Chapitre 3

Exécution des programmes

Le *sens* d'un programme est son comportement lorsqu'il est exécuté. Un programme est composé d'une suite d'instructions, qui sont exécutées les unes après les autres, *séquentiellement*. Chaque instruction peut effectuer trois sortes d'actions : d'entrée ou de sortie des données, de changement sur les variables, c'est à dire sur les cases mémoires qui leur sont associées. Ainsi chaque instruction modifie l'état mémoire de la machine, et un programme est une suite de modifications successives de l'état mémoire de la machine. Pour être bref, l'exécution d'un programme est une suite d'états mémoires.

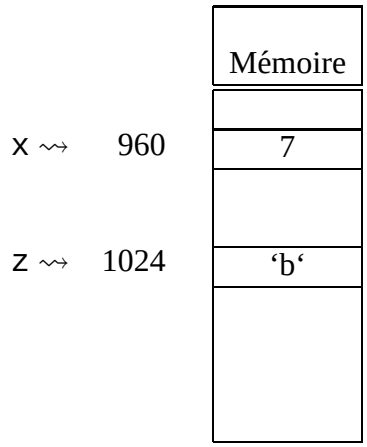
Comprendre un programme, c'est savoir comment il s'exécute et donc savoir retracer pas à pas cette suite d'états mémoires consécutifs. Cette suite d'états est connue aussi sous le nom de *trace d'une exécution*.

Nous allons étudier ces notions en détail dans ce chapitre.

3.1 Variables et mémoire

Les variables sont utilisées pour stocker les données du programme. A chaque variable correspond un emplacement en *mémoire*, où la donnée est stockée, et un nom qui sert à désigner cette donnée, ou ses valeurs futures, tout au long du programme.

La mémoire est divisée en emplacements, appelés *mots*, possèdant chacun une *adresse* qui permet d'acceder au contenu du mot de manière directe. Selon la nature de la valeur contenue dans une variable, elle peut occuper un ou plusieurs mots de suite. Ainsi, chaque variable correspond à son adresse de stockage en mémoire, et par voie de conséquence, à la valeur qui s'y trouve. Nous représentons la mémoire graphiquement :



Dans cet exemple, la variable `x` de type `int` est stockée à l'adresse 960, et contient la valeur 7, alors qu'à la variable `z`, de type `char`, correspond l'adresse 1024, dont le contenu est le caractère `'b'`. Le programme peut ensuite consulter (*lire*) le contenu d'une variable ou le modifier en *écrivant* une nouvelle valeur dans l'emplacement associé.

État de la mémoire

Nous désignons par ce terme, l'ensemble des valeurs stockées en mémoire, pour les variables du programme, à un moment donné de l'exécution. Il s'agit d'une sorte de photo de la mémoire du programme. Le dessin plus haut est un exemple, où la variable `x` vaut 7, et `z` vaut `'b'`. Nous l'appellerons également *Environnement du programme*.

Consulter la valeur d'une variable

Quel est le sens à donner à une variable si elle apparaît dans une action où sa valeur est nécessaire ? Par exemple, quel est le sens de `x` dans `x+2` ? La valeur d'une variable dépend de l'état courant de la mémoire.

- `System.out.println(x)` affiche non pas la lettre `x`, mais le contenu de la variable `x` en mémoire, au moment où cet instruction est exécutée.
- L'expression `x + 2` s'évalue dans la valeur courante de `x` augmentée de 2.

Exercices : Considérez le code suivant :

```
y = x + 3;
Terminal.ecrireInt(z);
Terminal.lireDouble(w);
m = 5;
n = n + 2;
```

1. Dans ces actions, quelles sont les variables ayant besoin d'intitialisation préalable, et quelles sont celles qu'on peut ne pas initialiser ?
2. Donnez les déclarations et initialisations nécessaires à la compilation de ces actions.

3.2 Comportement des déclarations de variables

Le sens d'une déclaration de variable est de *changer l'état de la mémoire* par :

- *Création de l'Environnement du programme* (voir notion introduite plus haut).
- *Allocation* d'un emplacement libre en mémoire, de taille suffisante pour stocker une valeur du type déclaré. Les valeurs succésivement acquises par la variable seront stockées dans cet emplacement. **Attention** : Un mot de la mémoire est une suite de bits, chacun est nécessairement 0 ou 1. Il est donc impossible qu'un emplacement, et a fortiori une variable, ne contienne "rien". Par exemple, une fois la place pour une variable entière reservée, il y a dans cet emplacement un entier (qu'on ne connaît pas forcément). Par ailleurs, chaque emplacement ne permet de stocker qu'une valeur à la fois. Ainsi, une nouvelle valeur sera réécrite sur la valeur précédente, à la place de celle-ci.
- *Liaison* (ou association) de la valeur initiale lorsqu'elle est donnée. L'association entre une variable et sa valeur est appélé *liaison*. Ainsi, si un moment donné, la valeur 5 est stockée dans la variable `x`, on dira que `x` *est liée* à la valeur 5. Si une variable est déclarée sans valeur initiale, on dira que sa valeur est inconnue au moment de la déclaration (mais ce n'est pas rien !). La valeur inconnue est notée `?`

Exemple : Considérons les déclarations :

```
int x,y;
int a = 0;
int b = 0;
int z = 0;
```

L'état de la mémoire ou environnement du programme, après déclarations, est donné par :

x	?
y	?
a	0
b	0
z	0

rôle du type de la variable : Le fait de donner un type à une variable lors de sa déclaration permet de prévoir combien d'espace mémoire il faut réserver pour la variable. Par exemple, un `double` est plus gros a stocker qu'un `int`.

Exercices : Quel est l'environnement créé (mémoire) après les déclarations suivantes ?

```
char b = 'c';
char c = b;
boolean d;
int x,y = 3;
double m = 3.5;
```

3.3 Comportement d'une affectation

C'est une instruction qui modifie la valeur d'une *variable* en mémoire [1].
La syntaxe d'une affectation a la forme :

nom_variable = *expression* ;

– *Comportement* : Modifie en mémoire le contenu de la variable à gauche, avec la valeur qui résulte du calcul de l'expression à droite du symbole =

– *Exécution* : suit les pas suivants. Prenons l'exemple de l'exécution de `x = x+2` dans un état mémoire où `x` vaut 5 .

1. *évaluation de l'expression à droite dans l'environnement courant*
   évaluons `x+2` avec `x` vaut `5` $\Rightarrow$ 5+2 $\Rightarrow$ 7,
2. *Modification de la variable* en mémoire : `x` vaut 7,

Trace d'une exécution

Nous allons maintenant suivre pas à pas l'execution d'un programme comportant des declarations et des affectations. Pour y parvenir, nous allons faire la *trace de l'exécution* de ce programme. Nous donnons l'état de la machine après exécution de chacune des instructions du programme, autrement dit, l'état des entrées, l'état des sorties, et de la mémoire (les valeurs des variables du programme). Comme notre programme ne fait pas de sd'entrées ni de sorties, la dernière case sera vide.

Listing 3.1 – (lien vers le code brut)

```
1 public class T1 {
2     public static void main (String[] args) {
3         int x;
4         int y=2;
5         x=3+y;
6         x=x+1;
7         y=x*2;
8     }
9 }
```

Après l'instruction	Mémoire		Entrées	Sorties
	x	y		
Déclarations	?	2		
6	5	2		
7	6	2		
8	6	12		

[1] En java, l'affectation retourne aussi un résultat. Dans ce cours, ceci ne nous sera pas utile.

3.4 Durée de vie d'une variable

Lorsqu'on déclare une variable, on associe un nom (le nom de la variable) à une case mémoire. On peut ensuite désigner cette case mémoire en utilisant le nom de la variable. Cette association entre un nom et une case mémoire a une durée de vie limitée :

- La **création de l'association** entre le nom d'une variable et la case mémoire se fait lors de l'exécution de la déclaration.
- la **destruction de l'association** se fait lorsque la dernière instruction du *bloc* contenant la déclaration est exécutée.

3.4.1 La notion de bloc

Qu'est ce qu'un bloc ?
En Java, ils sont faciles a reconnaitre : **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 `elseif` sont des blocs. Les instructions entre les accolades du `main` forment aussi un bloc.

Un bloc est un environnement local de déclarations de variable. Ceci signifie que toute declaration n'est connue qu'à l'intérieur du bloc ou elle est déclarée.

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

Listing 3.2 – (lien vers le code brut)

```
1     public static void main (String[] args) {/*debut bloc 1*/
2           int a=2;
3           Terminal.ecrireStringln("valeur de a: "+ a );
4           if (a==0){
5               /* debut bloc 2 */
6               int b=3+a;
7               Terminal.ecrireStringln("valeur de b: "+ b );
8           }       /* fin bloc 2*/
9           else {   /* debut bloc 3*/
10              int c=3+a;
11              Terminal.ecrireStringln("valeur de c: "+ c);
12          }       /* fin bloc 3*/
13         Terminal.ecrireStringln("valeur de a: "+ a );
14      }       /* fin bloc 1*/
15  }
```

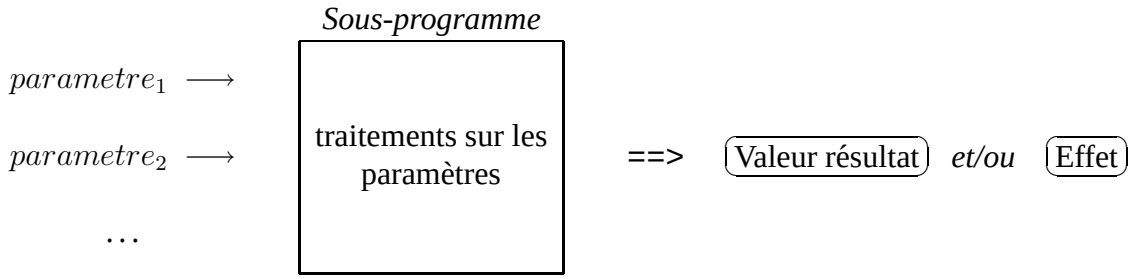
On a 3 blocs. Les blocs 2 et 3 sont à l'intérieur du bloc 1.
`a` est connue entre les lignes 4 et 14. Elle est donc connue aussi dans les bloc 2 et 3, puisqu'ils sont à l'intérieur du bloc 1.
`b` est connue entre les lignes 7 et 9.
`c` est connue entre les lignes 11 et 12.

3.5 Comportement d'un appel de méthode

La syntaxe d'un appel de sous-programme est de la forme :

Nom_sous_programme ($arg_1, arg_2, \ldots, arg_n$)

Schématiquement, un *sous-programme* est une boîte noire capable de réaliser des traitents sur les *arguments* ou entrées du sous-programmes. Si l'on désire effectuer les traitements, on *invoque* ou *appelle* le sous-programme en lui passant les données à traiter, qui peuvent varier pour chaque appel. En sortie on obtient : ou bien une valeur, résultat des calculs effectués, ou bien un *changement dans l'état de la machine* appelé aussi *effet* ; parfois, les deux.



Exemples :

- `Math.min(3,8)` : appel de la fonction `Math.min` pour calculer le plus petit de ses deux arguments. Renvoie `3` en résultat.
- `Terminal.ecrireStringln("Bonjour")` : appel de la fonction `Terminal.ecrireStringln` du fichier `Terminal.java`. Provoque un *effet* de sortie avec l'affichage de `Bonjour`. Ne renvoie pas de valeur résultat.

L'appel d'un sous-programme se traduit *grosso modo* en plusieurs actions :

1. L'exécution séquentielle du programme s'arrête au point de l'appel et attend la fin du sous-programme pour reprendre.
2. Les arguments passés à l'appel sont transmis au sous-programme en tant qu'entrées.
3. Les traitements du sous-programme s'appliquent sur ces entrées,
4. S'il y a une valeur résultat elle est *renvoyée* au point de l'appel.

5. L'exécution du programme reprend au point d'arrêt. Le cas écheant, on recupère la valeur renvoyée pour la suite des calculs.

Considérons le morceau de programme :

```
x = 5;  // x recoit la valeur 5
x = 3 + Math.min(x,10) + 2;
        ^             ^

        |             |
point d'appel    point de retour avec resultat 5
```

- *1ère instruction* : on donne à `x` la valeur `5`,
- *2ème instruction* : avant d'évaluer l'expression `3 + Math.min(x,10) + 2`, on doit traiter l'appel de fonction :
  1. On suspend le calcul de l'expression,
  2. On appelle la fonction `Math.min` :
     - on lui *transmet* `5,10` en arguments,
     - le *contrôle du programme* est donné aux actions dans la fonction, qui calcule 5 ,
     - le *contrôle du programme* revient au point où il avait été suspendu, en retournant la valeur résultat 5,
  3. On reprend l'évaluation de l'expression avec la valeur retournée :
     `3 + Math.min(x,10) + 2` $\Rightarrow$ `3 + 5 + 2` $\Rightarrow$ `10`.

  On termine l'exécution de la 2ème instruction en modifiant la valeur de `x <- 10`.

Trace d'une exécution

Nous décrivons le comportement d'une exécution pour notre exemple. Nous donnons l'état de la machine après exécution de chacune des instructions du programme, autrement dit, l'état des entrées, l'état des sorties, et de la mémoire (les valeurs des variables du programme). Comme notre programme saisit des valeurs au clavier, nous supposons que des valeurs arbitraires sont effectivement tapées. Afin de rendre plus facile la description, nous ajoutons des numéros de ligne au programme étudié :

Listing 3.3 – (lien vers le code brut)

```
1  public class Conversion {
2      public static void main (String[] args) {
3
4          double euros, francs;
5
6          Terminal.ecrireStringln("Somme en euros? ");
7          euros = Terminal.lireDouble();
8          francs = euros * 6.559;
9          Terminal.ecrireStringln("La somme en francs: "+ francs);
10 }
```

Après l'instruction	Mémoire		Entrées	Sorties
	`euros`	`francs`		
Déclarations	?	?		
6	?	?		`Somme en euros?`
7	10.0	?	10	
8	10.0	`65.59`		
9	10.0	`65.59`		`La somme en francs : 65.59`

Évolution de la mémoire :

`euros`	*?*	10.0
`francs`	*?*	65.59

3.6 Comportement du `if`

Rapellons l'essentiel sur cette instruction :

```
if  (cond) {                          ''si cond est vraie, faire suite1
    suite1
} else {                                sinon, faire suite2 ''
    suite2
}
```

Comportément : Selon la valeur de la condition, on réalise les actions de l'une des deux suites d'instructions `suite1` ou `suite2`. La séquence `suite1` est exécutée si `cond` est vraie, la séquence `suite2` est exécutée si `cond` est fausse. A chaque fois, exactement une des deux séquences est exécutée.

Listing 3.4 – (lien vers le code brut)

```
1  public class PrixTTC {
2      public static void main (String[] args) {
3
4          double pHT,pTTC;
5          int t;
6          Terminal.ecrireStringln("Entrer le prix HT: ");
7          pHT = Terminal.lireDouble();
8
9          Terminal.ecrireStringln("Entrer taux(0,1) : ");
10         t = Terminal.lireInt();
11         if (t==0){
```

```
12              pTTC=pHT + (pHT*0.196);
13            }
14            else {
15              pTTC=pHT + (pHT*0.05);
16            }
17            Terminal.ecrireStringln("La somme TTC: "+ pTTC );
18            }
19  }
```

Trace d'une exécution

Voici la trace d'exécution de ce programme :

Après l'instruction	Mémoire			Entrées	Sorties
	pHt	t	pTTC		
Déclarations	?	?	?		
6	?	?			Entrer le prix HT :
7	10.0	?		10.0	
9	?	?			Entrer le taux(0,1) :
10	10.0	1		1	
11	10.0	1			
14	10.0	1			
15	10.0	1	10.5		
17	10.0	1	10.5		La somme TTC : 10.5

3.7 Comportement des boucles

Une boucle `while` en Java a la forme :

Listing 3.5 – (lien vers le code brut)

```
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.

Exemple 1 :

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

Listing 3.6 – (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.

Trace de `Somme.java`

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. La toute première colonne donne la condition d'entrée à la boucle et les noms des variables modifiées. La colonne *init* donne les valeurs des variables avant la première itération. Chaque colonne *itération* k donne les renseignements suivants pour l'itération k :

- *pour le test* `(n !=0)` : sa valeur avant l'entrée à l'itération k ;
- *pour les variables de la boucle* `total` et `n` : leurs valeurs en fin d'itération k.

	init	*itération* 1	*itération* 2	*itération* 3	*itération* 4
`(n !=0)`		`(5 != 0)`	`(4 != 0)`	`(7 != 0)`	`(0 != 0)`
`total`	0	0+5	0+5+4	0+5+4+7	*arrêt*
`n`	5	4	7	0	*arrêt*

Par exemple, le premier test réalisé est `(5 != 0)`, et après l'itération 1, `total` vaut 5 et `n` vaut 4. Lors de la dernière itération, le nombre 0 est saisit dans `n`, et `total` accumule la somme des valeurs précédentes de `n` (0+5+4+7). Au prochain test, la condition `(0 != 0)` est fausse et la boucle s'arrête. Le programme affiche la valeur 16 pour `total`. 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
```