Chapitre 5

Compléments sur l'itération

Nous conaissons déjà superficiellement les boucles `while` et `for`. Dans ce chapitre nous allons rentrer plus en détail dans la description de ces notions.

5.1 les notions de base pour l'élaboration des boucles

Toute instruction d'itération possède une condition booléenne dite *d'arrêt*, et une suite d'instructions à répéter, qu'on appelle *corps* de la boucle. Lors de chaque itération, la condition d'arrêt est testée, et selon sa valeur, on décide de répéter une nouvelle fois ou pas, les instructions du corps.
Problème 1 : Reprenons notre problème consistant à calculer la somme d'une suite de nombres entiers saisis au clavier. Le calcul s'arrête lors de la saisie du nombre zéro. Pour réaliser le calcul, nous allons répéter la saisie d'un nombre n, puis son ajout à une variable `total` qui accumule la somme de tous les nombres saisis.

Entrées : n entier à saisir
Sortie : $total$ entier
Algorithme :

1. Initialiser n par une saisie,
2. Initialiser $total$ avec 0,
3. Tant que $n \neq 0$ faire :
   (a) $total \leftarrow total + n$
   (b) Saisir n
4. Afficher la valeur de $total$.

Le pas 3 de cet algorithme est une boucle :

Tant que $n \neq 0$, faire : — *Condition d'arrêt*
(a) $total \leftarrow total + n$
(b) $n \leftarrow$ nouvelle saisie
} *Corps de la boucle*

Dans une boucle nous allons distinguer :
- $(n \neq 0)$ est la *condition d'arrêt*.
- n est la *variable du test* : sa valeur est testée dans la condition d'arrêt.
- $total$ est la *variable calculée* : elle détient le résultat des calculs propres au problème.

– $n, total$ sont les *variables modifiées* par le corps de la boucle.

Supposons qu'on applique l'algorithme avec saisie de 5, 3, 7, 0. L'algorithme se déroule de la manière suivante :

1.	$n \leftarrow 5$	*Initialisations*
2.	$total \leftarrow 0$	
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5\,;\, n \leftarrow 3}$
3.	Tester $(n = 3 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5 + 3\,;\, n \leftarrow 7}$
3.	Tester $(n = 7 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5 + 3 + 7\,;\, n \leftarrow 0}$
3.	Tester $(n = 0 \neq 0)? \Rightarrow faux \Rightarrow$	*Arrêt*
4.	Afficher $total$	15

A chaque itération, $total$ accumule la somme de sa valeur initiale (0) et des valeurs prises par $n = 5, 3, 7$.

5.1.1 Étapes d'une boucle

Toute boucle s'organise autour d'activités d'initialisation, test, calculs, etc. Nous donnons un schéma d'écriture de boucle très courant, que nous illustrons avec l'algorithme précédent. L'ordre des étapes peut varier d'une boucle à l'autre. Ce qui importe, est de ne pas omettre aucune des étapes.

1. Initialisation des variables de la boucle ($n, total$).
2. Test de la condition (sur n)
3. Dans le corps de la boucle :
   – Nouveau calcul ou traitement des données (sur $total$),
   – Modification des variables du test (n)

Dans notre exemple, l'étape d'initialisation est nécessaire pour réaliser le tout premier test sur n, mais aussi pour le premier calcul de $total$. Une mauvaise initialisation, par exemple, de $total$ à une valeur autre que 0, peut donner lieu à un calcul incorrect. L'importance de l'étape de calcul est évidente.

5.1.2 Boucles qui ne terminent pas

L'étape de modification des variables du test est primordiale pour l'arrêt de la boucle. Supposons que nous l'oublions dans notre algorithme, dont la boucle devient alors :

Tant que $n \neq 0$, faire :

– $total \leftarrow total + n$

L'application de cet algorithme ne s'arrête pas ! En effet, la valeur de n, testée par la condition d'arrêt, ne change jamais, et donc, ne peut jamais devenir égale à zéro...

1.	$n \leftarrow 5(saisie)$	*Initialisations*
2.	$total \leftarrow 0$	
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 0 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5 + 5}$
3.	Tester $(n = 5 \neq 0)? \Rightarrow vrai \Rightarrow$	$\boxed{total \leftarrow 5 + 5 + 5 + 5}\ldots$

5.2 Boucle `while`

5.2.1 rappels

Une boucle `while` en Java a la forme :

Listing 5.1 – (pas de lien)

```
1   while (c) {                                   "tant que c est vrai, faire suiteInstructions"
2       suiteInstructions
3   }
```

où c est une expression booléenne *d'entrée* dans la boucle. Le comportement de cette boucle est :

1. c est évalué avant chaque itération.
2. Si c est vrai, on exécute $suiteInstructions$, puis le contrôle revient au point du test d'entrée (point 1).
3. Si c est faux, le contrôle du programme passe à l'instruction immédiatement après la boucle.

5.2.2 traduction de notre algorithme avec `while`

Listing 5.2 – (lien vers le code brut)

```
1  public class Somme {
2      public static void main (String[] args) {
3          int n, total;
4          // Initialisation de n,total
5          Terminal.ecrireString("Entrez un entier (fin avec 0): ");
6          n = Terminal.lireInt();
7          total = 0;
8          while ( n !=0 ) {
9              total = total + n;                  // Calcul
10             Terminal.ecrireString("Entrez un entier (fin avec 0): ");
11             n = Terminal.lireInt();          // Modification variable du test
12         }
13         Terminal.ecrireStringln("La somme totale est: " + total);
14     }
15 }
```

`total` est *initialisée* à zéro, qui est l'élément neutre de l'addition ; et `n` est initialisée avec première saisie. A chaque tour de boucle, une nouvelle valeur pour `total` est calculée : c'est la somme de la dernière valeur de `total` et du dernier nombre `n` saisi (lors de l'itération précédente). De même, une nouvelle valeur pour `n` est saisie. □

Remarque : Notez que cette boucle peut ne jamais exécuter $suiteInstructions$, si avant la toute première itération, le test c est faux. Dans l'exemple 1, c'est le cas, si lors de la première saisie, n vaut 0.

Étapes d'une boucle `while`

Le programme de l'exemple 1 contient plusieurs étapes différentes : d'initialisation, test, etc, décrites dans la partie précédente. Ici, elles sont organisées de la manière suivante :

Listing 5.3 – (pas de lien)

```
1        Instructions_init                  // Initialisation des variables de la boucle
2        while ( c ) {                      // Test
3            Instruction_1;...          // Calculs
4            Instruction_j;...          // Modification des variables du test
5        }
```

Il s'agit d'une structuration typique des boucles `while`, que nous reprendrons souvent. Les instructions de calcul et de modification des variables du test ne se font pas forcément dans cet ordre : selon le problème, ou selon la condition d'arrêt, on pourra les faire à des moments différents.

Voici les messages affichés par cette exécution :

```
% java Somme
Entrez un entier (fin avec 0): 5
Entrez un entier (fin avec 0): 4
Entrez un entier (fin avec 0): 7
Entrez un entier (fin avec 0): 0
La somme totale est: 16
```

5.3 Boucle `do-while`

La boucle `do-while` est une boucle `while` où les instructions du corps sont exécutées avant de tester la condition de la boucle.

Listing 5.4 – (pas de lien)

```
1    do                                   "faire suiteInstructions
2      {
3          suiteInstructions
4      }
5    while (c);                           tant que c est vrai"
```

où c est une expression booléenne. Le comportement de cette boucle est :

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

L'intérêt d'une boucle `do-while` est de pouvoir exécuter au moins une fois les instructions du corps avant de tester la condition d'arrêt. Cela peut éviter la duplication des instructions de modification des variables du test, qui est parfois nécessaire dans les boucles `while`. Rappelons le code de `Somme.java` :

Listing 5.5 – (lien vers le code brut)

```
1        Terminal.ecrireString("Entrez un entier (fin avec 0): ");
2        n = Terminal.lireInt();  // Saisie d'initialisation pour n
3        total = 0;
4        while ( n !=0 )  {
```

```
5            total = total + n;
6            Terminal.ecrireString("Entrez un entier (fin avec 0): ");
7            n = Terminal.lireInt(); // Nouvelle saisie de n
8        }
```

Nous sommes obligés de saisir une première valeur pour `n` avant le test d'entrée en boucle `(n != 0)`. Or, cette saisie doit être répétée lors de chaque itération. Une écriture plus élégante utilise `do-while` :

Exemple 2 : Réécriture de l'exemple 1 avec `do-while`. Dans ce code, la saisie de `n` n'est faite qu'une fois. Notez également, que l'ordre des instructions dans le corps de la boucle change : la saisie se fait avant les calculs de la même itération, et non pas pour les calculs de la prochaine.

Listing 5.6 – (lien vers le code brut)

```
1        total = 0;
2        do
3          {
4             Terminal.ecrireString("Entrez un entier (fin avec 0): ");
5             n = Terminal.lireInt();    // Saisie de n
6             total = total + n;
7          }
8        while ( n !=0 );
```

5.4 Boucle `for`

En Java, l'entête des boucles `for` incorpore les étapes d'initialisation, test, calcul et de modification des variables du test. Elle a la forme :

Listing 5.7 – (pas de lien)

```
1     for ( initInstrs ; boolExpr ; modifInstrs ) {
2           calculInstrs
3         }
```

- `(` $initInstrs$ `;` $boolExpr$ `;` $modifInstrs$ `)` est *l'entête de la boucle* ;
- $initInstrs$ sont les *initialisations* de la boucle. Il s'agit d'une ou plusieurs instructions **séparées par des virgules**, dont le but est d'initialiser ou déclarer les variables de la boucle ;
- $boolExpr$ est l'expression booléenne *condition* de la boucle ;
- $modifInstrs$ sont les instructions de *mise à jour*. Il s'agit d'une ou plusieurs instructions séparées par des virgules, pour la mise à jour des variables de la boucle, et plus particulièrement, des variables de la condition.
- $\{calculInsts\}$ est le *corps de la boucle*. C'est un bloc d'instructions dont le but est de réaliser les calculs où traitements propres au problème.

Le comportement de cette boucle est équivalent à celui de la boucle `while` :

Listing 5.8 – (pas de lien)

```
1   {
2     initInstrs;
3     while  (boolExpr) {
4              calculInstrs;
5              modifInstrs; }
```

```
6   }
```

1. Les instructions d'initialisation $initInstr$ sont évaluées une seule fois : avant le début des itérations.
2. $boolExpr$ est évalué avant le début de chaque itération.
3. S'il est vrai, on exécute :
   (a) les instructions $calculInstr$ du corps de la boucle,
   (b) puis celles de mise à jour des variables : $modifInstrs$,
   (c) puis, le contrôle revient au test de $boolExpr$ (point 2).
4. sinon, la boucle termine.

Exemple 3 : Calculons la somme des n premier entiers : 1 + 2 + 3 + .. + n, pour un nombre n entier positif saisit au clavier. On se donne deux variables, `i` et `somme` : à chaque tour de boucle `i` est incrémentée de 1, et `somme` accumule la somme des valeurs de `i` après un certain nombre d'itérations. On veut répéter ces actions tant que `i` est inférieur ou égal à n.

Entrées : n entier à saisir
Sortie : $somme$ entier
Algorithme :

1. Initialiser n par une saisie,
2. Initialiser $somme$ avec 0, et i avec 1, ($initInstrs$)
3. Tant que $i \leq n$ faire : ($boolExpr$)
   (a) $somme \leftarrow somme + i$ ($calculInstrs$)
   (b) $i \leftarrow i + 1$ ($modifInstrs$)
4. Afficher la valeur de $somme$.

Une traduction avec une boucle `for` est :

Listing 5.9 – (lien vers le code brut)

```
1  public class Somme_n {
2      public static void main (String[] args) {
3          int n, i, somme;
4          Terminal.ecrireString("Un entier positif? ");
5          n = Terminal.lireInt();
6          for (somme = 0, i = 1; i <= n; i = i+1) {
7                somme = somme + i;
8          }
9          Terminal.ecrireString("La somme 1+2+..+ "+ n);
10         Terminal.ecrireStringln(" = " + somme);
11     }
12 }
```

Voici une boucle `while` équivalente. Elle est une traduction presque textuelle de notre algorithme de départ :

Listing 5.10 – (lien vers le code brut)

```
1          somme = 0; i = 1;              // Initialisations
2          while (i <= n)  {
```

```
3                 somme = somme + i;        // Calculs
4                 i = i+1;                  // Increment
5         }
```

Une trace de l'exemple, en supposant que l'on saisit `n = 4` :

	init	*itération* 1	*itération* 2	*itération* 3	*itération* 4	*itération* 5
`(i <=n)`		`(1 <= 4)`	`(2 <= 4)`	`(3 <= 4)`	`(4 <= 4)`	`(5 <= 4)`
`somme`	0	0+1	0+1+2	0+1+2+3	0+1+2+3+4	*arrêt*
`i`	1	2	3	4	5	*arrêt*

5.4.1 Usage des boucles `for`

En Java, les boucles `for` sont plus expressives que dans la plupart de langages : avec elles, on peut écrire les mêmes boucles qu'avec un `while`. Mais, par convention, les boucles `for` sont utilisées pour faire varier une ou plusieurs variables d'*itération*, dans un intervalle de valeurs reliées entre elles, jusqu'à la vérification d'une condition. Dans l'exemple 3, la variable d'itération est `i`, qui varie de $i = 1$ jusqu'à $i \leq n$. Ces variations se font souvent par incrément ou décrément, à l'aide d'expressions `i++` ou `i--`. La boucle de l'exemple 3 devient alors :

Listing 5.11 – (pas de lien)

```
1         for (somme = 0, i = 1; i <= n; i++) {
2                 somme = somme + i;
3         }
```

Exemple 4 : Calculons la puissance a^b pour deux nombres entiers a, b saisis au clavier, où $b \geq 0$. Nous faisons varier (dégressive-ment) une variable d'itération i, de $i = b$ jusqu'à $i = 1$.

Entrées : a, b entiers à saisir, $b \geq 0$.
Sortie : p entier.
Variable d'itération : $i \in [b \ldots 1]$ (intervalle d'itération)
Algorithme :

1. Initialiser a, b par une saisie,
2. Initialiser p avec 1, et i avec b, $(initInstrs)$
3. Tant que $i \geq 1$ faire : $(boolExpr)$
   - $p \leftarrow p * a$ $(calculInstrs)$
   - $i \leftarrow i - 1$ $(modifInstrs)$
4. Afficher la valeur de p.

Écrite avec un `for`, cette boucle devient :

Listing 5.12 – (pas de lien)

```
1         for (p = 1, i = b; i >= 1; i--) {
2                 p = p*a;
3         }
```

5.4.2 Variables locales à une boucle `for`

Dans une boucle `for`, les variables d'itération n'ont souvent d'intérêt que le temps d'exécuter la boucle. Une fois finie, c'est le résultat calculé par celle-ci qu'il est important de récupérer, afficher, etc. Dans ce cas, une bonne pratique de programmation est d'introduire ces variables de manière locale à la boucle, par une déclaration dans sa partie initialisation ($initInstrs$).

Exemple 5 : Calcul de $1 + \ldots + n$ avec la variable d'itération `i` déclarée localement :

Listing 5.13 – (pas de lien)

```
1        somme = 0;
2        for (int i = 1; i <= n; i++) {
3            somme = somme + i;
4        }
```

La portée de `i` est l'entête et le corps de la boucle. En sortie de boucle, `i` n'est plus visible. Cette zone de visibilité est bien illustrée par la traduction vers une boucle `while` (voir *comportement* d'un `for`) :

Listing 5.14 – (pas de lien)

```
1  {
2    initInstrs;
3    while (boolExpr) {
4        calculInstrs;
5        modifInstrs; }
6  }
```

Toutes les instructions de l'entête et du corps sont entourées d'un bloc supplémentaire. C'est lui qui assure que les variables déclarées par $initInstrs$ ne seront visibles que par les instructions de la boucle.

5.4.3 Entête d'une boucle `for`

Quelques remarques sur les instructions de l'entête d'un `for` :

- Les instructions de l'entête sont toutes optionnelles (voir exemple 6.1).
- Comme dans une boucle `while`, la condition ($boolExpr$) d'une boucle `for` est évaluée avant chaque itération. Ainsi, les instructions du corps ne sont jamais exécutées si la condition est fausse dès le début.
- En revanche, les instructions d'initialisation ($initInstrs$) sont toujours exécutées, même s'il n'y a aucune itération.

Exemple 6 :

1. Pas d'instructions dans l'entête (boucle infinie). Cette boucle ne s'arrête que si `Instructions` exécute une instruction de *rupture de contrôle* (voir plus loin), ou *lève une exception* (voir chapitre sur les fonctions).

Listing 5.15 – (pas de lien)

```
1        for (;;) {
2            Instructions
3        }
```

2. `for` qui ne boucle pas. Si `n = 0`, la condition de la boucle est fausse, et il n'y a aucune itération.

Listing 5.16 – (pas de lien)

```
1           for (somme = 0, i = 1; i <= n; i++) {
2                 somme = somme + i;
3           }
```

L'exécution des initialisations fait que la valeur affichée pour `somme` est correcte (0).

5.5 Instructions de rupture de contrôle : `break`

Les instructions de *rupture de contrôle* permettent la sortie d'un bloc ou d'une boucle avant d'avoir complété leur exécution. Parmi ces instructions, nous étudions `break` dans ce chapitre, et `return` dans le chapitre dédié aux fonctions. `break` est utilisée pour sortir du corps d'une boucle[1] : son utilisation en dehors des boucles (ou d'un `switch`) provoque une erreur à la compilation.

Exemple 7 : Sortie d'une boucle infinie. L'exécution de ce programme affiche les messages A, B, et F. Le message D n'est pas affiché, car il se trouve dans le corps de la boucle, après le `if` qui exécute la sortie de boucle. Si nous enlevons les commentaires pour l'affichage de C, le compilateur signale que cette instruction est inaccesible. En effet, étant juste après un `break`, elle ne sera jamais exécutée.

```
public class TestBreak {
    public static void main (String[] args) {
        int n = 1;
        while (true) {
              Terminal.ecrireStringln("A");
              if (n >= 1) {
                           Terminal.ecrireStringln("B");
                           break;
                           // Terminal.ecrireStringln("C");
      }
              Terminal.ecrireStringln("D");
        }
        Terminal.ecrireStringln("F");
}}
```

```
Java/Essais> java TestBreak
A
B
F
```

□

Exemple 8 : Le jeu suivant pose des questions à l'utilisateur. Après 2 bonnes réponses, la boucle s'arrête et l'utilisateur gagne. Si en fin de boucle, il y a moins de 2 bonnes réponses, un nouveau tour de jeu est entamé, mais il tient compte du nombre des bonnes réponses des tours précédents.

[1] Mais aussi pour sortir d'un `switch`, que nous n'étudions pas dans ces notes.

Listing 5.17 – (lien vers le code brut)

```
1  public class JeuBete {
2      public static void main (String[] args) {
3          Terminal.ecrireStringln("␣␣␣␣␣␣␣*****␣Jouez␣au␣jeu␣JAVA␣******␣");
4          Terminal.ecrireStringln("Repondez␣par␣true␣ou␣false,␣");
5          Terminal.ecrireStringln("et␣gagnez␣au␣bout␣de␣2␣bonnes␣reponses␣\n");
6          int rep =0;
7          for (boolean c;;) {
8              Terminal.ecrireString("Toute␣instruction␣finit␣par␣un␣;␣?␣");
9              c = Terminal.lireBoolean();
10             if (!c) { rep = rep +1; }
11             if (rep == 2) {break;}
12
13             Terminal.ecrireString("L'expression␣(x=2␣==␣x=3)␣est␣mal␣typee␣?");
14             c = Terminal.lireBoolean();
15             if (!c) { rep = rep +1;}
16             if (rep == 2) {break;}
17
18             Terminal.ecrireString("Le␣resultat␣de␣6/4␣est␣1?␣");
19             c = Terminal.lireBoolean();
20             if (c) { rep = rep +1;}
21             if (rep == 2) {break;}
22
23             Terminal.ecrireString("Le␣resultat␣de␣6%4␣est␣2␣?␣");
24             c = Terminal.lireBoolean();
25             if (c) { rep = rep +1;}
26             if (rep == 2) {break;}
27             else {
28                 Terminal.ecrireString("\n␣****␣" + rep + "␣bonnes␣reponses!");
29                 Terminal.ecrireStringln("␣Encore␣un␣tour...");
30             }
31         }
32         Terminal.ecrireStringln("Bravo!␣Vous␣avez␣gagne!!");
33     }
34 }
```

```
Java/Essais> java JeuBete
       ***** Jouez au jeu JAVA ******
Repondez par true ou false,
et gagnez au bout de 2 bonnes reponses

Toute instruction finit par un ; ? false
L'expression (x=2 == x=3) est mal typee ? true
Le resultat de 6/4 est 1? false
Le resultat de 6%4 est 2 ? false

 **** 1 bonnes reponse!! Encore un tour...
Toute instruction finit par un ; ? true
L'expression (x=2 == x=3) est mal typee ? false
Bravo! Vous avez gagne!!
```

Remarques :

- Les instructions `break` sont à utiliser avec modération : elles peuvent rendre plus difficile la compréhension des programmes, et, la plupart du temps, on peut écrire les boucles aussi simplement sans elles.
- Lorsque `break` est exécuté par une boucle imbriquée dans d'autres boucles (voir exemple 11), il permet la sortie du corps de la boucle où il apparaît, mais pas des boucles plus externes.

5.6 Exemples

5.6.1 Boucles imbriquées

Exemple 9 : Le programme suivant affiche la table de multiplication d'un nombre entier n saisit au clavier :

Listing 5.18 – (lien vers le code brut)

```
1  public class UneTableMult {
2      public static void main (String[] args) {
3          int n;
4          Terminal.ecrireString("Un entier entre 2 et 9? ");
5          n = Terminal.lireInt();
6          Terminal.sautDeLigne();
7          Terminal.ecrireStringln("Table de " + n);
8          Terminal.ecrireStringln("**********");
9          for (int i=1; i <= 10; i++)  {
10             Terminal.ecrireStringln(n + " x " + i + " = " + (n*i));
11         }
12         Terminal.sautDeLigne();
13     }
14 }
```

```
Java/Essais> java UneTableMult

Un entier entre 2 et 9? 3

Table de 3
**********
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

Exemple 10 : Écrire un programme qui affiche toutes les tables de multiplication de 2 à 9. On écrira deux *boucles imbriquées* :

- La boucle la plus externe fait varier entre 2 et 9 le numéro n de la table à afficher.
- Dans cette première boucle, on veut, pour chaque n différent, afficher les lignes : `n x 1`, `n x 2`, ...
- Cela revient à réaliser pour chaque pas n de la boucle externe, toute l'exécution d'une boucle plus interne, chargée d'afficher `n x i` pour i qui varie entre 1 et 10 (comme dans `UneTableMult`).
- `n` et `i` n'étant pas nécessaires en dehors des boucles, nous pouvons les déclarer localement.

Listing 5.19 – (lien vers le code brut)

```
1  public class TablesMult {
2      public static void main (String[] args) {
3          Terminal.sautDeLigne();
4          Terminal.ecrireStringln("Tables de multiplication de 2 a 9");
5          Terminal.ecrireStringln("********************************");
6          Terminal.sautDeLigne();
7          for (int n=2; n <= 9; n++)  {
8              Terminal.ecrireStringln("Table de " + n);
9              Terminal.ecrireStringln("**********");
10             for (int i=1; i <= 10; i++)  {
11                 Terminal.ecrireStringln(n + " x " + i + " = " + (n*i));
12             }
13             Terminal.sautDeLigne();
14             Terminal.ecrireStringln("----------------");
15             Terminal.sautDeLigne();
16         }
17     }
18 }
```

et les premiers affichages à l'exécution :

```
Tables de multiplication de 2 a 9
*********************************

Table de 2
**********
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20

----------------

Table de 3
**********
3 x 1 = 3
```

```
3 x 2 = 6
3 x 3 = 9
........
```

Exemple 11 : Sortie d'une boucle imbriquée avec `break`. Dans cet exemple, `break` est exécuté par une boucle imbriquée, ce qui provoque l'arrêt et la sortie de la boucle qui entoure immédiatement cet instruction. Ainsi, seule la boucle sur `i` est arrêtée. Après sortie de celle-ci, le contrôle reprend dans la prochaine instruction de la boucle externe (affichage de `Terminal.ecrireStringln(" Fin boucle 1")`). Cette boucle externe exécute plusieurs fois la boucle interne, avec à chaque fois une sortie par `break`.

Listing 5.20 – (lien vers le code brut)

```
1  public class TestBreak2 {
2      public static void main (String[] args) {
3          for (int m=1; m <= 3; m++)  {
4              Terminal.ecrireStringln("Boucle 1: m = " + m );
5              for (int i=1; i <= 10; i++)  {
6                  Terminal.ecrireStringln("  Boucle 2: i = " + i);
7                  if (true) break;
8                  Terminal.ecrireStringln("  Fin boucle 2");
9              }
10             Terminal.ecrireStringln("Fin boucle 1");
11         }
12     }
13 }
```

```
Java/Essais> java TestBreak2
Boucle 1: m = 1
  Boucle 2: i = 1
Fin boucle 1
Boucle 1: m = 2
  Boucle 2: i = 1
Fin boucle 1
Boucle 1: m = 3
  Boucle 2: i = 1
Fin boucle 1
```

5.6.2 Validation des entrées

Parfois, certains calculs ne sont définis que pour un sous-ensemble de toutes les entrées possibles au programme. Par exemple, la somme des n premiers entiers positifs $1 + 2 + \ldots + n$, n'est défini que si $n \geq 1$. Or, un programme qui déclare n en tant que variable entière, peut saisir une valeur entière négative. Le programme de l'exemple 2, devant une entrée négative, affiche un message qui n'a pas beaucoup de sens :

Listing 5.21 – (lien vers le code brut)

```
1  public class Somme_n {
2      public static void main (String[] args) {
3          int n, i, somme;
```

```
4           Terminal.ecrireString("Un entier positif? ");
5           n = Terminal.lireInt();
6           for (somme = 0, i = 1; i <= n; i = i+1) {
7                 somme = somme + i;
8           }
9           Terminal.ecrireString("La somme 1+2+..+ "+ n);
10          Terminal.ecrireStringln(" = " + somme);
11      }
12  }
```

```
Java/Essais> java Somme_n
Un entier positif? -4
La somme 1+2+..+ -4 = 0
```

Ici, la boucle `for` initialise `somme` à 0 et `i` à 1, puis teste `i <= n` avec `n` est négatif. Le test étant faux, le corps de la boucle n'est jamais exécuté et la variable `somme` reste avec sa valeur initiale 0. Modifions ce programme pour conditionner l'exécution de la boucle à la validité des données saisies, et pour signaler une erreur dans le cas contraire.

Exemple 12 : Calcul de $1 + 2 + \ldots + n$ avec message d'erreur si $n \leq 0$.

Listing 5.22 – (lien vers le code brut)

```
1   public class Somme_nValBasic {
2       public static void main (String[] args) {
3           int n, somme;
4           Terminal.ecrireString("Un entier positif? ");
5           n = Terminal.lireInt();
6           if (n > 0) {
7               somme = 0;
8               for (int i = 1; i <= n; i++) {
9                     somme = somme + i;
10              }
11              Terminal.ecrireString("La somme 1+2+..+ "+ n);
12              Terminal.ecrireStringln(" = " + somme);
13          } else {
14              Terminal.ecrireStringln("Nombre negatif: calcul impossible");
15          }
16      }
17  }
```

```
Java/Essais> java Somme_nValBasic
Un entier positif? -4
Nombre negatif: calcul impossible

/Java/Essais> java Somme_nValBasic
Un entier positif? 4
La somme 1+2+..+ 4 = 10
```

Le problème ici est qu'il faut relancer l'exécution du programme en cas d'erreur de saisie. Une meilleure solution, est de demander une nouvelle saisie via une *boucle de saisie et validation*. Elle

fera la saisie des entrées puis testera leur validité, et cela tant que celles-ci ne sont pas correctes.

Exemple 13 : Calcul de $1 + 2 + \ldots + n$ avec boucle de saisie et validation de n.

Listing 5.23 – (lien vers le code brut)

```
1  public class Somme_nVal {
2      public static void main (String[] args) {
3          int n, somme;
4          do {
5                  Terminal.ecrireString("Un entier positif? ");
6                  n = Terminal.lireInt();
7                  if (n > 0) { break;}
8                  Terminal.ecrireStringln("*** Nombre negatif. Recommencez! ");
9          } while (true);
10         somme = 0;
11         for (int i = 1; i <= n; i++) {
12               somme = somme + i;
13         }
14         Terminal.ecrireString("La somme 1+2+..+ "+ n);
15         Terminal.ecrireStringln(" = " + somme);
16     }
17 }
```

```
Java/Essais> java Somme_nVal
Un entier positif? -4
*** Nombre negatif. Recommencez!
Un entier positif? -3
*** Nombre negatif. Recommencez!
Un entier positif? 5
La somme 1+2+..+ 5 = 15
```