Chapitre 5

Procédures et fonctions

5.1 Introduction

Considérons le programme suivant, dont le but est d'inverser les éléments d'un tableau :

```
public class InversionTableau1 {
    public static void main(String args[]) {
        int t[]= {8, 2, 1, 23};
        Terminal.ecrireStringln("Tableau␣avant␣inversion");
        for (int i= 0; i < t.length; i++) {
            Terminal.ecrireInt(t[i]);
            if (i != t.length -1)
                Terminal.ecrireChar('␣');
        }
        Terminal.sautDeLigne();
        // Inversion du tableau
        int a= 0; int b= t.length -1;
        while (a < b) {
            int tmp= t[a];
            t[a]= t[b];
            t[b]= tmp;
            a++;
            b--;
        }
        Terminal.ecrireStringln("Tableau␣après␣inversion");
        for (int i= 0; i < t.length; i++) {
            Terminal.ecrireInt(t[i]);
            if (i != t.length -1)
                Terminal.ecrireChar('␣');
        }
        Terminal.sautDeLigne();
    }
}
```

Le code affiche le tableau, l'inverse, et enfin, affiche le tableau inversé. On remarque que le code qui affiche le tableau est identique les deux fois. C'est gênant, pour plusieurs raisons :

- ça prend de la place ;
- mais surtout, si l'on s'aperçoit que l'algorithme d'affichage est faux, ou qu'on veut améliorer la présentation, il faut aller le modifier à tous les endroits où on l'a écrit.

– D'autre part, le programme est long et peu lisible. On pourrait l'améliorer un peu à l'aide de commentaires, mais il restera difficile de s'y retrouver dans la masse du code.

Au lieu de cela, nous allons créer une *procédure* que nous appellerons `afficherTableau`, qui contiendra le code d'affichage. De même nous isolerons le code qui inverse le tableau dans une procédure appelée `inverserTableau`, et le code deviendra :

```
public class InversionTableau2 {

    public static void main(String args[]) {
      int t[]= {8, 2, 1, 23};
      Terminal.ecrireStringln("Tableau␣avant␣inversion");
      afficherTableau(t);
      inverserTableau(t);
      Terminal.ecrireStringln("Tableau␣après␣inversion");
      afficherTableau(t);
    }

    public static void afficherTableau(int t[]) {
        for (int i= 0; i < t.length; i++) {
            Terminal.ecrireInt(t[i]);
            if (i != t.length -1)
                Terminal.ecrireChar('␣');
        }
        Terminal.sautDeLigne();
    }

    public static void inverserTableau(int t[]) {
      int a= 0; int b= t.length -1;
      while (a < b) {
        int tmp= t[a];
        t[a]= t[b];
        t[b]= tmp;
        a++;
        b--;
      }
    }
}
```

Nous détaillerons les éléments de ce programme plus tard. Pour l'instant, on constate que l'utilisation de procédures permet :

– d'éviter des copier-coller ;
– d'écrire du code plus lisible.
– de réutiliser des bouts de code d'un programme à l'autre ;

Ce que fait le `main` est compréhensible immédiatement : on affiche le tableau, on l'inverse, puis on affiche le tableau inversé. On peut lire et comprendre cette partie du code en faisant abstraction des détails, c'est à dire ici de la manière dont on affiche ou dont on inverse le tableau.

Par ailleurs, l'affichage et l'inversion sont isolés dans les procédures, et chacune d'entre elle est compréhensible séparément, ce qui améliore la lisibilité du code.

Enfin, nous verrons que les procédures peuvent être réutilisées d'un programme à l'autre, ce qui évite de réinventer la roue. On peut ainsi se construire une *bibliothèque* avec des procédures fréquemment utilisées, pour n'avoir pas besoin de les réécrire.

5.2 Procédures

Une procédure est un bout de code qui a un *nom* et qui peut être appelé depuis le reste du programme.

En java, une procédure est forcément déclarée dans une classe. Dans sa forme la plus simple, elle s'exprime ainsi :

```
public static void nomDeLaProcédure() {
   Corps de la procédure...
}
```

Supposons par exemple que nous voulions faire une jolie présentation, entrecoupée de lignes de 60 astérisques. Nous pourrions l'écrire :

```
public class Lignes1 {
    public static void main(String args[]) {
        dessinerLigne();
        Terminal.ecrireStringln("Un texte bien encadré");
        dessinerLigne();
        Terminal.ecrireStringln("Après la ligne");
        Terminal.sautDeLigne();
        Terminal.sautDeLigne();
        dessinerLigne();
    }

    public static void dessinerLigne() {
        for (int i= 0; i < 60; i++) {
            Terminal.ecrireChar('*');
        }
        Terminal.sautDeLigne();
    }
}
```

La procédure `dessinerLigne` est définie dans la classe `Lignes1`. *Par convention,* en java, le nom d'une procédure est supposé commencer par une minuscule.

La procédure est *appelée* plusieurs fois dans le `main` (qui, soit dit en passant, est aussi une procédure). L'appel d'une procédure a la forme :

nomDeLaProcédure() ;

Les parenthèses () sont obligatoires. Nous verrons leur raison d'être très bientôt.

Lors de l'appel, le code de la procédure est exécuté, puis on revient dans le `main`, à l'endroit de l'appel, et le code du main se poursuit.

Une procédure peut en appeler d'autres. On pourrait par exemple créer la procédure « `dessinerDeuxLignes()` » :

```
public class Ligne1 {
   ... même code que précédemment...

   public static void dessinerDeuxLignes() {
      dessinerLigne();
      dessinerLigne();
   }
}
```

Cette procédure appelle la procédure `dessinerLigne` déjà écrite.

5.2.1 Variables locales

On a déjà vu que les variables déclarées à l'intérieur d'un bloc étaient *locales* à ce bloc, c'est-à-dire qu'elles n'existent que durant l'exécution du bloc. Cela reste vrai pour les variables déclarées dans les procédures.

De plus, comme les procédures sont un moyen pour découper le code en unités plus ou moins indépendantes, que l'on souhaite pouvoir modifier et comprendre indépendamment les unes des autres, la question des variables locales y est d'une grande importance.

Une *variable locale* n'existe que le temps de l'exécution de la procédure qui la déclare. **Si deux méthodes déclarent des variables de même noms, ces variables sont complètement distinctes.**

Ainsi, dans le programme suivant :

```
public class VarLocales {

    public static void dessinerLigne() {
        int a= 0;
        while (a < 60) {
            Terminal.ecrireChar('*');
            a= a+1;
        }
        Terminal.sautDeLigne();
    }

    public static void main(String args[]) {
        int a= 20;
        dessinerLigne();
        Terminal.ecrireIntln(a);
    }
}
```

Nous avons deux variables nommées ń a ż : une qui est déclarée dans le `main,` l'autre qui est déclarée dans `dessinerLigne.` Quand le main appelle la procédure `dessinerLigne`, une seconde variable, nommée ń a ż est définie. Elle est complètement différente de la première, et occupe un autre espace mémoire.

Le début de l'exécution du programme donne donc la trace suivante :

ligne	a (de main)	a (de dessinerLigne)
13	20	(n'existe pas)
14	20	(n'existe pas)
4	20	0
5	20	0
6	20	0
7	20	1
... exécution de la boucle ...		
9	20	60
15	20	(n'existe pas)

La variable `a` définie dans le main existe jusqu'à la fin de l'exécution de celui-ci. La variable `a` de `dessinerLigne` est créée au début de l'exécution de `dessinerLigne,` et disparaît à la fin de l'exécution de cette procédure.

On remarquera que
- dans `dessinerLigne,` on ne ń voit ż pas les variables définies dans le `main.` Celles-ci occupent une place en mémoire, mais on ne peut pas y accéder (ni *a fortiori* les modifier) depuis `dessinerLigne.`
- si on appelait plusieurs fois dessinerLigne à partir du `main,` à chaque appel correspondrait une nouvelle variable `a,` initialisée à 0. C'est normal, puisque les variables locales des appels précédents auront été détruites.

5.2.2 Arguments d'une procédure

Le système que nous venons de décrire est un peu trop rigide. On désire très souvent fournir des informations à la procédure pour qu'elle adapte son comportement en conséquence. Dans notre cas, on pourrait par exemple vouloir choisir la longueur de la ligne au moment de l'appel (alors qu'il est actuellement fixé une fois pour toutes à 60).

Pour cela, on va utiliser des *arguments.* L'idée est de pouvoir appeler la procédure en écrivant

```
dessinerLigne(30);
```

pour dessiner une ligne de 30 '*'.

ń 30 ż est ici un *argument,* c'est à dire une information passée à la procédure lors de son appel.

C'est le même mécanisme qui est mis en ¡uvre quand nous appelons `Terminal.ecrireStringln("bonjour tout le monde")`, `Terminal.ecrireIntln(x*2)` ou `Terminal.ecrireDouble(z)`. Dans tous ces cas, `"bonjour tout le monde"`, `x*2` et `z` sont des valeurs passées en paramètre aux méthodes `ecrireStringln, ecrireIntln` et `ecrireDouble.`

Déclaration d'une procédure avec arguments

Une procédure peut donc avoir une liste d'arguments (éventuellement vide). Cette liste, ainsi que le nom de la procédure, constitue ce que l'on appelle la *signature* de la procédure.

On déclare les arguments entre parenthèses, juste après le nom de la procédure. On déclare un argument comme une variable, en le faisant précéder de son type. Ainsi, la nouvelle procédure `dessinerLigne` pourrait s'écrire :

```
public class ProcedureAvecArguments {
  public static void dessinerLigne(int longueurLigne) {
    for (int i= 0; i < longueurLigne; i++) {
      Terminal.ecrireChar('*');
    }
    Terminal.sautDeLigne();
  }

  ... suite de la classe...
```

La méthode pourra être appelée depuis une autre procédure de la même classe (par exemple depuis main). Pour appeler une procédure qui prend des arguments, on écrit le nom de la procédure, suivi des valeurs des arguments, entre parenthèses.

```
  ....
  public static void main(String args[]) {
    int k= 50;
```

```
    dessinerLigne(k);
    dessinerLigne(k/2);
    dessinerLigne(k/4);
  }
}
```

Une procédure peut aussi avoir plusieurs arguments. Notre méthode `dessinerLigne` actuelle utilise forcément le caractère '*'. Il serait intéressant de pouvoir dessiner des lignes de '-' ou de '+', par exemple. C'est possible en créant une méthode avec plusieurs arguments :

```
public class ProcedureAvecArguments2 {
  public static void dessinerLigne(int longueurLigne, char symbole) {
    for (int i= 0; i < longueurLigne; i++) {
      Terminal.ecrireChar(symbole);
    }
    Terminal.sautDeLigne();
  }

  ... suite de la classe...
```

Dans cet exemple, nous définissons une méthode `dessinerLigne` qui prend deux arguments : le premier est un entier, `longueurLigne,` et le second est un caractère, `symbole.` Les arguments sont ensuite utilisés dans le corps de la méthode.

Plus précisément, la liste des arguments d'une procédure est composée d'une suite de déclarations de variables, séparées par des virgules. Attention, on doit répéter le type devant chaque argument. Si j'ai par exemple deux arguments entiers, je devrais écrire :

```
  public static void dessinerRectangle(int largeur, int hauteur) {
    ....
  }
```

Quand on appelle une procédure en java, il est obligatoire de fournir des valeurs pour tous ses arguments. L'ordre des arguments est important, ainsi que leur type. Pour appeler la procédure `dessinerRectangle` de l'exemple précédent, il faut passer deux entiers. Le premier donnera sa valeur à l'argument ń `largeur` ż, le second à l'argument ń `hauteur` ż.

De même, pour appeler la dernière version de notre procédure `dessinerLigne,` on devra passer d'abord un entier (la longueur de la ligne), puis un caractère (le symbole à utiliser). Donc, pour dessiner une ligne de 50 '-', j'écrirai :

```
  dessinerLigne(50, '-');
```

Localité des paramètres

Le passage de paramètre en `Java` se fait par *valeur.* Le paramètre est une variable créée lors de l'appel de la procédure, qui *reçoit une* ***copie*** de la valeur qu'on lui passe.

Une modification d'un paramètre ne modifie pas *a priori* la variable correspondante du programme principal.

Ainsi, si nous considérons le programme suivant :

```
public class Localite {
    public static void augmenterMalEcrit(int val) {
        val= val + 1;
```

```
    }


    public static void main(String args[]) {
        int k= 10;
        augmenterMalEcrit(k);
        Terminal.ecrireIntln(k);
    }
}
```

Lors de l'appel de `augmenterMalEcrit,` ligne 9, on crée une variable `val,` locale à `augmenterMalEcrit.` la *valeur* de `k` va être copié dans `val.`

Nous avons donc deux variables distinctes : `k` et `val`, qui contiennent toutes les deux la valeur 10. La ligne 3 augmente la valeur de `val,` qui vaut 11, mais absolument pas la valeur de `k,` qui reste inchangé. Quand `augmenterMalEcrit` se termine, on revient dans `main. val` disparaît et on affiche la valeur de `k,` soit 10.

Supposons maintentant que nous définissions une variable `val` dans le `main` :

```
public class Localite2 {
    public static void augmenterMalEcrit(int val) {
        val= val + 1;
        Terminal.ecrireStringln("La␣valeur␣du␣val␣de␣augmenterMalEcrit␣:␣"+ val);
    }


    public static void main(String args[]) {
        int k= 10;
        int val= 30;
        augmenterMalEcrit(k);
        Terminal.ecrireIntln(k);
        Terminal.ecrireIntln(val);
    }
}
```

Cette variable est complètement distincte du `val` de `augmenterMalEcrit.` Le déroulement du programme sera le même que précédemment, et affichera :

```
La valeur du val de augmenterMalEcrit : 11
10
30
```

Surcharge

Nous avons écrit plusieurs variantes de `dessinerLigne.` Les dernières sont très souples, mais un peu plus complexes d'emploi. Dans certains cas, le programmeur veut juste dessiner une ligne. Il ne se soucie ni de sa longueur exacte, ni du symbole utilisé.

Il serait donc intéressant de disposer de plusieurs variantes de `dessinerLigne.` Dans un premier temps, on pourrait imaginer de leur donner tout simplement des noms différents :

- `dessinerLigne`;
- `dessinerLigneDeLongueurDonnée`
- `dessinerLigneDeLongueurDonnéeAvecSymbole`

Mais trouver des noms parlants pour nos procédures est déjà difficile. On n'a pas forcément envie de les multiplier par le nombre de variantes désirées. D'autre part, il est plus simple pour le programmeur qui *utilise* les procédures de ne pas s'encombrer la mémoire avec plusieurs noms. [1] Pour résoudre ce problème, `java` utilise la *surcharge.* Il s'agit tout simplement de la possibilité de déclarer plusieurs méthodes avec le même nom. Elles sont alors distinguées par leur liste d'arguments. Attention, les listes d'arguments sont uniquement basées sur le *type* des arguments, pas sur leur *noms.*

Nous pourrions donc écrire notre code de dessin de ligne :

```
public class LigneMania {

    /**
     *Dessine une ligne d'une certaine longueur, en utilisant le caractère symbole.
     */
    public static void dessinerLigne(int longueur, char symbole) {
        for (int i= 0; i < longueur; i++) {
            Terminal.ecrireChar(symbole);
        }
        Terminal.sautDeLigne();
    }

    /**
     *Dessine une ligne de '*' d'une certaine longueur.
     */

    public static void dessinerLigne(int longueur) {
        dessinerLigne(longueur, '*');
    }

    public static void dessinerLigne() {
        dessinerLigne(60, '*');
    }


    public static void main(String args[]) {
        dessinerLigne();
        Terminal.ecrireStringln("Un texte bien encadré");
        dessinerLigne(20);
        Terminal.ecrireStringln("Après la ligne");
        Terminal.sautDeLigne();
        Terminal.sautDeLigne();
        dessinerLigne(100,'-');
    }
}
```

On remarquera que dans cet exemple, nous avons *en plus* éviter de répéter le code de dessin de la ligne : les procédures « simplifiées » appellent la procédure la plus générale.

1. Pour être tout à fait honnête, le choix n'est pas si facile que cela. La surcharge peut parfois rendre le code plus difficile à lire.

5.2.3 Passage de tableaux en argument

Pour un tableau passé en paramètre, java passe la *référence* du tableau (son adresse, en fait). En conséquence, le tableau d'origine et celui vu par la procédure occupent le même espace mémoire.

Considérons le programme suivant :

```
public class TestTab1 {

    public static void main(String args[]) {
      int tab[]= {8, 2, 1, 23};
      afficherTableau(tab);
    }

    public static void afficherTableau(int t[]) {
        for (int i= 0; i < t.length; i++) {
            Terminal.ecrireInt(t[i]);
            if (i != t.length -1)
                Terminal.ecrireChar(' ');
        }
        Terminal.sautDeLigne();
    }
}
```

Lors de l'appel de la procédure `afficherTableau,` on copie dans `t` la valeur de `tab.` Mais `tab` contient *l'adresse* du tableau | 8 | 2 | 1 | 23 |. `t` va donc désigner le même espace mémoire que `tab.` et donc, le même tableau, et non pas une *copie* du tableau.

Durant l'exécution de `afficherTableau`, nous avons donc en mémoire :

– le tableau | 8 | 2 | 1 | 23 |, à une certaine adresse (par exemple 0xA0300).
– `tab,` qui contient l'adresse du tableau (donc 0xA0300)
– `t,` qui contient aussi l'adresse du tableau (donc 0xA0300)

Une première conséquence de cela est que le passage d'un tableau en paramètre n'entraîne pas la copie de celui-ci (ce qui serait prohibitif pour un gros tableau).

Une seconde conséquence est que si la procédure modifie le contenu du tableau, comme le paramètre et la variable d'origine désignent le *même* tableau en mémoire, le tableau d'origine est aussi modifié.

On peut donc considérer qu'une procédure peut modifier le *contenu* d'un tableau passé en paramètre. C'est ce qui se passe lors de l'appel d'`inverserTableau` dans notre premier exemple :

```
public class InversionTableau2 {

    public static void main(String args[]) {
      int tab[]= {8, 2, 1, 23};
      Terminal.ecrireStringln("Tableau avant inversion");
      afficherTableau(tab);
      inverserTableau(tab);
      Terminal.ecrireStringln("Tableau après inversion");
      afficherTableau(tab);
    }

    public static void afficherTableau(int t[]) {
        for (int i= 0; i < t.length; i++) {
            Terminal.ecrireInt(t[i]);
```

```
            if (i != t.length -1)
                Terminal.ecrireChar('␣');
        }
        Terminal.sautDeLigne();
    }

    public static void inverserTableau(int t[]) {
      int a= 0; int b= t.length -1;
      while (a < b) {
        int tmp= t[a];
        t[a]= t[b];
        t[b]= tmp;
        a++;
        b--;
      }
    }
}
```

5.2.4 Appel d'une procédure définie dans une autre classe

Jusqu'ici, quand nous avons voulu réutiliser une procédure dans un nouveau programme, nous l'avons tout bonnement recopiée. Mais, en pratique, on préfère ne pas dupliquer le code, fût-ce au moyen d'un copier/coller. L'idéal est donc de regrouper

On tente donc de créer des *bibliothèques* de procédures. Vous en utilisez déjà une : la classe `Terminal`. On pourrait vouloir utiliser notre `LigneMania` de la même manière. C'est en fait très facile. Dans l'état actuel de vos connaissances en java, pour utiliser les méthodes d'une classe *A* depuis une classe `B` :

- il faut que le code des deux classes soit dans le même dossier ;
- on peut appeler les méthodes de A en préfixant simplement le nom de la méthode par « `A.` »

Exemple :

```
public class TestLignes {
  public static void main(String args[]) {
    LigneMania.dessinerLigne(100,'-');
  }
}
```

5.2.5 Récapitulation

Une procédure est un morceaux de code situé dans une classe. Elle a un *nom* et une liste de paramètres, éventuellement vide.

En théorie, une procédure *agit* sur son environnement :

- soit elle modifie l'environement du programme (en écrivant par exemple sur l'écran, sur l'imprimante, en sauvant des données sur le disque, en produisant un son) ;
- soit elle modifie la valeur d'un de ses paramètres ; en java, ça n'est véritablement possible que lorsque l'un d'entre eux est une *référence*, donc, pour l'intant, dans le cas des tableaux.

5.3 Fonctions

Une *fonction* ressemble à une procédure, mais sa pour tâche est de renvoyer une valeur, qui sera utilisée par la suite. L'exemple le plus immédiat, ce sont les fonctions mathématiques. La fonction `cos`, de la classe `Math`, calcule ainsi le cosinus de son argument. On l'utilise de la manière suivante :

```
double x= 3;
double y= Math.cos(x/2);
```

Vous utilisez par ailleurs plusieurs fonctions : `Terminal.lireInt()`, `Terminal.lireString()`, etc... Elles ne prennent pas d'argument, mais renvoient l'entier, le texte... saisis par l'utilisateur.

La grande différence entre procédures et fonctions est donc que ces dernières renvoient quelque chose. D'un point de vue pratique, dans une fonction :

– on met le type de la valeur renvoyée à la place du « `void` » des procédures :

public static *TYPE_RETOURNÉ NOM_FONCTION* (*ARGUMENTS*)

– on renvoie la valeur du résultat à l'aide du mot-clef « `return` ».

Commençons par un exemple : la fonction valeur absolue.

```
1  public class F1 {
2      public static double valeurAbsolue(double x) {
3          double resultat;
4          if (x < 0)
5              resultat= -x;
6          else
7              resultat= x;
8          return resultat ;
9      }
10
11     public static void main(String args[]) {
12         double v= Terminal.lireDouble();
13         Terminal.ecrireDoubleln(valeurAbsolue(v));
14     }
15
16 }
```

La fonction `valeurAbsolue` déclare dans son en-tête qu'elle retourne une valeur de type `double`. La valeur est effectivement renvoyée ligne 8, à l'aide de la commande `return`.

La forme de cette commande est

`return` ***EXPRESSION*** ;

où *EXPRESSION* est du même type que la fonction.L'exécution de cette commande a deux effets :

1. elle fixe la valeur de retour ;
2. elle termine l'exécution de la fonction.

On aurait pu aussi écrire :

```
public static double valeurAbsolue(double x) {
    if (x < 0)
        return -x;
```

```
        else
            return x;
    }
```

mais on considère généralement qu'il est plus élégant de ne placer qu'un seul `return` dans la fonction (on sait alors exactement où la fonction se termine, et cela simplifie sa relecture).

Attention, une fonction doit toujours se terminer par un `return`[2] Si java détecte (parfois à tort) qu'il est possible que ça ne soit pas le cas, il refuse de compiler. La fonction suivante, par exemple :

```
    public static double valeurAbsolue(double x) {
        if (x < 0)
            return -x;
    }
```

produira *à la compilation* le message d'erreur :

```
Err.java:5: missing return statement
    }
    ^
```

5.3.1 Fonctions qui retournent un tableau

Une fonction peut très bien renvoyer un tableau. Il suffit pour cela qu'elle le déclare comme type de retour. La fonction suivante renvoie par exemple un tableau, dont la taille et le contenu des cases sont donnés en paramètre.

```
public static double[] creerTableau(int taille, double valeur) {
  double [] t= new double [taille]; // création du tableau
  for (int i= 0; i < taille; i++)
    t[i]= valeur;
  return t;
}
```

Remarquez en particulier la dernière ligne. On renvoie tout simplement `t` (sans crochet).

5.4 Fonctions partielles

Il arrive qu'une fonction ne soit pas définie pour toutes les valeurs possibles de ses paramètres. Considérons par exemple la fonction suivante :

```
/**
 *Renvoie la somme des trois premiers éléments d'un tableau
 */
public static double sommeDesTroisPremiers(double t[]) {
  if (t.length >= 3) {
    return t[0]+ t[1]+ t[2];
  }
}
```

2. Sauf dans le cas où elle lève une exception, voir *infra.*

Cette fonction n'est pas définie si la taille du tableau est inférieure à trois. En fait, telle qu'elle est écrite, elle ne compilera même pas, puisque, si le test est faux, elle ne retourne rien, et qu'une fonction doit toujours retourner quelque chose.

Java fournit un mécanisme pour résoudre le problème : les exceptions. Il s'agit en fait de messages d'erreur, qui interrompront le programme. On reformule quelque peu la règle précédente en :

> une fonction doit toujours, ou se terminer en renvoyant une valeur, ou lancer (ou *lever)* une exception.

Pour lever une exception, on écrira :

> **`throw new Error(`***MessageDErreur*`)`** ;

Notre fonction devra donc s'écrire :

```
/**
 *Renvoie la somme des trois premiers éléments d'un tableau
 */
public static double sommeDesTroisPremiers(double t[]) {
   if (t.length >= 3) {
      return t[0]+ t[1]+ t[2];
   } else
      throw new Error("appel de sommeDesTroisPremiers avec un tableau trop petit");
}
```

Le mécanisme que nous venons de décrire est très simplifié. Nous détaillerons par la suite le système des exceptions, en voyant en particulier comment il est possible de « rattraper » celles-ci.

Les exceptions sont aussi utilisables dans les procédures.

5.5 Varia

5.5.1 Notes de vocabulaire

Dans la littérature, vous constaterez peut-être des usages variables pour les termes de procédures et de fonctions. Vous lirez sans doute aussi le terme « méthode ». Le tout est souvent une question d'école, ou de point de vue. La terminologie que nous avons adoptée pour notre exposé est, en gros, celle du langage Pascal. Elle nous permet de distinguer les deux concepts de procédures et de fonctions.

Pour des raisons historiques, en langage C, les procédures sont vues comme « des fonctions qui ne renvoient rien ». Comme `java` a hérité la terminologie de C, on utilise donc aussi le mot `fonction` pour désigner indifféremment les procédures et les "vraies" fonctions. D'où, entre autres, le `void` devant les noms des procédures, à la place du type du résultat.

Le terme « méthode », que vous rencontrerez très probablement, provient des langages orientés objets. Il désigne une procédure ou une fonction définies dans une classe. Ce qui est le cas de toutes les procédures et fonctions en java.

Pour résumer :

méthode procédure ou fonction définie dans une classe ;

procédure sous-programme nommé, ne renvoyant pas de résultat, et modifiant généralement son environnement ;

fonction théoriquement, sous-programme renvoyant une valeur. Parfois utilisé (en C ou java) pour désigner tout sous programme.

5.5.2 Notion d'effet de bord

La tâche principale d'une fonction est de renvoyer une valeur. Il arrive cependant qu'une fonction modifie aussi son environnement (l'affichage, le contenu d'un fichier, ses paramètres quand il s'agit de tableaux, etc...)

C'est un procédé à utiliser avec précaution, dans la mesure où il est souhaitable qu'une procédure ou une fonction ait une tâche unique et bien définie.

5.6 Erreurs fréquentes et remarques finales

– Un sous-programme ne demande jamais à *l'utilisateur* les valeurs de ses paramètres... celles-ci sont connues au moment de l'appel. Le code suivant est illogique :

```
public static double carre(double x) {
   x= Terminal.lireDouble(); // NON ! x est déjà connu et on le redemande sans raison
   return x*x;
}
```

– Une fonction n'affiche pas son résultat ; elle le renvoie ;
– Les paramètres formels sont des variables locales et sont donc indépendants des variables utilisées dans la méthode appelante.