorithme compterJusque10 () // version avec pour
    pour nb de 1 à 10 faire // par défaut le pas est de 1
        afficher nb
    fin pour
fin algorithme
```

### Afficher les nombres plus petits que n

On affiche uniquement les nombres inférieurs (pas strictement) à n.

```
// Reçoit un nombre et affiche les nombres de 1 à ce nombre.
algorithme afficherN (n↓ : entier)
    pour nb de 1 à n faire
        afficher nb
    fin pour
fin algorithme
```

### Afficher les nombres pairs plus petits que 10

On affiche uniquement les nombres pairs jusqu'à 10.

```
// Reçoit un nombre et affiche les nombres pairs jusqu'à ce nombre.
// n : limite des nombres à afficher.
Exemple : si n vaut 10, les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.
algorithme afficherPair (n↓ : entier)
    pour nb de 2 à n par 2 faire
        afficher nb
```

```

    fin pour
fin algorithme

```

### Afficher les nombres pairs plus petits que n

On affiche uniquement les nombres pairs jusqu'à la limite n.

```

// Reçoit un nombre et affiche les nombres pairs jusqu'à ce nombre.
// n : limite des nombres à afficher.
// Exemple : si n vaut 10, les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.
algorithme afficherPair (n↓ : entier)
    pour i de 1 à n DIV 2 faire
        afficher 2 * i
    fin pour
fin algorithme

```

### Afficher n nombres pairs

On affiche les n premiers nombres pairs.

```

// Reçoit un nombre et affiche ce nombre de nombres pairs.
// n: le nombre de nombres à afficher.
// Exemple : si n vaut 10, les 10 premiers nombres pairs sont : 2, 4, 6, 8, 10, 12,
algorithme afficherPair ()
    pour i de 1 à n faire
        afficher 2 * i
    fin pour
fin algorithme

```

### Somme de nombres

L'utilisateur indique le nombre de termes au départ.

```

// Lit des valeurs entières et retourne la somme des valeurs lues.
algorithme sommeNombres () → entier
    nbValeurs : entier // nb de valeurs à additionner
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
    somme ← 0 // la somme se construit petit à petit. Elle vaut 0 au départ
    demander nbValeurs

```

```

    pour i de 1 à nbValeurs faire
        demander valeur
        somme ← somme + valeur
    fin pour
    retourner somme
fin algorithme

```

## 1.6 for

L'équivalent Java de la boucle

```

pour variable de début à fin [par pas] faire
    séquence d'instructions à exécuter
fin pour

```

est

```

for ( int i = début; i <= fin; i = i+pas ) {
    séquence d'instructions à exécuter
}

```

### Les nombres pairs entre 2 et 100

```

public class Exemple {
    /**
     * Affiche la somme des nombres pairs entre 2 et 100 inclus.
     * @param args non utilise
     */
    public static void main(String [] args) {
        int somme;
        somme = 0;
        for ( int i = 2; i <= 100; i = i+2 ) {
            somme = somme + i;
        }
        System.out. println (somme);
    }
}

```

### Compte à rebours

```

public class Exemple {

```

```

/**
 * Affiche un compte a rebours a partir de 10.
 * @param args non utilise
 */
public static void main(String [] args) {
    for ( int i = 10; i >= 1; i = i-1 ) {
        System.out. println ( i );
    }
    System.out. println ("Partez !");
}
}

```

`i++`; est un raccourci pour `i = i+1`;

```

public class Exemple {
    /**
     * Affiche les nombres de 1 a 10 inclus.
     * @param args non utilise
     */
    public static void main(String [] args) {
        for ( int i = 1; i <= 10; i++ ) {
            System.out. println ( i );
        }
    }
}

```

## 1.7 Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle **tant que** qui peut s'adapter à toutes les situations. Cependant,

- il est plus clair d'utiliser la boucle **pour** dans les cas où le nombre d'itérations est fixé et connu à l'avance (par là, on veut dire que le nombre d'itérations est déterminé au moment où on arrive à la boucle).
- La boucle **faire** convient quant à elle dans les cas où le contenu de la boucle doit être parcouru au moins une fois,
- alors que dans **tant que**, le nombre de parcours peut être nul si la condition initiale est fausse.

## 1.8 Sentinelle

Quand l'utilisateur entre une valeur spéciale pour indiquer la fin (d'une boucle **tant que** ou **faire**), on parle de **valeur sentinelle**. Ceci n'est possible que si cette valeur sentinelle ne peut pas être un terme valide. Par

exemple, si on veut additionner des nombres positifs uniquement, la valeur -1 peut servir de valeur sentinelle. Mais sans limite sur les nombres à additionner (positifs, négatifs ou nuls) il n'est pas possible de choisir une sentinelle.

## 1.9 suite de nombres

Un exemple simple pourrait être celui-ci : « *Écrire l'algorithme qui affiche les  $n$  premiers termes de la suite : 2, 4, 6. . .* »

Puisqu'on doit écrire plusieurs nombres et qu'on sait exactement combien, on se tournera tout naturellement vers une boucle **pour**.

Le cas le plus simple est lorsque le nombre à afficher à l'étape  $i$  peut être calculé en fonction de  $i$  seulement. L'algorithme est alors

```
pour i de 1 à n faire
    afficher f(i)
fin pour
```

Par exemple, pour afficher la suite des  $n$  premiers nombres pairs

```
algorithme nombrePair ( $n \downarrow$  : entier)
    pour i de 1 à n faire
        afficher 2 * i
    fin pour
fin algorithme
```

Parfois, il est difficile (voire impossible) de trouver  $f(i)$ .

On suivra alors une autre approche qui revient à calculer un nombre à afficher à partir du nombre précédemment affiché (ou, plus exactement, de calculer le suivant à partir du nombre qu'on vient d'afficher).

La structure générale est alors

```
nb ← {1re valeur à afficher}
pour i de 1 à n faire
    afficher nb
    nb ← {calculer ici le nb suivant}
fin pour
```

Dans l'exemple de la suite paire, le 1er nombre à afficher est 2 et le nombre suivant se calcule en ajoutant 2 au nombre courant.

```
algorithme suite1 (n↓ : entier)
  nb : entier
  nb ← 2
  pour i de 1 à n faire
    afficher nb
    nb ← nb + 2
  fin pour
fin algorithme
```

### 1.10 3 pas en avant, 2 pas en arrière

Dans certains cas, il n'est pas possible de déduire directement le nombre suivant en connaissant juste le nombre précédent. Prenons un exemple un peu plus compliqué pour l'illustrer. «*Écrire l'algorithme qui affiche les  $n$  premiers termes de la suite : 1, 2, 3, 4, 3, 2, 3, 4, 5, 4, 3. . .*»

Si on vient d'écrire, disons un 3, impossible sans information supplémentaire, de connaître le nombre suivant. Il faudrait savoir si on est en phase d'avancement ou de recul et combien de pas il reste à faire dans cette direction.

Ajoutons des variables pour retenir l'état où on est.

```
algorithme suite3Avant2Arrière(n↓ : entier)
  nb, nbPasRestants, direction : entiers
  nb ← 1
  nbPasRestants ← 3 // 3 pas
  direction ← 1 // en avant
  pour i de 1 à n faire
    afficher nb
    nb ← nb + direction // faire un pas dans la bonne direction
    nbPasRestants ← nbPasRestants - 1
    si nbPasRestants = 0 alors // il est temps de changer de direction
      direction ← -direction
      si direction = 1 alors
        nbPasRestants ← 3
      sinon
        nbPasRestants ← 2
    fin si
  fin si
```

```
    fin pour
fin algorithme
```

On obtient un algorithme plus long mais qui respecte toujours le schéma vu.

Un conseil : essayez de respecter ce schéma et vous obtiendrez plus facilement un algorithme correct et lisible, également dans les cas particuliers.

## 2 La gestion des erreurs : try - catch

### 2.1 L'instruction try catch

- **try** : contient les instructions qui peuvent mal se passer
- **catch** : contient le code qui est en charge de gérer le problème

Quand un problème se présente dans le try

- Le code du **try** est interrompu
- Le code du **catch** est exécuté
- Ensuite, on continue après le **try-catch**

Exemple :

```
package be.heb.esi.lgj1 ;
import java.util.Scanner;
public class Affiche {
    /**
     * Affiche l ' entier lu au clavier ou un message si ce n ' est pas un entier .
     * @param args inutilisé .
     */
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb;
        try {
            nb = clavier . nextInt ();
            System.out. println (nb);
        }
        catch(Exception e) {
            System.out. println ("Ce_n'est_pas_un_entier!");
        }
    }
}
```



### 3 Exercices

Maintenant, mettons tout ça en pratique.

#### 3.1 Compréhension d'algorithme

Pour ces exercices, nous vous demandons de comprendre des algorithmes donnés.

##### Compréhension

Que vont-ils afficher ?

```
— algorithme boucle1 ()
    x : entier
    x ← 0
    tant que x < 12 faire
        x ← x+2
    fin tant que
    afficher x
finalgorithme

— algorithme boucle2 ()
    ok : booléen
    x : entier
    ok ← vrai
    x ← 5
    tant que ok faire
        x ← x+7
        ok ← x MOD 11 ≠ 0
    fin tant que
    afficher x
fin algorithme

— algorithme boucle3 ()
    ok : booléen
    cpt, x : entiers
    x ← 10
    cpt ← 0
    ok ← vrai
    tant que ok ET cpt < 3 faire
        si x MOD 2 = 0 alors
            x ← x+1
        ok ← x < 20
```

```

        sinon
            x ← x+3
            cpt ← cpt + 1
        fin si
    fin tant que
    afficher x
fin algorithme

—
— algorithme boucle4 ()
    pair, grand : booléens
    p, x : entiers
    x ← 1
    p ← 1
    faire
        p ← 2*p
        x ← x+p
        pair ← x MOD 2 = 0
        grand ← x > 15
    tant que NON pair ET NON grand
    afficher x
fin algorithme

—
— algorithme boucle5 ()
    i, x : entiers
    ok : booléen
    x ← 3
    ok ← vrai
    pour i de 1 à 5 faire
        x ← x+i
        ok ← ok ET (x MOD 2 = 0)
    fin pour
    si ok alors
        afficher x
    sinon
        afficher 2 * x
    fin si
fin algorithme

—
— algorithme boucle6 ()
    i, j, fin : entiers
    pour i de 1 à 3 faire
        fin ← 6 * i - 11
        pour j de 1 à fin par 3 faire
            afficher 10 * i + j

```

```
    fin pour
  fin pour
fin algorithme
```

---

## 3.2 Compréhension de codes Java

### Instructions répétitives

Quelles instructions répétitives sont correctes parmi les suivantes ? Expliquez pourquoi les autres ne le sont pas.

- ☐ proposition 1

```
While ( condition ) {  
    // instructions  
}
```

- ☐ proposition 2

```
do while ( condition ) {  
    // instructions  
}
```

- ☐ proposition 3

```
while ( true ) {  
    // instructions  
}
```

- ☐ proposition 4

```
while ( true ) do {  
    // instructions  
}
```

- ☐ proposition 5

```
FOR ( int i=0; i<=10; i=i+2 ) DO {  
    // instructions  
}
```

- ☐ proposition 6

```
for ( int i=0; i<=10; i=i+2 ) {  
    // instructions  
}
```

- ☐ proposition 7

```
for ( int i=0; i<=10; i=i+2 ) do {  
    // instructions  
}
```

- ☐ proposition 8

```
for ( int i=9; i>=0; i=i-2 ) {  
    // instructions  
}
```

### Activité 'remplir les blancs'

Quel opérateur de comparaison Java représente la relation suivante ?

1. "est égal à" ? \_\_\_\_
2. "est différent de" ? \_\_\_\_

Quel opérateur booléen Java représente l'opérateur logique suivant ?

1. le ET : \_\_\_\_
2. le OU : \_\_\_\_
3. le NON : \_\_\_\_

### Expérience

Indiquez l'affichage obtenu par ce code.

Que vont-ils afficher ?

```
public class Boucles {  
  
    public static void main ( String[] args ) {  
        int facteur;  
        final int VALEUR = 3;  
  
        for (facteur = 1 ; facteur <= 10 ; facteur++){  
            System.out.print(facteur*VALEUR+" ");  
        }  
        System.out.println();  
    }  
}
```

---

### Exercice Tant que

Écrivez en Java l'algorithme suivant.

algorithme Test

```

    nb, produit : Entier
    produit ← 1

    demander nb
    tant que nb ≠ 0 faire
        produit ← produit * nb
        demander nb
    fin tant que
    afficher produit

fin algorithme

```

### Exercice Pour

Écrivez en Java l'algorithme suivant.

```

algorithme Test

    nb: Entier
    i : Entier

    demander nb
    pour i de 1 à nb faire
        afficher i
    fin pour

fin algorithme

```

### Mots clés

Complétez le morceau de code suivant afin de pouvoir attraper l'exception de type `IllegalArgumentException` susceptible d'être lancée par la méthode `maxPositif` de la même classe.

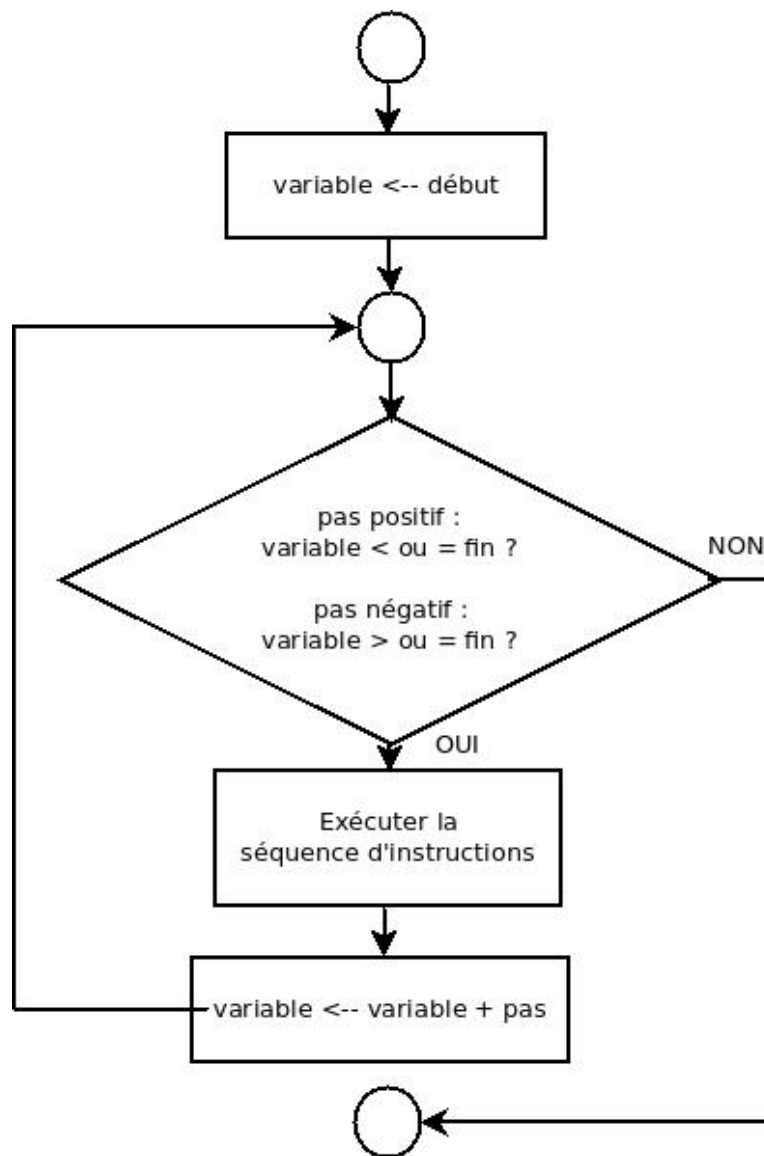


FIGURE 3 – bouclePour.jpg

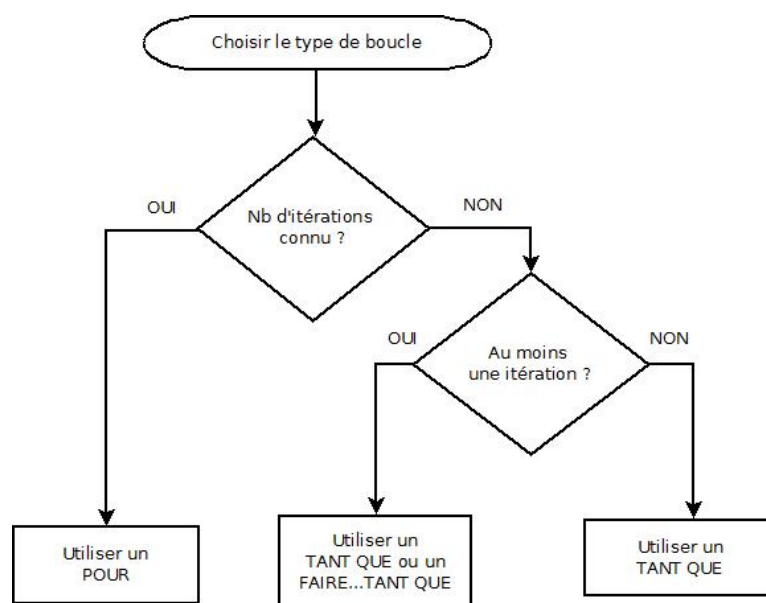


FIGURE 4 – boucleChoixType.jpg

```

public static void appelMaxPositif() {
    Scanner clavier = new Scanner(System.in);

    _____ {
        int max = maxPositif(clavier.nextInt(), clavier.nextInt());
    } _____ (IllegalArgumentException ex) {
        // gestion de l'exception
    }
}

```

### 3.3 À vous de jouer...

Voici quelques conseils pour vous guider dans la résolution de tels problèmes :

- il convient d'abord de bien comprendre le problème posé ; assurez-vous qu'il est parfaitement spécifié ;
- résolvez le problème via quelques exemples précis ;
- mettez en évidence les variables «**données** », les variables «**résultats** » et les variables de travail ;
- n'hésitez pas à faire une ébauche de résolution en français avant d'élaborer l'algorithme définitif pseudo-codé ;
- déclarez ensuite les variables (et leur type) qui interviennent dans chaque algorithme ; les noms des variables risquant de ne pas être suffisamment explicites.
- Écrivez la partie algorithmique **AVANT** de vous lancer dans la programmation en Java.
- Pour la partie Java, dessinez l'arborescence des fichiers, vous travaillerez dans un package `g12345.Boucles`.

Écrivez les algorithmes et codez les programmes Java correspondant qui

1. reçoit un naturel `n` et affiche
  - (a) les `n` premiers entiers strictement positifs ;
  - (b) les `n` premiers entiers strictement positifs en ordre décroissant ;
  - (c) les `n` premiers carrés parfaits ;
  - (d) les `n` premiers naturels impairs ;
  - (e) les naturels impairs qui sont inférieurs ou égaux à `n`.

Si le `n` reçu n'est pas strictement positif, votre programme générera une erreur/une exception qui sera gérée par un affichage explicite dans la méthode `main`.



2. demande à l'utilisateur d'introduire un entier positif (non strictement). Cet algorithme permet à l'utilisateur de se tromper à plusieurs reprises mais l'utilisateur devra donner une bonne valeur pour arrêter le programme. (On suppose tout de même que l'utilisateur ne donne que des valeurs entières!)
3. lit une série de nombres entiers positifs, jusqu'à ce que l'utilisateur encode la valeur 0. Les nombres multiples de 3 seront affichés au fur et à mesure et le nombre de ces multiples sera affiché en fin de traitement.

Pensez à utiliser l'algorithme écrit ci-dessus qui permet de demander un entier positif.

4. retourne la somme des chiffres qui forment un nombre naturel **n** Attention, on donne au départ le nombre et pas ses chiffres. Exemple : 133045 doit donner comme résultat 16, car  $1 + 3 + 3 + 0 + 4 + 5 = 16$ .
5. lit une suite de nombres positifs entrés au clavier et affiche le maximum, le minimum, leur somme et la moyenne.  
La fin de la suite de nombre sera signifiée par une valeur sentinelle que vous choisirez judicieusement.
6. vérifie si un entier donné forme un palindrome ou non. Un nombre palindrome est un nombre qui lu dans un sens (de gauche à droite) est identique au nombre lu dans l'autre sens (de droite à gauche). Par exemple, 1047401 est un nombre palindrome.
7. affiche les **n** premiers termes de la suite suivante : 1, -1, 2, -3, 5, -8, 13, -21, 34, -55, où **n** est un entier strictement positif reçu en paramètre.

Si le **n** reçu n'est pas strictement positif, votre programme générera une erreur/une exception qui sera gérée par un affichage explicite dans la méthode **main**.

En java, n'oubliez pas d'écrire la javadoc et la méthode **main** pour tester vos méthodes.

### Jeu de la fourchette

Écrivez un algorithme qui simule le jeu de la fourchette. Ce jeu consiste à essayer de découvrir un nombre quelconque compris entre 0 et 100 inclus, tiré au sort par l'ordinateur (la primitive **hasard(n : entier)** retourne un entier entre 1 inclus et n exclus).

L'utilisateur a droit à huit essais maximum.

À chaque essai, l'algorithme devra afficher un message indicatif

— «nombre donné trop petit »

— ou «nombre donné trop grand ».

En conclusion,

- soit «bravo, vous avez trouvé en [nombre] essai(s) »
- soit «désolé, le nombre était [valeur] ».

Écrivez le code java correspondant.

Aide en Java : un petit tour dans l'API de la classe Random devrait vous aider à trouver l'équivalent du `hasard(n : entier)` en Java