Chapitre 3

Conditionnelle et boucles

Dans ce chapitre, nous allons étudier plusieurs instructions qui permettent de faire varier le comportement des programmes en fonction de certaines conditions.

3.1 Conditionnelle

Nous voudrions maintenant écrire un programme qui, étant donné un prix Hors Taxe (HT) donné par l'utilisateur, calcule et affiche le prix correspondant TTC. Il y a 2 taux possible de TVA : la TVA normale à 19.6% et le taux réduit à 5.5%. On va demander aussi à l'utilisateur la catégorie du taux qu'il faut appliquer.

données

- entrée : un prix HT de type `double` (`pHT`), un taux de type `int` (`t`) (0 pour normal et 1 pour réduit)
- sortie : un prix TTC de type `double` (`pTTC`)

algorithme

1. afficher un message demandant une somme HT à l'utilisateur.
2. recueillir la réponse dans `pHT`
3. afficher un message demandant le taux (0 ou 1).
4. recueillir la réponse dans `t`
5. 2 cas :
   (a) Cas 1 : le taux demandé est normal `pTTC= pht+ (pht*0.196)`
   (b) Cas 2 : le taux demandé est réduit `pTTC= pht+ (pht*0.05)`
6. afficher `pTTC`

Avec ce que nous connaissons déjà en Java, nous ne pouvons coder cet algorithme. La ligne 5 pose problème : elle dit qu'il faut dans un certain cas exécuter une tâche, et dans l'autre exécuter une autre tâche. Nous ne pouvons pas exprimer cela en Java pour l'instant, car nous ne savons qu'exécuter, les unes après les autres de façon inconditionnelle la suite d'instructions qui constitue notre programme.

Pour faire cela, il y a une instruction particulière en Java, comme dans tout autre langage de programmation, l'instruction conditionnelle, qui à la forme suivante :

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

et s'exécute de la façon suivante : si la condition est vraie c'est la suite d'instructions `instructions1` qui est exécutée ; si la condition est fausse, c'est la suite d'instructions `instructions2` qui est exécutée.

La condition doit être une expression booléenne c'est à dire une expression dont la valeur est soit `true` soit `false`.

Voilà le programme Java utilisant une conditionnelle qui calcule le prix TTC :

```
public class PrixTTC {
    public static void main (String[] args) {
        double pHT,pTTC;
        int t;
        Terminal.ecrireString("Entrer␣le␣prix␣HT:␣");
        pHT = Terminal.lireDouble();
        Terminal.ecrireString("Entrer␣taux␣(normal->0␣reduit␣->1)␣");
        t = Terminal.lireInt();
        if (t==0){
            pTTC=pHT + (pHT*0.196);
        }
        else {
            pTTC=pHT + (pHT*0.05);
        }
        Terminal.ecrireStringln("La␣somme␣TTC:␣"+ pTTC );
    }
}
```

Ce programme est constitué d'une suite de 8 instructions. Il s'exécutera en exécutant séquentiellement chacune de ces instructions :

1. déclaration de `pHT` et pTTC
2. déclaration de `t`
3. affichage du message `"Entrer le prix HT:"`
4. la variable `pHT` reçoit la valeur entrée au clavier.
5. affichage du message `"Entrer taux (normal->0 reduit ->1)"`
6. la variable `t` reçoit la valeur entrée au clavier (0 ou 1)
7. Java reconnaît le mot clé `if` et fait donc les choses suivantes :
   (a) il calcule l'expression qui est entre les parenthèses `t==0`. Le résultat de `t==0` dépend de ce qu'a entré l'utilisateur. S'il a entré `0` le résultat sera `true`, sinon il sera `false`.
   (b) Si le résultat est `true`, les instructions entre les accolades sont exécutées. Ici, il n'y en a qu'une : `pTTC=pHT + (PHT*0.196);` qui a pour effet de donner a `pTTC` le prix TTC avec taux normal. Si le résultat est `false`, il exécute les instructions dans les accolades figurant après le `else` :`pTTC=pHT + (PHT*0.05);`
8. la valeur de `pTTC` est affichée. cette dernière instruction **est toujours exécutée** : elle n'est pas à l'intérieur des accolades du `if` ou du `else`. La conditionnelle a servi a mettre une valeur différente dans la variable `pTTC`, mais il faut dans les deux cas afficher cette valeur.

3.1.1 if sans else

Lorsqu'on veut dire : si `condition` est vraie alors faire ceci, sinon ne rien faire, on peut omettre le `else` ;

3.1.2 tests à la suite

Lorsque le problème à résoudre nécessite de distinguer plus de 2 cas, on peut utiliser la forme suivante :

```
if (condition1){
  s1
}else if (condition2){
  s2
}
...
else if (conditionp){
  sp
}else {
  sf
}
```

On peut mettre autant de `else if` que l'on veut, mais 0 ou 1 `else` (et toujours à la fin). `else if` signifie sinon si. Il en découle que les conditions sont évaluées dans l'ordre. La première qui est vraie donne lieu à la séquence d'instructions qui est y associée. Si aucune condition n'est vraie, c'est le `else` qui est exécuté.

3.1.3 Trace d'une exécution

Voici la trace d'exécution du programme de prix TTC :

Après l'instruction	Mémoire			Entrées	Sorties
	`pHt`	`t`	`pTTC`		
3	?	NEP	?		
4	?	?	?		
5	?	?			`Entrer le prix HT:`
6	10.0	?		10.0	
7	10.0	?			`Entrer le taux(0,1):`
8	10.0	1		1	
9	10.0	1			
13	10.0	1	10.5		
15	10.0	1	10.5		`La somme TTC: 10.5`

3.2 La boucle for

Ecrivons un programme qui affiche à l'écran un rectangle formé de 5 lignes de 4 étoiles. Ceci est facile : il suffit de faire 5 appels consécutifs à la méthode `Terminal.ecrireStringln`

```
public class Rectangle {
    public static void main (String[] args) {
            Terminal.ecrireStringln("****");
            Terminal.ecrireStringln("****");
            Terminal.ecrireStringln("****");
            Terminal.ecrireStringln("****");
            Terminal.ecrireStringln("****");
    }
}
```

Ceci est possible, mais ce n'est pas très élégant ! Nous avons des instructions qui nous permettent de répéter des tâches. Elles s'appellent les *boucles* ou instructions d'*itérations*.

La boucle *for* permet de répéter une tâche un nombre de fois connus à l'avance. Ceci est suffisant pour l'affichage de notre rectangle :

```
public class Rectangle {
    public static void main (String[] args) {
          for (int i=0;i<5;i=i+1){
             Terminal.ecrireStringln("****");
          }
    }
}
```

Répète `i` fois, pour `i` allant de 0 à 4 les instructions qui sont dans les accolades, c'est à dire dans notre cas : afficher une ligne de 4 `*`.
Cela revient bien à dire : répète 5 fois "afficher une ligne de 4 étoiles".

Détaillons cela : La boucle `for` fonctionne avec un compteur du nombre de répétition qui est géré dans les 3 expressions entre les parenthèses.

- La première : `int i=0` donne un nom a ce compteur (`i`) et lui donne une valeur initiale `0`. Ce compteur ne sera connu qu'à l'intérieur de la boucle for. (il a été declaré dans la boucle for `int i`)
- La troisième : `i=i+1` dit que les valeurs successives de `i` seront 0 puis 0+1 puis 1+1, puis 2+1 etc.
- La seconde (`i<5`) dit quand s'arrête l'énumération des valeurs de `i` : la première fois que `i<5` est faux.

Grâce à ces 3 informations nous savons que `i` va prendre les valeurs successives 0, 1, 2, 3, 4. Pour chacune de ces valeurs successives, on répètera les instructions dans les accolades.

Plus généralement, une boucle `for` comporte trois choses entre parenthèses, suivies du corps de la boucle entre accolades :

Listing 3.1 – (pas de lien)

```
for(initialisation; test; incrémentation){
    corps de la boucle;
}
```

- la partie initialisation est exécutée une fois, avant de commencer la boucle. Il s'agit généralement de l'initialisation d'une variable et éventuelement sa d'éclaration.
- la partie test est évaluée avant de rentrer dans la boucle, à chaque répétition. Si le test a pour résultat la valeur `true` (vrai), alors le corps de la boucle est exécuté. Si le résultat est `false` (faux), la boucle est terminée. Le programme passe à l'expression suivante, après la boucle.

- la partie incrémentation est exécutée après le corps de la boucle, avant de refaire le test pour un tour de plus. Il s'agit généralement de l'incrémentation d'une variable.
- le corps de la boucle comporte n'importe quelle suite d'instructions.

3.2.1 Trace d'exécution d'une boucle for

Dans cette trace d'exécution, nous allons distinguer les trois éléments d'une boucle `for` bien qu'ils soient sur la même ligne de code.

ligne	i	test $i < 5$	écran
3.initialisation	0		
3.test	0	true	
4	0		****
3.incrémentation	1		
3.test	1	true	
4	1		****
3.incrémentation	2		
3.test	2	true	
4	2		****
3.incrémentation	3		
3.test	3	true	
4	3		****
3.incrémentation	4		
3.test	4	true	
4	4		****
3.incrémentation	5		
3.test	5	false	

3.2.2 Jouons un peu

Pour être sur d'avoir compris, examinons les programmes suivants :

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=0;i<5;i=i+2){
            Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 0, 2, 4. Il y aura donc 3 répétitions. Ce programme affiche 3 lignes de 4 étoiles.

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=1;i<5;i=i+2){
            Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 1, 3. Il y aura donc 2 répétitions. Ce programme affiche 2 lignes de 4 étoiles.

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=1;i<=5;i=i+2){
            Terminal.ecrireStringln("****");
        }
    }
}
```

`i` va prendre les valeurs successives 1, 3, 5. Il y aura donc 3 répétitions. Ce programme affiche 3 lignes de 4 étoiles.

```
public class Rectangle {
    public static void main (String[] args) {
        for (int i=1;i==5;i=i+2){
            Terminal.ecrireStringln("****");
        }
    }
}
```

`i` ne prendra aucune valeur car sa première valeur (`1`) n'est pas égale à `5`. Les instructions entre accolades ne sont jamais exécutées. Ce programme n'affiche rien.

3.2.3 Les boucles sont nécessaires

Dans notre premier exemple, nous avons utilisé une boucle pour abréger notre code. Pour certains problèmes, l'utilisation des boucles est absolument nécessaire. Essayons par exemple d'écrire un programme qui affiche un rectangle d'étoiles dont la longueur est donnée par l'utilisateur (la largeur restera 4).

Ceci ne peut de faire sans boucle car le nombre de fois où il faut appeler l'instruction d'affichage dépend de la valeur donnée par l'utilisateur. En revanche, cela s'écrit très bien avec une boucle `for` :

```
public class Rectangle2 {
    public static void main (String[] args) {
        int l;
        Terminal.ecrireString("combien␣de␣lignes␣d'etoiles␣?:");
        l=Terminal.lireInt();
        for (int i=0;i<l;i=i+1){
            Terminal.ecrireStringln("****");
        }
    }
}
```

La variable `l` est déclarée dans la première instruction. Elle a une valeur à l'issue de la troisième instruction. On peut donc tout à fait consulter sa valeur dans la quatrième instruction (le `for`).
Si l'utilisateur entre 5 au clavier, il y aura 5 étapes et notre programme affichera 5 lignes de 4 étoiles.
Si l'utilisateur entre 8 au clavier, il y aura 8 étapes et notre programme affichera 8 lignes de 4 étoiles.

3.2.4 le compteur d'étapes peut intervenir dans la boucle

Le compteur d'étapes est connu dans la boucle. On peut tout à fait consulter son contenu dans la boucle.

```
public class Rectangle3 {
    public static void main (String[] args) {
        int l;
        Terminal.ecrireString("combien␣de␣lignes␣d'etoiles␣?:");
        l=Terminal.lireInt();
        for (int i=1;i<=l;i=i+1){
            Terminal.ecrireInt(i);
            Terminal.ecrireStringln("****");
        }
    }
}
```

Ce programme affiche les lignes d'étoiles précédées du numéro de ligne.

C'est pour cela que l'on a parfois besoin que la valeur du compteur d'étapes ne soit pas son numéro dans l'ordre des étapes. Voici un programme qui affiche les n premiers entiers pairs avec n demandé à l'utilisateur.

```
public class PremPairs {
    public static void main (String[] args) {
        int n;
        Terminal.ecrireString("combien␣d'entiers␣pairs␣?:");
        n=Terminal.lireInt();
        for (int i=0;i<n*2;i=i+2){
            Terminal.ecrireInt(i);
            Terminal.ecrireString("␣,␣");
        }
    }
}
```

Et voici un programme qui affiche les 10 premiers entiers en partant de 10 :

```
public class PremEntiers {
    public static void main (String[] args) {
        for (int i=10;i>0;i=i-1){
            Terminal.ecrireString(i + ",");
        }
    }
}
```

Note : la boucle `for` en Java est une instruction complexe avec la possibilité d'écrire des initialisations et des instructions de fin de boucle complexes. Dans ce cours, nous n'utilisons pas cette possibilité et nous n'encourageons pas l'écriture de boucle `for` exotiques qui sont peu lisibles.

3.3 La boucle while

Certaines fois, la boucle `for` ne suffit pas. Ceci arrive lorsqu'au moment où on écrit la boucle, on ne peut pas déterminer le nombre d'étapes.

Reprenons notre exemple de calcul de prix TTC. Dans cet exemple, nous demandions à l'utilisateur d'entrer 0 pour que l'on calcule avec le taux normal et 1 pour le taux réduit.
Que se passe-t-il si l'utilisateur entre 4 par exemple ? 4 est diffèrent de 0 donc le `else` sera exécuté. Autrement dit : toute autre réponse que 0 est interprétée comme 1.
Ceci n'est pas très satisfaisant. Nous voulons maintenant améliorer notre programme pour qu'il redemande à l'utilisateur une réponse, tant que celle-ci n'est pas 0 ou 1.
Nous sentons bien qu'il faut une boucle, mais la boucle `for` est inadaptée : on ne peut dire a priori combien il y aura d'étapes, cela dépend de la vivacité de l'utilisateur ! Pour cela, nous avons la boucle `while` qui a la forme suivante :

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

Cette boucle signifie : tant que la condition est vraie, exécuter les instructions entre les accolades (le *corps de la boucle*)
Grâce à cette boucle, on peut répéter une tâche tant qu'un **évènement dans le corps de la boucle** ne s'est pas produit.
C'est exactement ce qu'il nous faut : nous devons répéter la demande du taux, tant que l'utilisateur n'a pas fourni une réponse correcte.

3.3.1 écriture de la boucle

La condition de notre boucle devra être une expression booléenne Java qui exprime le fait que la réponse de l'utilisateur est correcte. Pour faire cela, il suffit d'ajouter une variable à notre programme, que nous appellerons `testReponse`. Notre programme doit s'arranger pour qu'elle ait la valeur `true` dès que la dernière saisie de l'utilisateur est correcte, c'est à dire si `t` vaut 0 ou 1, et fausse sinon.

Notre boucle pourra s'écrire :

```
while (testReponse==false){
  Terminal.ecrireStringln("Entrer taux (normal->0 reduit ->1) ");
  t = Terminal.lireInt();
  if (t==0 || t==1){
    testReponse=true;
  }
  else {
    testReponse=false;
  }
}
```

Il faudra, bien entendu, avoir déclaré `testReponse` avant la boucle.

3.3.2 comportement de la boucle

La condition de la boucle est testée **avant** chaque exécution du corps de la boucle. On commence donc par tester la condition ; si elle est vraie le corps est exécuté une fois, puis on teste a nouveau la condition et ainsi de suite. L'exécution de la boucle se termine la première fois que la condition est fausse.

3.3.3 initialisation de la boucle

Puisque `testReponse==false` est la première chose exécutée lorsque la boucle est exécutée, il faut donc que `testReponse` ait une valeur **avant** l'entrée dans la boucle. C'est ce qu'on appelle *l'initialisation de la boucle*. ici, puisque l'on veut entrer au moins une fois dans la boucle, il faut initialiser `testReponse` avec `false`.

3.3.4 état de sortie de la boucle

Puisqu'on sort d'une boucle `while` la première fois que la condition est fausse, nous sommes sûrs que dans notre exemple, en sortie de boucle, nous avons dans `t` une réponse correcte : 0 ou 1. Les instructions qui suivent la boucle sont donc le calcul du prix TTC selon des 2 taux possibles.

Voici le code Java :

```
public class PrixTTC2 {
    public static void main (String[] args) {
        double pHT,pTTC;
        int t=0;
        boolean testReponse=false;
        Terminal.ecrireString("Entrer␣le␣prix␣HT:␣");
        pHT = Terminal.lireDouble();
        while (testReponse==false){
            Terminal.ecrireString("Entrer␣taux␣(normal->0␣reduit␣->1)␣");
            t = Terminal.lireInt();
            if (t==0 || t==1){
                testReponse=true;
            }
            else {
                testReponse=false;
            }
        }
        if (t==0){
            pTTC=pHT + (pHT*0.196);
        }
        else {
            pTTC=pHT + (pHT*0.05);
        }
        Terminal.ecrireStringln("La␣somme␣TTC:␣"+ pTTC );
    }
}
```

3.3.5 Terminaison des boucles while

On peut écrire avec les boucles while des programmes qui ne s'arrêtent jamais. C'est presque le cas de notre exemple : si l'utilisateur n'entre jamais une bonne réponse, la boucle s'exécutera à l'infini. Dans ce cours, les seuls programmes qui ne terminent pas toujours et que vous aurez le droit d'écrire seront de cette catégorie : ceux qui contrôlent les saisies utilisateurs.

Pour qu'une boucle `while` termine, il faut s'assurer que le corps de la boucle contient des instructions qui mettent à jour la condition de boucle. Autrement dit, le corps de la boucle est toujours constitué de deux morceaux :

– le morceau décrivant la tâche à effectuer à chaque étape
– le morceau décrivant la mise à jour de la condition de sortie

Pour qu'une boucle termine **toujours**, il faut que **chaque** mise à jour de la condition de sortie, nous rapproche du moment où la condition sera fausse.

C'est bien le cas dans l'exemple suivant :

```
public class b1 {
    public static void main (String[] args) {

        int i=1;

        while (i!=10){
          Terminal.ecrireString(i + ",");
          i=i+1;
        }
    }
}
```

Au début de la première exécution du corps de boucle `i` vaut 1, Au début de la deuxième exécution du corps de boucle `i` vaut 2, ... A chaque fois, on progresse vers le moment où `i` vaut 10. Ce cas est atteint au bout de 10 étapes.

Attention, il ne suffit de se rapprocher du moment où la condition est fausse, il faut l'atteindre un jour, ne pas la rater.

Le programme suivant, par exemple, ne termine jamais :

```
public class b3 {
    public static void main (String[] args) {

        int i=1;

        while (i!=10){
          Terminal.ecrireString(i + ",");
          i=i+2;
        }
    }
}
```

Car `i` progresse de 1 à 3, puis à 5, à 7, à 9, à 11 et 10 n'est jamais atteint.

3.3.6 la boucle while suffit

La boucle `while` est plus générale que la boucle `for`. On peut traduire tous les programmes avec des boucles `for` en des programmes avec des boucles `while`. On peut donc se passer de la boucle `for` ; il est cependant judicieux d'employer la boucle `for` à chaque fois que c'est possible, parce qu'elle est plus facile à écrire, à comprendre et à maîtriser.

3.3.7 Trace de programme avec boucle while

Le programme suivant calcule la somme d'une suite de nombres entiers saisis au clavier. Le calcul s'arrête lors de la saisie du nombre zéro.

```
public class Somme {
    public static void main (String[] args) {
        int n, total;
        // Initialisation de n,total
        Terminal.ecrireString("Entrez␣un␣entier␣(fin␣avec␣0):␣");
        n = Terminal.lireInt();
        total = 0;
        while ( n !=0 ) {
            total = total + n; // Calcul
            Terminal.ecrireString("Entrez␣un␣entier␣(fin␣avec␣0):␣");
            n = Terminal.lireInt(); // Modification variable du test
        }
        Terminal.ecrireStringln("La␣somme␣totale␣est:␣" + total);
    }
}
```

`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 le corps de la boucle, si avant la toute première itération, la condition $n! = 0$ est fausse. Dans l'exemple 1, c'est le cas, si lors de la première saisie, n vaut 0.

Étudions la trace d'une exécution de `Somme.java` en supposant saisis 5, 4, 7 et 0. Le tableau suivant montre l'évolution des variables pendant l'exécution. Comme d'habitude, nous allons consacrer une colonne à chaque variable, une à l'écran, une au clavier. Nous allons aussi insister sur la valeur de la condition d'arrêt en lui consacrant une colonne.

ligne	n !=0	total	n	écran	clavier
3		?	?		
5		?	?	Entrez un entier (fin avec 0) :	
6			5		5
7		0	5		
8	true	0	5		
9		5	5		
10		5	5	Entrez un entier (fin avec 0) :	
11		5	4		4
8	true	5	4		
9		9	4		
10		9	4	Entrez un entier (fin avec 0) :	
11		9	7		7
8	true	9	7		
9		16	7		
10		16	7	Entrez un entier (fin avec 0) :	
11		16	0		0
8	false	16	0		
13		16	0	La somme totale est 16	

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
```

3.4 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 3.2 – (pas de lien)

do "faire $suiteInstructions$
{
$suiteInstructions$
}
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` :

```
Terminal.ecrireString("Entrez un entier (fin avec 0): ");
n = Terminal.lireInt(); // Saisie d'initialisation pour n
total = 0;
while ( n !=0 ) {
    total = total + n;
    Terminal.ecrireString("Entrez un entier (fin avec 0): ");
    n = Terminal.lireInt(); // Nouvelle saisie de n
}
```

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.

```
total = 0;
do
```

```
        {
            Terminal.ecrireString("Entrez␣un␣entier␣(fin␣avec␣0):␣");
            n = Terminal.lireInt(); // Saisie de n
            total = total + n;
        }
    while ( n !=0 );
```

3.5 La notion de bloc

Un bloc est une suite d'instructions contenue entre des accolades. Ainsi, le corps d'une boucle est un bloc, les instructions entre les accolades d'un `if`, d'un `else`, d'un `else if` sont des blocs. Les instructions entre les accolades du `main` forment aussi un bloc.

La vie des variables est liée à un bloc. La variable commence à exister à partir de sa déclaration. Cette déclaration est située dans un bloc. La variable cesse d'exister à la fin de ce bloc.

Dans tous les programmes que nous vous avons présentés jusqu'ici, les variables étaient déclarées dans le plus gros bloc possible : celui du `main`. Leur vie se termine lorsque l'accolade fermante du `main` est exécutée, donc à la fin du programme. Mais si une variable est déclarée dans un bloc plus interne (dans le corps d'une boucle par exemple) sa vie se termine lorsque l'accolade fermante de ce bloc est exécutée.

Prenons un exemple.

```
public static void main (String[] args) {/*debut bloc 1*/
    int a=2;
    Terminal.ecrireStringln("valeur de a: "+ a );
    if (a==0){
        /* debut bloc 2 */
        int b=3+a;
        Terminal.ecrireStringln("valeur de b: "+ b );
    }  /* fin bloc 2*/
    else { /* debut bloc 3*/
        int c=3+a;
        Terminal.ecrireStringln("valeur de c: "+ c);
    } /* fin bloc 3*/
    Terminal.ecrireStringln("valeur de a: "+ a );
 } /* fin bloc 1*/
}
```

On a 3 blocs. Les blocs 2 et 3 sont à l'intérieur du bloc 1.

`a` est connue entre les lignes 4 et 14. Elle est donc connue aussi dans les bloc 2 et 3, puisqu'ils sont à l'intérieur du bloc 1.

`b` est connue entre les lignes 7 et 9.

`c` est connue entre les lignes 11 et 12.

3.5.1 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. Une variables déclarée à l'intérieur d'une boucle `for` n'existe qu'à l'intérieur du bloc qui suit cette boucle.

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

Listing 3.3 – (pas de lien)

```
somme = 0;
for (int i = 1; i <= n; i++) { // dans ce bloc, i existe
        somme = somme + i;
} // fin de vie de i
// ici, i n'existe pas
// Impossible par exemple d'ecrire Terminal.ecrireInt(i)
```

3.5.2 Boucles imbriquées

Le traitement que l'on doit répéter plusieurs fois au moyen d'un boucle est un bloc qui comporte une séquence d'instructions à exécuter. Ces instructions peuvent être une affectation ou un appel de méthode, mais également un `if` ou une autre boucle. Quand il y a une boucle à l'intérieur d'une boucle, on parle de boucles imbriquées.

En voici un exemple.

```
public class ExImb{
    public static void main(String[] args){
        for (int i=0; i<5; i=i+1){
            for (int j=0; j<8; j=j+1){
                Terminal.ecrireInt(j);
            }
            Terminal.sautDeLigne();
        }
    }
}
```

La première boucle qui utilise la variable `i`, consiste à éxécuter la deuxième boucle (celle qui utilise `j`) puis à passer à la ligne au moyen de la méthode `Terminal.sautDeLigne()`. La première boucle s'exécute 5 fois, donc le programme affiche 5 lignes à l'écran. La deuxième boucle consiste à afficher les chiffres compris entre 0 et 7.

Le résultat de l'exécution est donc le suivant :

```
> java ExImb
01234567
01234567
01234567
01234567
01234567
```