Chapitre 7

Sous-programmes

7.1 Fonction : notion mathématique

Voici la définition mathématique classique du mot fonction :
Définition Soient A et B deux ensembles. Une fonction f définie sur A (ou de domaine A, ou d'espace de départ A, ou de domaine de définition A) à valeurs dans B (ou de codomaine B, ou d'espace d'arrivée B, ou d'espace image B) est une correspondance qui, à tout élément x de A, fait correspondre un élément et un seul, noté f(x), de B. Cet élément f(x) est appelé résultat de l'application de f à l'élément x (parfois image de x par f).

Remarque : Il ne faut pas confondre la fonction f, qui est un élément de l'ensemble des fonctions de A dans B (en général noté A -> B), et le résultat de l'application de f à un argument x, qui est un élément de B. Dans certains cours de mathématiques, lorsque l'on ne s'intéresse pas aux fonctions en tant que telles mais seulement aux résultats de leurs applications, on parle parfois de la fonction f(x).

La notion de fonction pose plusieurs questions. La première est celle de la *calculabilité*, c'est à dire la possibilité de calculer la valeur f(x) pour une valeur x de A donnée. On peut définir certaines fonctions sans pour autant avoir de moyen de réaliser le calcul correspondant.

Par exemple, la fonction qui à un numéro de département associe le nombre de personnes présentes actuellement dans ce département. Cette fonction a un sens parfaitement compréhensible, et ce nombre de personnes existe. Simplement, il n'y a aucun moyen réaliste pour calculer ce nombre. Même l'INSEE, l'Institut National des Statistiques ne peut faire qu'un calcul approximatif.

L'informaticien s'intéresse presque exclusivement à des fonctions calculables, et parmi celles-ci, plus spécialement à celles qu'un ordinateur peut calculer dans des conditions acceptables, c'est à dire en un temps limité et avec des moyens en rapport avec les enjeux du calcul. Par exemple, si l'on fait un programme de prévision météorologique pour les cinq jours qui viennent, si on a une fonction qui calcule très précisément ces prévisions en fonction des relevés de stations météo, mais qu'il faut deux mois pour calculer cette fonction, elle n'a aucun intérêt pratique.

Une autre questions qui se pose est celle du langage que l'on emploie pour définir une fonction. Nous aborderons cette question d'abord du point de vue des mathématiques qui fournissent une base théorique pour les constructions offertes par les langages informatiques pour définir des fonctions.

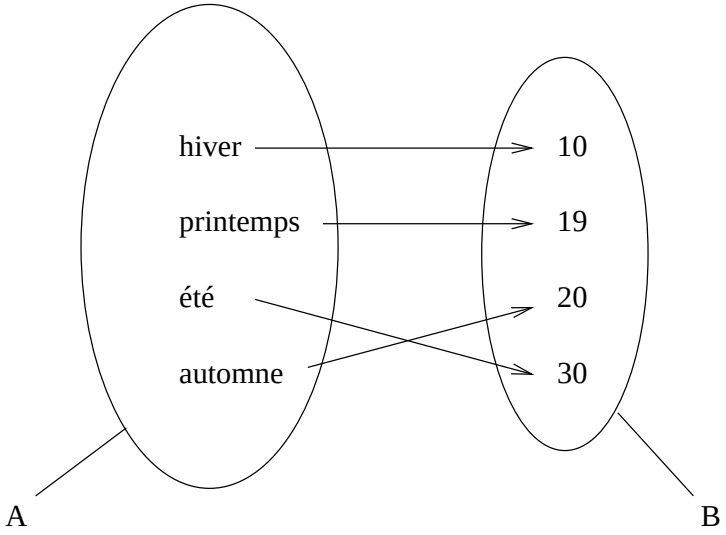
En mathématiques, il existe plusieurs méthodes pour définir une fonction. Voici celles qui sont le plus couramment employées.

7.1.1 Construire le graphe

Le graphe d'une fonction f de A dans B est l'ensemble des couples `(x,y)` où `x` est un élément de A, `y` est un élément de B et `y=f(x)`. Si le domaine A est fini, on peut indiquer explicitement quel est l'élément de B qui correspond à un élément donné de A. Dans ce cas, on peut définir la fonction par son graphe de manière effective.

Soit l'ensemble A = {hiver, printemps, été, automne}. On peut définir une fonction `max_temp` par le graphe suivant : `{(hiver,10),(printemps,19),(été,30),(automne,20)}`.

Un tel graphe peut aussi se représenter graphiquement, de la façon suivante :



graphe de f

Calcul du résultat de l'application

Calculer le résultat de l'application d'une fonction définie par un graphe à un argument donné `v` consiste simplement à chercher le couple `(v,y)` dans le graphe et en extraire la valeur `y` qui est la valeur de `f(v)`. Par définition de la notion de fonction, il existe un seul couple commençant par `v` dans le graphe de la fonction `f`.

7.1.2 Donner une expression

La fonction peut être définie par une expression dont le calcul donne la valeur de la fonction en tout point de son domaine de définition. Pour cela, il faut choisir un nom, disons x, pour désigner un élément quelconque du domaine. Ce nom est appelé variable en mathématiques.

Par exemple on peut définir la fonction `f(x)=3*x+2` Dans cette fonction, `x` est la variable. C'est un nom que l'on donne pour désigner un élément quelconque de l'ensemble A. L'expression est `3*x2+`. Cette expression contient la variable, des constantes (2 et 3) et des opérations (+ et *). Ces opérations sont elles-memes des fonctions, définies avant `f`.

Le nom de la variable n'a pas d'importance. Par exemple, les deux définitions suivantes définissent une seule et même fonction : `f(x)=x+3` et `f(y)=y+3`.

Calcul du résultat de l'application

Pour calculer la valeur d'une fonction `f` définie par une expression pour une valeur donnée de l'ensemble de définition A, il faut remplacer dans l'expression la variable par cette valeur. Cela donne

une expression dans laquelle il n'y a plus d'inconnue. On peut réaliser le calcul pour obtenir une valeur de l'ensemble B.

Par exemple, pour calculer la valeur de `f` définie par `f(x)=3*x+2` pour la valeur 5, on remplace `x` par 5 dans l'expression `3*x+2`. Cela donne l'expression `3*5+2` dans laquelle il n'y a pas d'inconnue et qu'on peut calculer pour obtenir le résultat `17`. On en conclut que +f(5)=17+.

Par rapport à la définition au moyen d'un graphe, la définition de fonction par une expression a l'avantage de permettre de définir des fonctions dont le graphe est infini. Par exemple, la fonction `f(x)=3*x+2` est définie pour tout l'ensemble des entiers relatifs, qui est un ensemble infini. Il y a donc une infinité de couples dans le graphe de cette fonction.

L'ordre des calculs n'est pas important, c'est à dire que le résultat obtenu à la fin est le même quel que soit l'ordre de réalisation du calcul. Il existe d'ailleurs un certains nombres de lois définissant l'équivalence d'expressions, permettant de réaliser les calculs plus simplement (factorisation, associativité, commutativité).

7.1.3 Utiliser une construction par cas

Une fonction peut aussi ne pas être définie de la même façon suivant les valeurs de la variable. On utilise différentes expressions pour différents sous-ensemble de l'ensemble de définition A. On parle de fonction définie par morceau.

Voici quelques fonctions définies à l'aide de constructions par cas, écrites de différentes manières.

1. abs(x) = si x $\leq$ 0 alors x sinon (- x)
2. `par_morceaux` (x) = x + 1 si x $\leq$ 1
   x + 4 si x > 1 et x $\leq$ 100
   x + 2 si x > 100
3. continue (x) = sin(x) / x si x $\neq$ 0
   1 sinon

Pour qu'une telle définition soit valide, il faut que les différents cas soient mutuellement exclusifs, c'est à dire que pour une valeur donnée, il n'y ait qu'une définition.

Calcul du résultat de l'application

Pour calculer le résultat de l'application d'une fonction définie par morceau pour une valeur `v` donnée, il faut d'abord déterminer quel cas s'applique à cette valeur puis effectuer le calcul de l'expression donnée pour ce cas.

7.1.4 Utiliser la récursion

Les moyens déjà vus ne sont pas suffisants pour décrire par exemple la fonction factorielle de N dans N. La suite des valeurs de factorielle est souvent décrite comme suit :

```
0! = 1
n! = 1 * 2 * 3 * ... * (n-1) * n
```

L'écriture ci-dessus, même si elle est évocatrice, n'est en rien effective. Il est impossible d'écrire un algorithme contenant des points de suspension ! Pour définir une fonction calculant effectivement la factorielle d'un entier, il faut autoriser l'emploi du nom de la fonction en cours de définition dans l'expression qui la définit. Une telle définition sera dite récursive. Nous avons déjà rencontré des définitions récursives dans les descriptions de syntaxes. Voici une définition effective de la fonction factorielle : fact (n) = si n=0 ou n=1 alors 1 sinon n * fact (n-1)

Suivons le calcul de fact(3) en utilisant le symbole $>>$ pour abréger la phrase “se simplifie en”. fact(3) $>>$ `3 * fact(2)` $>>$ `3 * 2 * fact(1)` $>>$ `3 * 2 * 1` $>>$ 6 Le calcul a pu être mené à bien parce que le calcul de fact(1) est fait directement, sans appel de la fonction fact.

Pour être correcte, toute définition récursive de fonction, disons f, doit utiliser au moins une construction par cas avec au moins un cas dans lequel l'expression ne comporte pas la fonction f, de façon à permettre l'arrêt des calculs. Ce cas est appelé cas de base de la récursion.

Un autre exemple bien classique de fonction récursive est la fonction de Fibonnacci, définie par :

fib (n) = 1 si n =0 ou n=1
fib(n -1) + fib(n-2) sinon

Un autre exemple tout aussi célèbre est celui de la fonction d'Ackerman définie sur N$\times$ N par :

Ack(m, p) = p+1 si m=0
Ack (m-1,1) si p = 0
Ack(m-1, Ack(m,p-1)) sinon

Le lecteur est invité à calculer les valeurs Ack(m,p), pour les premières valeurs de m et p.

7.2 Fonction dans un programme

Il arrive que l'on fasse des calculs dans un programme. On peut vouloir exprimer ces calculs sous forme de fonction pour deux raisons :

1. éviter les répétitions. Si un même calcul apparaît à de multiples reprises dans un programme, en définissant une fonction, on n'écrit ce calcul qu'une fois, lorsque l'on décrit la fonction. Ensuite, chaque calcul consiste à utiliser cette fonction.
2. rendre le programme plus clair, plus lisible, en donnant un nom au calcul.

En java, il existe quelques fonctions prédéfinies appelées opérateurs. Ce sont les fonctions les plus courantes utilisées par chaque type de donnée. Pour les types numériques, ce sont les quatre opérations arithmétiques usuelles, pour le type boolean, les connecteurs logiques, et chaque type primitif possède ainsi quelques fonctions.

Il est possible d'écrire des fonctions dans un programme en utilisant la construction Java qui s'appelle *méthode*. Voyons un exemple simple : la fonction qui calcule la valeur absolue d'un nombre.

Si l'on cherche à caractériser mathématiquement cette fonction, on dira que c'est une fonction dont le domaine de définition est l'ensemble des entiers relatifs et les valeurs appartiennent à l'ensemble des entiers naturels (une valeur absolue est toujours positive ou nulle). Puis, on donnera une définition par cas :
$abs(x) = x$ si $x \geq 0$
$abs(x) = -x$ si $x < 0$

Voyons maintenant comment on peut décrire la fonction en Java. Nous avons déjà vu au chapitre 5 un programme qui calcule la valeur absolue, sans utiliser de fonction. Nous le rappelons ici, avant de voir le programme avec fonction.

Listing 7.1 – (lien vers le code brut)

```
1 public class ValAbs {
2     public static void main (String args[]) {
3         int x, abs;
4         Terminal.ecrireString("Donnez un entier : ");
5         x = Terminal.lireInt();
```

```
6           if (x > 0) {
7                abs = x;
8           } else if (x < 0) {
9                 abs = -x;
10          } else {
11                abs = 0;
12          }
13          Terminal.ecrireStringln("La valeur absolue est " + abs);
14      }
15  }
```

Avec une fonction, le programme peut s'écrire comme suit.

Listing 7.2 – (lien vers le code brut)

```
1   public class ValAbsFunc {
2       static int valeurAbsolue(int n){
3           int res;
4           if (n > 0) {
5                 res = n;
6           } else if (n < 0) {
7                 res = -n;
8           } else {
9                 res = 0;
10          }
11          return res;
12      }
13      public static void main (String args[]) {
14          int  x, abs;
15          Terminal.ecrireString("Donnez un entier: ");
16          x = Terminal.lireInt();
17          Terminal.ecrireStringln("La valeur absolue est " + valeurAbsolue(x));
18      }
19  }
```

Dans ce programme, il y a d'abord la définition d'une méthode appelée `valeurAbsolue`, puis utilisation de cette méthode dans le corps du programme (méthode `main`).

Voyons de plus près la définition. Elle comprend une ligne d'entête suivie d'un bloc (rappel : un bloc est une séquence d'instructions entre accolades). La ligne d'entête,
`static int valeurAbsolue(int x)`, comprend plusieurs informations :

- le mot-clé `static`, nécessaire, qu'on n'expliquera pas dans ce chapitre. Son rôle sera détaillé plus tard dans le cours.
- le type `int` : c'est le type du **résultat** du calcul.
- le nom `valeurAbsolue` : c'est le nom de la méthode, qui est au libre choix du programmeur, comme un nom de variable.
- entre parenthèses, deux informations : le type (`int`) et le nom `n` du paramètre de la fonction. Il s'agit de la valeur dont on cherche la valeur absolue.

Le bloc qui suit la ligne d'entête est un bloc d'instructions tout à fait classique, sauf qu'il utilise une nouvelle instruction appelée `return`. Cette instruction a pour effet de terminer l'exécution de la méthode. Elle est suivie du résultat renvoyé par la méthode, c'est à dire le résultat de la fonction. Dans notre exemple, on a calculé la valeur absolue de `n` et mis le résultat dans une variable locale `res`. Cette variable est locale au bloc.

A l'intérieur du bloc de définition d'une fonction, on a le droit d'utiliser le nom du paramètre défini dans l'entête (`n` apparaît plusieurs fois dans la méthode).

En ce qui concerne l'utilisation de la méthode, la terminologie propose deux variantes : on peut parler **d'appel** de la méthode ou **d'envoi de message**. On voit l'utilisation de la méthode `valeurAbsolue` dans la ligne :

```
Terminal.ecrireStringln("La valeur absolue est " + valeurAbsolue(x));
```

L'appel de méthode proprement dit est : `valeurAbsolue(x)`, c'est à dire une expression composée du nom de la fonction suivie de la valeur de son paramètre entre parenthèses. Il s'agit de calculer la valeur absolue du contenu de la variable `x`.

7.3 Notion de paramètre

Dans une fonction, on veut exprimer un calcul qui dépend d'une ou plusieurs valeurs susceptibles de varier dans un domaine. Ces valeurs, en mathématiques sont appelées les variables de la fonction. On évitera d'utiliser cette terminologie puisqu'en programmation, on utilise le mot variable pour autre chose. On parle de *paramètres* de la fonction.

Une fonction est un calcul dans lequel il y a des inconnues. Tant que ces inconnues restent inconnues, on ne peut pas effectuer le calcul et connaître son résultat.

Une fonction est une moulinette qui prend une ou plusieurs choses en entrée et ressort une purée en sortie. Une purée, pas deux. Par exemple, supposons qu'on mette des carottes et des patates dans la moulinette. Il en ressort une purée carotte-patate et non deux purées, une de carotte et une de patates.

Les inconnues, les carottes et les patates, ce sont les paramètres de la fonction. On donne un nom à ces paramètres.

Par exemple, quand on décrit : `f(x)=3*x+2`, `x` est un nom qu'on donne à une inconnue. Tant qu'on ne sait pas ce que vaut `x`, le calcul reste impossible. Le nom qu'on donne est arbitraire. Si on écrit : `f(y)=3*y+2`, c'est toujours la même fonction.

En Java, en plus du nom, il faut donner un type aux paramètres, ce qui permet de vérifier que le calcul est possible. Par exemple, pour la fonction `f`, `x` peut être du type `int`.

L'application, l'exécution, l'appel de la fonction pourra se faire en donnant une valeur à `x` et cette valeur devra être du bon type.

Il faut également donner le type du résultat, le type de la purée produite. En touillant de l'int avec les touillettes multiplication et addition, on obtient un int. Le type doit être cohérent avec la valeur calculée.

Dans le corps de la fonction, c'est à dire dans le bloc qui suit l'entête, on peut utiliser le nom des paramètres pour dénoter une valeur inconnue au moment où l'on écrit la fonction. Cette valeur sera connue au moment de l'utilisation de la fonction, au moment de l'appel.

Nous sommes habitués à manipuler des fonctions mathématiques, c'est à dire des fonctions numériques ayant des paramètres et résultats entiers ou réels. En informatique, on peut utiliser des fonctions pour tous les types possibles.

Par exemple, on peut écrire des fonctions intéressantes utilisant des caractères. Nous illustrons cela avec une fonction qui calcule si un caractère est une lettre en majuscule ou non. Pour comprendre comment il fonctionne, il faut savoir que Java utilise un codage appelé *Unicode* où toutes les lettres majuscules (sauf celles qui ont des accents, mais généralement, on n'utilise pas les majuscules accentuées) sont contiguës. Cela signifie que, dans l'ordre du type char, entre deux majuscules, il n'y a que des majuscules.

Listing 7.3 – (lien vers le code brut)

```
1  public class TestMajuscule{
2      static boolean estMajuscule(char c){
3          return (c >= 'A') && (c <='Z');
4      }
5      public static void main (String args[]) {
6          char x;
7          Terminal.ecrireString("Donnez un caractere: ");
8          x = Terminal.lireChar();
9          if (estMajuscule(x)){
10             Terminal.ecrireStringln("C'est une majuscule");
11         }else{
12             Terminal.ecrireStringln("Ce n'est pas une majuscule");
13         }
14     }
15 }
```

Peut-être certains d'entre vous auraient préféré écrire la fonction de la façon suivante :

Listing 7.4 – (pas de lien)

```
1      static boolean estMajuscule(char c){
2          boolean res;
3          if ((c >= 'A') && (c <='Z')){
4              res = true;
5          }else{
6              res = false;
7          }
8          return res;
9      }
```

C'est strictement équivalent, simplement moins efficace et moins élégant. En effet, quand la condition `(C>='A') && (C<='Z')` vaut `true`, la fonction `estMajuscule` renvoie la valeur `true`, et quand la condition vaut `false`, la fonction renvoie la valeur `false`. Donc, elle renvoie toujours la valeur de la condition, le `if` ne sert à rien.

7.4 Résultat d'une fonction

Une fonction sert à calculer un résultat qui est une valeur du type qui apparaît le premier dans l'entête. Nous avons vu deux exemples, celui de la valeur absolue qui renvoie un résultat de type int, le test de majuscule qui renvoie un résultat booléen (vrai ou faux, le caractère est une majuscule).

Dans ces deux exemples, nous avons calculé cette valeur dans une variable locale appelée `res`. Puis, sur la dernière ligne de la méthode, nous avons renvoyé cette valeur au moyen de l'instruction `return`.

Il est possible de renvoyer la valeur calculée dès qu'on la connaît, sans passer par le stockage dans une variable locale.

Listing 7.5 – (lien vers le code brut)

```
1      static int valeurAbsolue(int n){
2          if (n > 0) {
3              return n;
```

```
4           } else if (n < 0) {
5                   return -n;
6           } else {
7                   return 0;
8           }
9       }
```

La seule contrainte qui existe est que dans tous les cas, la fonction doit se terminer par un return. Par exemple, si l'on écrit :

Listing 7.6 – (lien vers le code brut)

```
1       static int valeurAbsolue(int n){
2           int res;
3           if (n > 0) {
4                   res = n;
5           } else if (n < 0) {
6                   return -n;
7           } else {
8                   return 0;
9           }
10      }
```

Dans le cas où `n` est supérieur à 0, il n'y a pas d'instruction `return`. C'est une erreur, le compilateur s'en aperçoit et donne un message :

```
> javac ValAbsFunc3.java
ValAbsFunc3.java:11: missing return statement
    }
    ^
1 error
```

7.5 Fonction à plusieurs paramètres

Une fonction peut comporter plusieurs inconnues. Le calcul n'est possible que si on donne une valeur à toutes les inconnues.

Par exemple : `f(x,y)=2*x+3*x*y+y+1`. Dans les fonctions prédéfinies de Java, il y en a qui ont plusieurs paramètres :

- les fonctions arithmétiques `+`, `*`, `-` et `/` sont des fonctions qui prennent deux paramètres de type `int` (ou un autre type numérique) et renvoie un résultat de type `int`.
- les fonctions de comparaison `<`, `>`, `<=` (notation Java pour $\leq$), `>=` (notation Java pour $\geq$), `!=` (notation Java pour $\neq$), et `=` sont des fonctions qui prennent deux paramètres de même type et renvoient un résultat de type `boolean`.
- `&&` (*et* logique), `||` (*ou* logique) sont des fonctions qui prennent deux paramètres booléens et rendent un résultat booléen.

Ces fonctions prédéfinies ont la particularité de pouvoir être utilisées avec une notation spéciale, dite notation infixe, dans laquelle la fonction apparaît entre ses deux paramètres. Lorsqu'on écrit de nouvelles fonctions non prédéfinies, la notation est un peu différente, les paramètres sont donnés entre parenthèses à la définition de la fonction comme à l'appel.

Listing 7.7 – (lien vers le code brut)

```
1      static int f(int x, int y){
2          return 2*x+3*x*y+y+1;
3      }
```

et l'appel à cette fonction (c'est à dire l'utilisation de la fonction en donnant des valeurs aux inconnues `x` et `y`) s'écrit comme suit : `f(45,12)`.

Prenons un autre exemple. Nous venons de mentionner qu'il existe des fonctions prédéfinies pour les connecteurs logique `et` et `ou`. Il n'y en a pas pour le connecteur `implique` (le connecteur le plus pénible de la logique des propositions). Nous allons écrire la fonction `implique`.

Rappelons la table de vérité de ce connecteur :

p	q	p => q
true	true	true
true	false	false
false	true	true
false	false	true

Une première façon d'écrire la fonction consiste à coder directement la table de vérité avec des `if` imbriqués.

Listing 7.8 – (lien vers le code brut)

```
1  public class Implique1{
2      static boolean implique(boolean a, boolean b){
3          if (a){
4              if (b){
5                  return true;     // a et b vrais
6              }else{
7                  return false;    // a vrai, b faux
8              }
9          }else{
10             return true;     // a vrai, b vrai ou faux
11         }
12     }
13     public static void main (String args[]) {
14         Terminal.ecrireString("true => false vaut: ");
15         Terminal.ecrireBooleanln(implique(true,false));
16         Terminal.ecrireString("false => true vaut: ");
17         Terminal.ecrireBooleanln(implique(false,true));
18     }
19 }
```

Une autre façon de coder la fonction serait d'utiliser une formule qui nous assure que `a => b` est toujours équivalent à `(non a) ou b`. On peut utiliser le non (`!`) et le ou (`||`) qui existent en Java.

Listing 7.9 – (lien vers le code brut)

```
1  public class Implique2{
2      static boolean implique(boolean a, boolean b){
3          return (!a) || b;
4      }
5      public static void main (String args[]) {
6          Terminal.ecrireString("true => false vaut: ");
7          Terminal.ecrireBooleanln(implique(true,false));
```

```
8           Terminal.ecrireString("false => true vaut: ");
9           Terminal.ecrireBooleanln(implique(false, true));
10      }
11  }
```

Comme le montre le programme, pour utiliser la fonction `implique`, on donne son nom suivi de deux expressions de type boolean entre parenthèses, séparées par des virgules : `implique(true,false)`. Le résultat est une valeur de type bolean, c'est à dire `true` ou `false`, si bien que l'appel de fonction peut être utilisé comme condition d'un `if`.

D'autres appels possibles :

- `implique(5<3*2, 'C' = 'd')`
- `implique(5<3*2, implique(True,13/=12))`
- `implique(x,y)` si `x` et `y` sont des variables de type boolean.
- `implique(x,5<y*2)` si `x` est une variable de type boolean et `y` une variable de type `int`.

7.6 Appel de fonction

Il est très important de comprendre que les paramètres donnés à une fonction à l'appel sont des valeurs. Les paramètres sont un moyen de communiquer entre le programme et le sous-programme. La fonction `f` est une moulinette. L'appel de fonction `f(carotte,patate)` désigne ce qui sort de la moulinette (à savoir la purée). On peut utiliser `f(carotte,patate)` à tous les endroits du programme où on a besoin de purée. Il faut bien distinguer la fonction (une moulinette) de ce que son usage produit (une purée).

Lors de l'exécution de l'appel de fonction, le passage de valeurs est automatique : les valeurs données entre parenthèses sont stockées dans des cases mémoire allouées aux paramètres.

Détaillons un exemple d'appel de fonction pas à pas.

Listing 7.10 – (lien vers le code brut)

```
1  public class Divisible{
2      static boolean estDivisiblePar(int a, int b){
3          if (a % b == 0){
4              return true;
5          }else{
6              return false;
7          }
8      }
9      public static void main(String[] args){
10         int x = 35;
11         int y = 2;
12         if (estDivisiblePar(x, 3 * y - 1)){
13             Terminal.ecrireStringln("divisible");
14         }
15     }
16 }
```

Il y a dans ce programme un seul appel de fonction, à savoir `estDivisiblePar(x, 3 * y - 1))` et c'est cet appel dont nous allons détailler l'exécution pas à pas.

1. la première étape est le calcul de la valeur des paramètres.

- calcul de la valeur de `x` (valeur du premier paramètre). Au moment de l'appel de fonction, la variable `x` contient la valeur 35.
- calcul de la valeur de `3 * y - 1` (valeur du deuxième paramètre). Au moment de l'appel, la variable `y` contient la valeur 2. Cette valeur est utilisée à la place de `y` dans l'expression qui devient `3 * 2 - 1`. Cette expression vaut `5`.

2. de la mémoire est allouée pour les deux paramètres `a` et `b` et les valeurs 35 et 5 sont stockées respectivement dans ces deux cases mémoires.
3. le corps de la fonction, c'est à dire le bloc des lignes 4 à 9, est exécuté dans l'ordre.
   - ligne 4 : la condition `a % b == 0` est évaluée. On va chercher dans la mémoire la valeur de `a` (35) et celle de `b` (5) et on calcule `35 % 5`, c'est à dire le reste de la division entière de 35 par 5. Ce reste vaut 0 qui est égal à 0. Donc la condition vaut la valeur `true`.
   - la condition étant vraie, c'est le premier bloc entre `if` et `else` qui est exécuté, c'est à dire la ligne 5. Il y a là une instruction `return` suivi du résultat calculé par la fonction. Ce résultat est `true`. L'effet de cette instruction est de terminer le calcul de la fonction.
4. la mémoire donnée au paramètres `a` et `b` est libérée. Les deux paramètres cessent d'exister (jusqu'au prochain appel).
5. la valeur de l'appel `estDivisiblePar(x, 3 * y - 1))` est `true`. Cette valeur est utilisée dans l'instruction `if` de la ligne 13 : la condition est vraie, le message de la ligne 14 est affiché.

Deux erreurs à ne pas faire :

- penser qu'on ne peut donner comme paramètres de la fonctions que deux variables de nom `a` et `b`. Si on déclare deux variables `a` et `b` elles n'ont rien à voir avec les deux paramètres `a` et `b`. En particulier, des espaces mémoires différents seront attribués, même si le nom est identique.
- penser qu'il faut donner une valeur à `a` et `b` par un `Terminal.lireInt()` ou une affectation. Ce n'est ni nécessaire ni possible. L'attribution d'une valeur à `a` et `b` se fait automatiquement lors de l'appel, comme montré dans l'exemple.

7.7 Fonction partielle

Certaines fonctions ne sont pas définies sur tout le type de leurs paramètres.

Par exemple, la division entière est une fonction qui prend deux entiers et rend un entier, mais elle n'est pas définie pour tous les couples d'entiers que l'on peut lui donner. Elle n'est pas définie pour un diviseur nul. 17/0 n'est pas défini.

On dit que la division est une fonction partielle.

Que se passe-t-il en Java quand on utilise la division hors de son domaine de définition ? Vous pouvez le voir en compilant et en exécutant le programme suivant :

Listing 7.11 – (lien vers le code brut)

```
1 public class DivZero{
2     public static void main(String[] args){
3         int x = 0;
4         Terminal.ecrireIntln(17/x);
5     }
6 }
```

A la compilation, il n'y a pas de problème : le programme est bien typé ; il est considéré comme correct (si on avait écrit directement `17/0`, le compilateur aurait produit une erreur, mais en utilisant la variable `x`, on le trompe facilement).

A l'exécution, il se produit une erreur, avec un message :

```
> java DivZero
java.lang.ArithmeticException: / by zero
        at DivZero.main(DivZero.java:5)
```

En termes Java, on dit qu'une exception a été levée. Le programme s'arrête.

Lorsqu'on écrit ses propres fonctions, on veut parfois obtenir le même comportement. On veut que le programme s'arrête lorsqu'un appel à la fonction est fait avec des valeurs pour les paramètres qui sont hors de son domaine de définition.

Prenons un exemple simple : la fonction factorielle. Cette fonction est définie seulement sur les nombres positifs. Pas sur tous les entiers. Nous allons définir une nouvelle erreur et la lever lorsque le paramètre donné est négatif.

Listing 7.12 – (lien vers le code brut)

```
1  public class Factorielle{
2      static int factorielle(int n){
3          int res = 1;
4          if (n<0){
5              throw new PasDefini();
6          }
7          for (int i = 1; i<=n; i++){
8              res = res * i;
9          }
10         return res;
11     }
12     public static void main(String[] argv){
13         int x;
14         Terminal.ecrireString("Entrez un nombre (petit): ");
15         x = Terminal.lireInt();
16         Terminal.ecrireIntln(factorielle(x));
17     }
18 }
19 class PasDefini extends Error{
20 }
```

Essayez ce programme et voyez ce qui se produit quand on entre au clavier un nombre négatif.

Définir la nouvelle erreur consiste à écrire une nouvelle classe avec la clause `extends Error`. Nous verrons beaucoup plus tard ce que cela signifie exactement. Pour l'instant, il suffit d'utiliser cela comme une incantation dans laquelle vous changerez à volonté le nom de la classe en remplaçant `PasDefini` par autre chose.

Dans le programme, le déclenchement de l'erreur se fait au moyen de l'instruction `throw new PasDefini()`. Là encore, dans un premier temps, nous utiliserons cela comme une incantation, avant de voir ce que cela signifie.

On verra plus tard dans l'année qu'il est possible de lever une exception à la place d'une erreur, ce qui permet au programme de corriger l'erreur plutôt que de s'arrêter purement et simplement. Pour l'instant, nous allons seulement créer des erreurs, qui, contrairement aux exceptions, ne peuvent pas être récupérées.

7.8 Méthodes faisant des entrées-sorties

Nous avons présenté longuement comment on peut utiliser une méthode pour coder une fonction en Java. Il existe des méthodes qui ne correspondent pas à des fonctions parce qu'elles ne calculent pas un résultat. Ce sont des sous-programmes qui font des effets, notamment des sorties à l'écran.

Les méthodes de `Terminal` telles que `ecrireString` et `ecrireInt` sont de bons exemples de telles méthodes. Elles produisent une affichage mais ne renvoient pas de valeur. On n'utilise jamais ces méthodes à gauche d'une affectation ou comme condition d'un `if` ou comme portion d'un calcul.

Ces méthodes correspondent à ce qu'on appelle *procédure* dans les langages impératifs ou procéduraux.

Voyons un exemple de méthode qui n'est pas une fonction.

Listing 7.13 – (lien vers le code brut)

```
1  public class AfficheTable{
2      static void afficheTable(int[] t){
3          Terminal.ecrireChar('+');
4          for (int i=0; i<t.length; i++){
5              Terminal.ecrireString("---+");
6          }
7          Terminal.sautDeLigne();
8          Terminal.ecrireChar('|');
9          for (int i=0; i<t.length; i++){
10             Terminal.ecrireString(" " + t[i] + " |");
11         }
12         Terminal.sautDeLigne();
13         Terminal.ecrireChar('+');
14         for (int i=0; i<t.length; i++){
15             Terminal.ecrireString("---+");
16         }
17         Terminal.sautDeLigne();
18     }
19     public static void main(String[] args){
20         int[] ex = {1,5,8,9,7};
21         afficheTable(ex);
22     }
23 }
```

La méthode `afficheTable` affiche de façon pseudo-graphique les tableaux contenant des entiers à un chiffre. Dans la définition de la méthode, il y a deux différences avec les fonctions :

- dans l'entête, au lieu de déclarer le type de la valeur calculée, il y a le mot-clé `void` qui signifie précisément que la méthode ne calcule pas de valeur. En anglais, *void* signifie *vide*.
- dans le corps de la méthode, il n'y a pas nécessairement d'instruction `return`.

On voit dans l'exemple que l'appel à la méthode (`afficheTable(ex);`) est utilisé seul sur une ligne, alors que cela n'aurait pas de sens d'appeler une fonction seule sur une ligne.

Il existe enfin des méthodes hybrides qui font à la fois des entrées-sorties et qui calculent un résultat. C'est le cas par exemple de `Terminal.lireInt()`. Cette méthode renvoie une valeur de type `int`. On l'emploie souvent à droite d'une affectation (par exemple `x = Terminal.lireInt()`). Mais ce n'est pas à proprement parler une fonction parce que le résultat ne dépend pas que de la valeur des paramètres. Elle dépend aussi des entrées au clavier.