MCours.com

Chapitre 2

L'essentiel pour commencer

L'objectif de ce chapitre est de vous donner les bases du langage Java vous permettant d'écrire vos propres programmes. Nous y passons donc rapidement en revue les éléments de base du langage Java, en vous expliquant *à quoi ils servent* et *comment ils s'écrivent*. Nous ne serons évidemment pas exhaustifs, c'est pourquoi les notions abordées dans ce chapitre feront l'objet soit de chapitres particuliers dans la suite du cours, soit d'annexes dans les notes.

Pour illustrer notre propos, nous allons reprendre l'exemple de la conversion d'une somme des euros vers les francs :

Problème : Calculer et afficher la conversion en francs d'une somme en euros saisie au clavier.

Listing 2.1 – (lien vers le code brut)

```
public class Conversion {
    public static void main (String[] args) {
        double euros, francs;
        Terminal.ecrireStringln("Somme en euros? ");
        euros = Terminal.lireDouble();
        francs = euros * 6.559;
        Terminal.ecrireStringln("La somme en francs: "+ francs);
    }
}
```

2.1 Structure générale d'un programme Java

Ce programme a un squelette identique à tout autre programme Java. Le voici :

Listing 2.2 – (pas de lien)

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

Autrement dit :

- Tout programme Java est composé au minimum d'une *classe* (mot-clé `class`) dite *principale*, qui elle même, contient une *méthode* de nom `main` (c'est aussi un mot-clé). Les notions de *classe* et de *méthode* seront abordées plus tard.
- **public, class, static, void, main** : sont des mots reservés c'est à dire qu'ils ont un sens particuliers pour Java. On ne peut donc pas les utiliser comme nom pour des classes, des variables, etc...
- La **classe principale** : celle qui contient la méthode `main`. Elle est déclarée par
  public class *Nom_classe*
  C'est vous qui choisissez le nom de la classe.
  Le code qui définit une classe est délimité par les caractères { et }
- **Nom du programme :** Le nom de la classe principale donne son nom au programme tout entier et doit être également celui du fichier contenant le programme, complété de l'extension `.java`
- La **méthode** `main` : obligatoire dans tout programme Java : c'est elle qui "commande" l'exécution. Définie par une suite de déclarations et d'actions délimitées par { et }. Pour l'instant, et jusqu'à ce que l'on sache définir des sous-programmes, c'est ici que vous écrirez vos programmes.

Autrement dit, pour écrire un programme que vous voulez appeler `"truc"`, ouvrez un fichier dans un éditeur de texte, appelez ce fichier `"truc.java"` ; Ecrivez dans ce fichier le squelette donné plus haut, puis remplissez les ... :

2.2 comprendre le code

Le programme Java exprime la chose suivante dans le langage Java (c'est-à-dire au moyen de phrases comprises par le compilateur Java) :
(ligne 4 :) déclarer 2 variables appelées `euros` et `francs` destinées à contenir des réels **puis**
(ligne 6 :) afficher à l'écran la phrase `Somme en euros? :` **puis**
(ligne 7 :) récupérer la valeur entrée au clavier et la stocker dans la variable `euros`, **puis**
(ligne 8 :) Multiplier la valeur de la variable `euros` par 6.559 et stocker le résultat de ce calcul dans la variable `francs`, **puis**
(ligne 9 :) afficher à l'écran la valeur de `francs`

La première chose importante à remarquer est que pour écrire un programme, on écrit une suite d'ordres séparés par des `;`. L'exécution du programme se fera en exécutant d'abord le premier ordre, puis le second, etc. C'est ce qu'on appelle le *principe de séquentialité de l'exécution*.

Dans ce programme on a trois catégories d'ordres :
- des déclarations de variables. Les déclarations servent à donner un nom à une case mémoire, de façon a pouvoir y stocker, le temps de l'exécution du programme, des valeurs. Une fois qu'une variable est déclarée et qu'elle possède une valeur, on peut consulter sa valeur.
- des instructions d'entrée-sortie. Un programme a bien souvent (mais pas toujours) besoin d'informations pour lui viennent de l'extérieur. Notre programme calcule la valeur en francs d'une somme en euros *qui est donnée au moment de l'exécution* par l'utilisateur. On a donc besoin de dire qu'il faut faire `entrer` dans le programme une donnée par le biais du clavier. Il y a en Java, comme dans tout autre langage de programmation, des ordres prédéfinis qui permettent de faire cela (aller chercher une valeur au clavier, dans un fichier, sortir du programme vers l'écran ou un fichier une valeur etc...).
- l'instruction d'affectation (=) qui permet de manipuler les variables déclarées. Elle permet de mettre la valeur de ce qui est a droite de = dans la variable nommée à gauche du =.

2.3 Les variables

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 tout au long du programme.
Une variable doit être déclarée dans le programme. On peut alors consulter sa valeur ou modifier sa valeur.

2.3.1 déclaration

Nous avons déjà vu une déclaration : `double euros, francs;`
Cette déclaration déclare 2 variables à la fois de nom respectif `euros` et `francs`. La forme la plus simple de déclaration de variables est la déclaration d'une seule variable à la fois. On aurait pu remplacer notre déclaration précédente par celles ci :

```
double euros;
double francs;
```

Ici, on déclare d'abord la variable `francs` puis la variable `euros`.

le nom des variables

`euros` ou `francs` sont les noms des variables et ont donc été choisis librement par l'auteur du programme. Il y a cependant quelques contraintes dans le choix des symboles constituant les noms de variables. Les voici :

- Les noms de variables sont des *identificateur* c'est à dire commencent nécessairement par une lettre, majuscule ou minuscule, qui peut être ou non suivie d'autant de caractères que l'on veut parmi l'ensemble a . . .z, A . . .Z, 0..9, _, `$` *Unicode*.
- Un nom de variable ne peut pas être un mot reservé : (`abstract, boolean, if, public, class, private, static`, etc).
- Certains caractères ne peuvent pas apparaître dans un nom de variable (`^, [, ], {, +, -,` . . .).

Exemples : `a`, `id_a` et `X1` sont des noms de variables valides, alors que `1X` et `X-X` ne le sont pas.

le type de la variable

Dans notre exemple de déclaration `double euros;`, le mot `double` est le nom d'un type prédéfini en Java. Un type est un ensemble de valeurs particulières connues de la machine. Nous allons pendant quelques temps travailler avec les types java suivants :

- Le type `int` désigne l'ensemble de tous les nombres entiers représentables en machine sur 32 bits (31 plus le signe) $\{-2^{31}, \ldots, 2^{31}\}$. Les éléments de cet ensemble sont `0, 1, 2...`
- le type `double` désigne les réels (à précision double 64 bits). Les éléments de cet ensemble sont `0.0, 0.1, ...18.58 ...`
- Le type `boolean` modélise les deux valeurs de vérité dans la logique propositionnelle. Ses éléments sont `true` (vrai) et `false` (faux).
- Le type `char` modélise l'ensemble des caractères Unicode (sur 16 bits). Ses éléments sont des caractères entourés de guillemets simples, par exemple : `'a'`, `'2'`, `'@'` ;

– le type `string` modélise les chaînes de caractères. Ses éléments sont les suites de caractères entourées de guillemets doubles : `"coucou"`, `"entrer une somme en euros :?"`, `"a"`...

syntaxe des déclarations de variables

Ainsi, pour déclarer une variable, il faut donner un nom de type parmi `int, double, char, boolean, string` suivi d'un nom de variable que vous inventez.

Pour déclarer plusieurs variables de même type en même temps, il faut donner un nom de type suivi des noms de vos variables (séparés par des virgules).

C'est ce qu'on appelle la *syntaxe* Java des déclarations de variables. Il faut absolument se conformer à cette règle pour déclarer des variables en Java, faute de quoi, votre code ne sera pas compris par le compilateur et produira une erreur. Ainsi, `int x; string "coucou"` sont corrects alors que `entier x;` ou encore `x int;` seront rejetés.

Execution des déclarations de variables

L'exécution d'un programme, rapellons le, consiste en l'exécution successive de chacune de ses instructions.

Que se passe-t-il lorsqu'une déclaration est rencontrée ? Une place mémoire libre de taille suffisante pour stocker une valeur du type de la variable est recherchée, puis le nom de la variable est associé a cette case mémoire. Ainsi, dans la suite du programme, le nom de la variable pourra être utilisé et désignera cet emplacement mémoire.

Une question un peu subtile se pose : combien de temps cette liaison entre le nom de la variable et la case mémoire est elle valable ? Autrement dit, jusqu'à quand la variable est-elle connue ? Pour l'instant, nous pouvons répondre simplement : cette liaison existe durant l'exécution de toutes les instructions qui suivent la déclaration de la variable et jusqu'à la fin du programme.

Cette notion (la portée des variables) se compliquera lorsque nous connaîtrons plus de choses en Java, notamment la notion de bloc. Nous y reviendrons à ce moment là.

2.3.2 Affecter une valeur à une variable

Une fois qu'une variable a été declarée, on peut lui donner une valeur, c'est à dire mettre une valeur dans la case mémoire qui lui est associée. Pour cela, il faut utiliser l'instruction d'affectation dont la syntaxe est

nom_variable = expression ;

par exemple `x=2;`

Il est de bonne habitude de donner une valeur initiale à une variable dès qu'elle est déclarée. Java nous permet d'affecter une valeur à une variable au moment de sa déclaration :
par exemple `int x=2;`

Des valeurs du bon type

A droite de =, on peut mettre n'importe quelle valeur *du bon type*, y compris des variables. Comme 2 est une valeur du type `int`, notre exemple ne sera possible que si plus haut dans notre programme,

on a la déclaration `int x;` [1].

Les expressions

Les valeurs ne sont pas forcément simples, elles peuvent être le résultat d'un calcul. On pourrait par exemple écrire `x=(18+20)*2;`. On peut écrire l'expression `(18+20)*2` car + et * sont des *opérateurs* connus dans le langage Java, désignant comme on s'y attend l'addition et la multiplication et que ces opérateurs, appliqués à des entiers, calculent un nouvel entier. En effet, $18 + 20$ donne 38 (un entier) et 38×2 donne 76 qui est un entier.

- Pour construire des expressions arithmétiques, c'est à dire des expressions dont le résultat est de type `int` ou `double` on peut utiliser les opérateurs :`+,-,*,/,%(reste de la division entière)`
- Pour construire des expressions booléennes (i.e dont le résultat est de type `boolean`, on peut utiliser les opérateurs : `&&`(et), `||` (ou), `!`(non).
- Les opérateurs de comparaison permettent de comparer deux *valeurs* ou *expressions* toutes les deux de de type numérique ou `char` et renvoient une valeur booléenne en résultat.
  `==` (egalité), `<` (plus petit), `>` (plus grand), `>=` (plus grand ou égal), `<=` (plus petit ou égal), `!=` (différent).

Exécution d'une affectation

L'exécution d'une affectation se fait en deux temps :

1. On calcule le résultat de l'expression **à droite** de = (on dit évaluer l'expression). Si c'est une valeur simple, il n'y a rien a faire, si c'est une variable, on va chercher sa valeur, si c'est une expression avec des opérateurs, on applique les opérateurs à leurs arguments. Cela nous donne une valeur qui doit être du même type que la variable à gauche de =.
2. On met cette valeur dans l'emplacement mémoire associé à la variable **à gauche** de =.

C'est ce qui explique que l'on peut écrire le programme suivant :

Listing 2.3 – (lien vers le code brut)

```
public class essaiVariable {
    public static void main (String[] args) {
      int x;
      int y;
      y=2;
      x=y+5;
      x=x+2;
    }
}
```

Les lignes 3 et 4 déclarent les variables `x` et `y`. Elle sont donc connues dans la suite du programme. Puis (l5) la valeur 2 est donnée à `y`. Ensuite est calculé `y+5`. La valeur de `y` en ce point de l'exécution est `2` donc `y+5` vaut `7`. A l'issue de l'exécution de la ligne 6, `x` vaut `7`. Finalement, (l7), on évalue `x+2`. `x` à ce moment vaut `7` donc `x+2` vaut `9`. On donne à `x` la valeur `9`.

[1] Ceci n'est pas tout à fait exact. Nous apprendrons plus tard que nous pouvons affecter à une variable des valeurs d'autres types qui sont de la même famille (conversion implicite de type)

Cet exemple illustre le fait qu'une variable peut (puisque l'exécution d'un programme est séquentielle) avoir plusieurs valeurs successives (d'où son nom de variable), et que l'on peut faire apparaître une même variable à gauche et à droite d'une affectation : on met à jour son contenu en tenant compte de la valeur qu'elle contient juste avant.

2.4 Les appels de méthodes prédéfinies

Revenons à notre exemple de départ, la conversion en francs d'une somme en euros. Nous avions expliqué que les lignes 6 et 7 du programme effectuaient des opérations d'entrée sortie. Nous allons analyser cela plus en détail.

2.4.1 La méthode `Terminal.ecrireStringln`

La ligne 6 donne l'ordre d'afficher à l'écran le message `somme en euros ? :`
Elle le fait en utilisant un programme tout fait que nous avons écrit pour vous. On appelle cela en Java une *méthode*.
Ce programme s'appelle `ecrireStringln`. Il fait partie de la classe `Terminal`.

A condition d'avoir copié le fichier `Terminal.java` contenant cette classe dans votre répertoire de travail, vous pourrez utiliser autant de fois que vous le désirez cette méthode. Utiliser une méthode existante dans un programme s'appelle *faire un appel* de la méthode.

Pour fonctionner, cette méthode a besoin que l'utilisateur, lorsqu'il l'utilise, lui transmette une information : la chaîne de caractère qu'il veut afficher à l'écran. Cette information transmise par l'utilisateur de la méthode est ce qui se trouve entre les parenthèses qui suivent le nom de la méthode. C'est ce qu'on appelle *l'argument* ou le *paramètre* de la méthode.
On a ici utilisé `Terminal.ecrireStringln` pour afficher le message `Somme en euros ? :` en faisant l'appel `Terminal.ecrireStringln("Somme en euros ? :);`
Pour afficher coucou, il faut faire l'appel `Terminal.ecrireStringln("coucou")`.

Lors d'un appel, vous devez nécessairement transmettre une et une seule chaîne de caractère de votre choix à `Terminal.ecrireStringln` : c'est l'auteur de la méthode qui a fixé le nombre, le type et l'ordre des arguments de cette méthode, lorsqu'il l'a écrit. Ainsi on ne pourra écrire `Terminal.ecrireStringln(` ni `Terminal.ecrireStringln("coucou","bidule")`. Ces 2 lignes provoqueront des erreurs de compilation.

2.4.2 La méthode `Terminal.lireInt`

La ligne 7 donne l'ordre de récupérer une valeur au clavier.

```
euros=Terminal.lireDouble();
```

Cette ligne est une affectation =. On trouve à droite du = un appel de méthode.

La méthode appelée s'appelle `lireDouble` et se trouve dans la classe `Terminal`.

Le fait qu'il n'y ait rien entre les parenthèses indique que cette méthode n'a pas besoin que l'utilisateur lui fournisse des informations. C'est une méthode sans arguments.

En revanche, le fait que cet appel se trouve à droite d'une affectation indique que le résultat de son exécution produit un résultat (celui qui sera mis dans la variable `euros`). En effet, ce résultat est la valeur provenant du clavier. C'est ce qu'on appelle la *valeur de retour* de la méthode.

Comme pour les arguments de la méthode, le fait que les méthodes retournent ou pas une valeur est fixé par l'auteur de la méthode une fois pour toutes. Il a fixé aussi le type de cette valeur de retour. Une méthode, lorsqu'elle retourne une valeur, retourne une valeur toujours du même type. Pour `lireDouble`, cette valeur est de type `Double`.

Lorsqu'ils retournent un résultat, les appels de méthode peuvent figurer dans des expressions.
Exemple : La méthode `Math.min` :prends 2 arguments de type int et retourne une valeur de type `int` : le plus petit de ses deux arguments. Ainsi l'instruction `x = 3 + (Math.min(4,10) + 2);` donne à `x` la valeur `9` car `3+(4+2)` vaut `9`.
L'instruction `x = 3 + (Terminal.lireInt() + 2);` a aussi un sens. Elle s'exécutera de la façon suivante : Pour calculer la valeur de droite, l'exécution se mettra en attente qu'une touche du clavier soit pressée (un curseur clignotant a l'écran indiquera cela). Dès que l'utilisateur presse une touche, le calcul de `Terminal.lireInt()` se termine avec pour résultat cette valeur. Imaginons que l'utilisateur ait pressé 6. 3+6+2 donne 11. La valeur de `x` sera donc 11.

Que se passera-t-il si l'utilisateur presse une touche ne correspondant pas à un entier ? Une erreur se produira, et l'exécution du programme tout entier sera arrêtée. Nous verrons plus tard qu'il y a un moyen de récupérer cette erreur pour relancer l'exécution.

2.4.3 Les appels de méthodes en général

De nombreuses classes contenant des méthodes existent en Java, soit dans la bibliothèque commune au langage, soit dans des répertoires particulier qu'il faut alors indiquer (nous verrons cela plus tard).

Toute méthode existante possède un nom, appartient à une classe, possède des arguments dont le nombre le type et l'ordre sont fixés. Une méthode peut renvoyer une valeur ; le type de la valeur de retour est alors fixé.

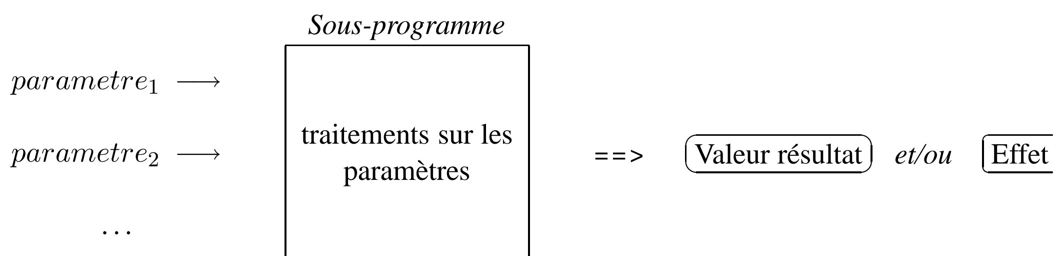
On trouve ces informations dans la documentation associée aux classes. Pour `Terminal.ecrireStringln` ces informations seraient données de la façon suivante :
`void ecrireStringln(String ch)` : `void` avant le nom indique que cette méthode ne renvoie pas de résultat, `(String ch)` indique qu'elle a un seul argument de type `String`.
`double lireDouble()` : `double` avant le nom indique que cette méthode renvoie un résultat de type `Double`, `()` indique qu'elle a n'a pas d'argument
Pour la méthode `min` de la classe `Math` on trouverait :
`int min(int a, int b)` :prends 2 arguments de type int et retourne une valeur de type `int` : le plus petit de ses deux arguments.

Pour appeler une méthode, il faut connaître toutes ces informations. l'appel de méthode sera conforme à la syntaxe suivante :

MCours.com


```
NomClasse.NomMethode(arg1,...,argn);
```

Ainsi, on peut voir un *sous-programme* comme une boîte noire capable de réaliser des traitements sur les *arguments* ou entrées du sous-programme. 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.



2.5 Les méthodes prédéfinies d'entrée-sortie

Disons maintenant quelques mots sur les méthodes d'entrée-sortie.

Java est un langage qui privilégie la communication par interfaces graphiques. En particulier, la saisie des données est gérée plus facilement avec des fenêtres graphiques dédiées, des boutons, etc. Mais, pour débuter en programmation, il est beaucoup plus simple de programmer la saisie et l'affichage des données au terminal : la saisie à partir du clavier et l'affichage à l'écran. Nous vous proposons de suivre cette approche en utilisant la bibliothèque `Terminal`.

La bibliothèque `Terminal`

La classe `Terminal` (écrite par nous), regroupe les principales méthodes de saisie et d'affichage au terminal pour les types predéfinis que nous utiliserons dans ce cours : `int`, `double`, `boolean`, `char` et `String`. Le fichier source `Terminal.java`, doit se trouver présent dans le même répertoire que vos programmes. Pour l'employer, il suffit de faire appel à la méthode qui vous intéresse précédé du nom de la classe. Par exemple, `Terminal.lireInt()` renvoie le premier entier saisit au clavier.

Saisie avec `Terminal` : Se fait toujours par un appel de méthode de la forme :

`Terminal.lire`*`Type`*`();`

où *Type* est le nom du type que l'on souhaite saisir au clavier. La saisie se fait jusqu'à validation par un changement de ligne. Voici la saisie d'un `int` dans `x`, d'un `double` dans `y` et d'un `char` dans `c` :

```
int x; double y; char c; // Declarations
x = Terminal.lireInt();
y = Terminal.lireDouble();
c = Terminal.lireChar();
```

Méthodes de saisie dans `Terminal` :

– `Terminal.lireInt()`

MCours.com

- `Terminal.lireDouble()`
- `Terminal.lireChar()`
- `Terminal.lireBoolean()`
- `Terminal.lireString()`

Affichage avec `Terminal` : Les méthodes d'affichage traitent les mêmes types que celles de lecture. Il y a deux formats d'affichage :

- `Terminal.ecrireType(v);` affiche la valeur `v` qui doit être du type indiqué par le nom de la méthode.
- `Terminal.ecrireTypeln(v);` affiche comme avant plus un saut à la ligne.

```
Terminal.ecrireInt(5);
Terminal.ecrireInt('a');
Terminal.ecrireIntln(5);
Terminal.ecrireDoubleln(1.3);
```

Ce programme affiche :

```
5a5
5
1.3
```

L'affichage de messages, ou chaînes de caractères (type `String`), autorise l'adjonction d'autres messages ou valeurs en utilisant l'opérateur de *concaténation* `+`.

```
Terminal.ecrireString("bonjour " + 5 + 2 );   ---> affiche:  bonjour 52
Terminal.ecrireString("bonjour " + (5 + 2) ); ---> affiche:  bonjour 7
Terminal.ecrireString(5 + 2);   ---> Erreur de Typage!
```

Méthodes d'affichage :

- `Terminal.ecrireInt(n);`
- `Terminal.ecrireDouble(n);`
- `Terminal.ecrireBoolean(n);`
- `Terminal.ecrireChar(n);`
- `Terminal.ecrireString(n);`
- `Terminal.ecrireIntln(n);`
- `Terminal.ecrireDoubleln(n)`...

Affichage avec la bibliothèque `System`

La bibliothèque `System` propose les mêmes fonctionnalités d'affichage au terminal pour les types de base que notre bibliothèque `Terminal`. Elles sont simples d'utilisation et assez fréquentes dans les ouvrages et exemples que vous trouverez ailleurs. Nous les présentons brièvement à titre d'information.

- `System.out.print` : affiche une valeur de base ou un message qui lui est passé en paramètre.
  ```
  System.out.print(5);   ---> affiche 5
  System.out.print(bonjour);   ---> affiche le contenu de bonjour
  System.out.print("bonjour");   ---> affiche bonjour
  ```
  Lorsque l'argument passé est un message ou *chaîne de caractères*, on peut lui joindre une autre valeur ou message en utilisant l'opérateur de *concaténation* `+`. **Attention** si les opérandes de + sont exclusivement numériques, c'est leur addition qui est affichée !

```
System.out.print("bonjour " + 5 );    ---> affiche:  bonjour 5
System.out.print(5 + 2);    ---> affiche 7
```

– `System.out.println` : Même comportément qu'avant, mais avec passage à la ligne en fin d'affichage.

2.6 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 :

Listing 2.4 – (lien vers le code brut)

```
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 exécution 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.

2.6.1 `if` **sans** `else`

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

2.6.2 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.

2.7 Itération

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`

Listing 2.5 – (lien vers le code brut)

```
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 instructions d'*itérations*.

2.7.1 La boucle for

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 :

Listing 2.6 – (lien vers le code brut)

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

RÃ©pÃ¨te `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.

Jouons un peu

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

Listing 2.7 – (lien vers le code brut)

```
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.

Listing 2.8 – (lien vers le code brut)

```
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.

Listing 2.9 – (lien vers le code brut)

```
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.

Listing 2.10 – (lien vers le code brut)

```
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.

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` :

Listing 2.11 – (lien vers le code brut)

```
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.

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.

Listing 2.12 – (lien vers le code brut)

```
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.

Listing 2.13 – (lien vers le code brut)

```
public class Rectangle3 {
  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 :

Listing 2.14 – (lien vers le code brut)

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

Pour en savoir plus

La syntaxe de la boucle `for` en Java est beaucoup plus permissive que ce qui à été exposé ici. Nous nous sommes contentés de décrire son utilisation usuelle. Nous reviendrons plus en détail sur les boucles dans un chapitre ultérieur.

2.7.2 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.

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 :

Listing 2.15 – (lien vers le code brut)

```
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.

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.

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`.

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 :

Listing 2.16 – (lien vers le code brut)

```
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 );
    }
}
```

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 :

Listing 2.17 – (lien vers le code brut)

```
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 :

Listing 2.18 – (lien vers le code brut)

```
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.

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.

MCours.com