# Chapitre 7

# Méthodes

Une méthode que l'on peut appeler également un sous-programme, une fonction ou une procédure, est un petit programme qui réalise une tâche déterminée. Elle prend en entrée des paramètres et peut renvoyer un résultat ou ne pas renvoyer de résultat.

## 7.1 Rappel de ce que l'on a déjà vu

Nous utilisons des méthodes prédéfinies depuis le début de ce cours. Nous connaissons donc ce qu'est un appel de méthode.

Les exemples de méthodes que nous utilisons couramment sont :

— les méthodes de sortie à l'écran : `System.out.println`
— les méthodes d'entrée au clavier : `Terminal.lireInt`
— les méthodes mathématiques : `Math.min`, `Math.random`

### 7.1.1 Signature d'une méthode

Pour appeler une de ces méthodes, il faut écrire son nom suivi d'une liste de paramètres entre parenthèses. Pour utiliser une méthode, il faut connaitre le nombre et le type de ses paramètres. Il faut également savoir si elle renvoie un résultat et le cas échéant, le type de son résultat. On appelle **signature** d'une méthode la liste des types de paramètres et du résultat éventuel de la méthode.

Les quatre méthodes mentionnées ci-dessus ont les signatures suivantes, notées graphiquement.

On peut également décrire les signatures textuellement. On utilise alors le petit mot-clé void pour dire qu'une méthode ne renvoie pas de résultat.

System.out.println : String → void
Terminal.lireInt : → int
Math.min : int, int → int
Math.random : → double

### 7.1.2 Instruction ou expression

Les méthodes ne s'utilisent pas de la même façon selon qu'elles renvoient un résultat ou pas. Si elles ne renvoient pas de résultat, on les considère comme des instructions. Les appels à ces méthodes apparaissent généralement seuls sur une ligne de programme. On les appelle parfois des **procédures**.

```
System.out.println("Bonjour les amis");
```

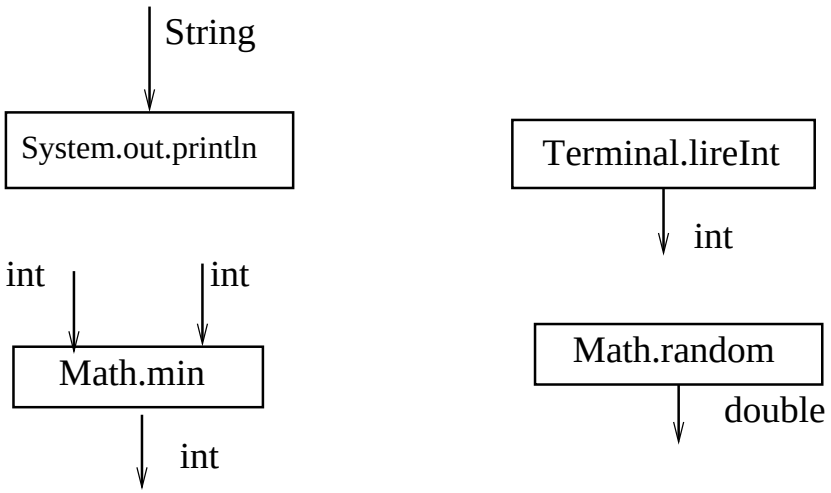


FIGURE 7.1 – Signatures de quelques méthodes

Les méthodes qui renvoient un résultat ne sont généralement pas seules sur une ligne : elles apparaissent au milieu d'un code. Par exemple, un appel de méthode peut apparaitre à droite d'une affectation ou en tant que paramètre, entre les parenthèses d'un autre appel de méthode. Ces méthodes sont parfois appelées des **fonctions**.

```
x = Math.min(x,12);
System.out.println(Math.min(x,12));
```

### 7.1.3 Paramètres dans un appel de méthode

Lorsqu'on appelle une méthode, il faut lui fournir entre parenthèses le bon nombre de paramètres, avec les bons types et dans le bon ordre. Ce qui apparaît entre les parenthèses sont des expressions. Au moment de l'exécution, chacune de ces expression est calculée et la valeur qui est le résultat du calcul est transmis à la méthode appelée.

Il y a quatre sortes d'expressions : valeurs littérales, variables, expression avec opérateur, appel d'une méthode qui renvoie un résultat. Chacune de ces sortes peut être utilisée dans un appel de méthode.

```
int x = 12;
System.out.println(45); // valeur littérale
System.out.println(x); // variable
System.out.println(1+2); // expression avec opérateur
System.out.println(Math.min(x,5)); // appel de méthode
```

## 7.2 Pourquoi et comment écrire une méthode

### 7.2.1 Pourquoi écrire des méthodes ?

Jusqu'à présent, nous avons écrit des programmes qui ne comportaient qu'une méthode : la méthode main. Il est souvent intéressant d'écrire plusieurs méthodes et ce pour plusieurs raisons :

- Pour diviser un gros programme en petits éléments plus facile à écrire, à comprendre et à tester qu'un grand main, avec beaucoup de boucles et de if imbriqués les uns dans les autres.
- Pour éviter de réécrire les mêmes suites d'instructions plusieurs fois dans le programme. Par exemple, s'il y a plusieurs tableaux qu'on souhaite afficher dans un programme, il y aura autant de boucles for à écrire que de tableaux à afficher et leur corps seront quasiment identiques, seul le nom du tableau étant différent à chaque fois. On pourra avantageusement remplacer les boucles par des appels à une méthode unique d'affichage de tableaux d'un certain type.

Diviser un code en plusieurs morceaux, chacun correspondant à une méthode, ne doit pas être fait au hasard : chaque méthode doit avoir une cohérence interne, c'est-à-dire réaliser un travail bien précis que l'on peut expliquer.

### 7.2.2 Étude de la méthode main

Pour savoir comment écrire une méthode, on peut commencer par regarder la méthode que nous savons déjà écrire, à savoir la méthode `main`. Prenons un exemple de programme.

```
1  public class Conversion {
2     public static void main (String[] args) {
3        double euros;
4        double dollars;
5        System.out.println("Somme␣en␣euros?␣");
6        euros = Terminal.lireDouble();
7        dollars = euros *1.118;
8        System.out.println("La␣somme␣en␣dollars:␣");
9        System.out.println(dollars);
10    }
11 }
```

La méthode commence par un entête qui figure sur la ligne numéro 2, suivi d'une liste d'instructions entre accolades (accolade ouvrante en fin de ligne 2, fermante en ligne 10, instructions des lignes 3 à 9).

Voyons les différentes composantes de l'entête :

- `public static` : deux mot-clés de Java que nous n'expliquerons pas pour le moment et qui seront toujours écrits en début d'entête pour toutes les méthodes.
- `void` : spécifie que la méthode ne renvoie pas de résultat. Si au contraire, la méthode renvoyait un résultat, ce serait le type de ce résultat qui apparaîtrait à cette position de l'entête.
- `main` : le nom de la méthode. Dans le cas de la méthode `main`, ce nom est imposé. C'est cette méthode qui est toujours exécutée lorsqu'on demande à exécuter un programme. Pour les autres méthodes, leur nom sera au libre choix du programmeur, comme le nom des variables du programme.
- `(String[] args)` : c'est la liste des paramètres de la méthode. Ici, il n'y a qu'un paramètre. Ce paramètre a pour nom `args` et pour type `String[]`. Ce paramètre est comme une sorte de variable, utilisable dans le programme.

On peut se demander à quoi sert le paramètre de la méthode `main`. Il sert à transmettre des données entrées lors de la demande d'exécution du programme. Par exemple, si l'on demande l'exécution d'un programme depuis une ligne de commande dans un terminal, on commence cette ligne par la commande `java`, ensuite on met le nom de la classe contenant la méthode `main` et ensuite on peut ajouter des données séparées par des espaces. Ces données sont transmises au programme dans le tableau `args`.

Voyons quelques exemples d'appels d'un programme qui utilise ce paramètre.

```
public class ArgMain{
   public static void main(String[] args){
      System.out.println("Taille de args: " + args.length);
      for (int i=0; i<args.length; i=i+1){
         System.out.println("Contenu de args[" + i + "]: " + args[i]);
      }
   }
}
```

Et deux exécutions de ce programme :

```
> java ArgMain
Taille de args: 0
> java ArgMain un deux 47 789
Taille de args: 4
Contenu de args[0]: un
Contenu de args[1]: deux
Contenu de args[2]: 47
Contenu de args[3]: 789
```

## 7.3 Méthodes sans résultat (void)

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

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

```
 public static void nomDeLaMéthode() {
   Corps de la méthode...
 }
```

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();
      System.out.println("Un texte bien encadré");
      dessinerLigne();
      System.out.println("Après la ligne");
      System.out.println();
```

```
      System.out.println();
      dessinerLigne();
   }

   public static void dessinerLigne() {
      for (int i= 0; i < 60; i++) {
         System.out.print('*');
      }
      System.out.println();
   }
}
```

Il y a deux méthodes dans la classe : `main` et `dessinerLigne`. L'ordre dans lequel sont écrites les méthodes dans la classe n'a aucune importance. Ici, il y d'abord le `main` puis `dessinerLigne`. Ce pourrait être l'inverse sans changer le fonctionnement du programme.

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

La méthode est *appelée* plusieurs fois dans le `main`. L'appel d'une méthode sans résultat a la forme :

**NomDeLaClasse.nomDeLaMéthode**() ;

Si l'appel de la méthode est dans la même classe que la méthode appelée, on peut ne pas mettre le nom de la classe qui est alors sous-entendu et l'appel devient :

**nomDeLaMéthode**() ;

Lors de l'exécution de l'appel, les instructions de la méthode sont exécutés, puis on revient dans le `main`, et les instructions venant après l'appel sont exécutées à leur tour.

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

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

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

Cette méthode appelle la méthode `dessinerLigne` déjà écrite.

### 7.3.1 Variables locales

Les variables déclarées dans une méthode sont locales à cette méthode : elles ne peuvent être utilisées que par les instructions de la méthode.

De plus, comme les méthodes 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 méthode 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 `a` : une qui est déclarée dans le `main`, l'autre qui est déclarée dans `dessinerLigne`. Quand le main appelle la méthode `dessinerLigne`, une seconde variable, nommée `a` 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 méthode.

On remarquera que

— dans `dessinerLigne`, on ne *voit* 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.

### 7.3.2 Paramètres d'une méthode

Le système que nous venons de décrire est un peu trop rigide. On désire très souvent fournir des informations à la méthode 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 *paramètres.* L'idée est de pouvoir appeler la méthode en écrivant

```
dessinerLigne(30);
```

pour dessiner une ligne de 30 '*'.

`30` est ici un *paramètre,* c'est-à-dire une information passée à la méthode 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 méthode avec paramètres

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

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

```
public class ProcedureAvecParamètres {
  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 méthode de la même classe (par exemple depuis main). Pour appeler une méthode qui prend des paramètres, on écrit le nom de la méthode, suivi des valeurs des paramètres, entre parenthèses.

```
  ....
  public static void main(String[] args) {
    int k= 50;
    dessinerLigne(k);
    dessinerLigne(k/2);
    dessinerLigne(k/4);
  }
}
```

Une méthode peut aussi avoir plusieurs paramètres. 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 paramètres :

```
public class ProcedureAvecParamètres2 {
  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 paramètres : le premier est un entier, `longueurLigne,` et le second est un caractère, `symbole.` Les paramètres sont ensuite utilisés dans le corps de la méthode.

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

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

Quand on appelle une méthode en Java, il est obligatoire de fournir des valeurs pour tous ses paramètres. L'ordre des paramètres est important, ainsi que leur type. Pour appeler la méthode `dessinerRectangle` de l'exemple précédent, il faut passer deux entiers. Le premier donnera sa valeur au paramètre `largeur`, le second au paramètre `hauteur`.

De même, pour appeler la dernière version de notre méthode `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 sorte de variable créée lors de l'appel de la méthode, qui reçoit la valeur qui est calculée pour ce paramètre lors de l'appel.

Une modification d'un paramètre ne modifie pas 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
```

### 7.3.3 Appel d'une méthode définie dans une autre classe

Jusqu'ici, quand nous avons voulu réutiliser une méthode 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 méthodes. 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,'-');
  }
}
```

## 7.4 Méthodes calculant un résultat

Une méthode calculant un résultat ou *fonction* ressemble à une méthode void, mais sa 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 paramètre. 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 de paramètre, mais renvoient l'entier, le texte... saisis par l'utilisateur.

La grande différence entre méthodes sans et avec résultat 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 méthodes :

```
public static TYPE_RETOURNÉ NOM_FONCTION (PARAMÈTRES)
```

— on renvoie la valeur du résultat à l'aide de l'instruction `return`.

Commençons par un exemple : la méthode valeur absolue.

```
public class F1 {
   public static double valeurAbsolue(double x) {
      double resultat;
      if (x < 0)
         resultat= -x;
      else
         resultat= x;
      return resultat;
   }

   public static void main(String[] args) {
      double v= Terminal.lireDouble();
      Terminal.ecrireDoubleln(valeurAbsolue(v));
   }

}
```

La méthode `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 l'instruction `return`.

La forme de cette instruction est

`return` ***EXPRESSION*** ;

où *EXPRESSION* est du même type que la méthode. L'exécution du `return` a deux effets :

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

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 méthode (on sait alors exactement où la méthode se termine, et cela simplifie sa relecture).

Attention, une méthode doit toujours se terminer par un `return`[1] Si Java détecte (parfois à tort) qu'il est possible que ça ne soit pas le cas, il refuse de compiler. La méthode 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
    }
    ^
```

### 7.4.1 Méthodes qui retournent un tableau

Une méthode peut très bien renvoyer un tableau. Il suffit pour cela qu'elle le déclare comme type de retour. La méthode 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=i+1){
    t[i]= valeur;
  }
  return t;
}
```

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

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

## 7.5 Exécution d'un appel de méthode

Un appel de méthode est exécuté en plusieurs étapes faisant intervenir deux méthodes distinctes : la méthode **appelante** dans laquelle se situe l'appel ; la méthode **appelée** qui doit être exécutée.

1. calcul de la valeur des paramètres. Ces valeurs sont obtenues en exécutant des expressions situées dans la méthode appelante. Ce calcul donne une valeur pour chacun des paramètres.
2. allocation en mémoire d'un espace réservé à la méthode appelée, avec dedans les paramètres et leurs valeurs calculées à l'étape précédente.
3. exécution des instructions du corps de la méthode appelée. Si dans ce corps, des variables locales sont déclarées, elles sont ajoutées dans l'espace mémoire réservée à la méthode.
4. lorsque l'instruction à exécuter est un `return`, la valeur renvoyée est calculée.
5. l'exécution de la méthode est alors terminée. L'espace mémoire réservé est rendu libre, les variables locales et paramètres sont détruits. la valeur calculée est renvoyée à la méthode appelante.

Une fois la méthode appelée exécutée, la main repasse à la méthode appelante qui reprend l'exécution de ses propres instructions, généralement en utilisant la valeur renvoyée.

## 7.6 Varia

### 7.6.1 Notion d'effet de bord

La tâche principale d'une méthode est de renvoyer une valeur. Il arrive cependant qu'une méthode modifie aussi son environnement (l'affichage, le contenu d'un fichier, etc...).

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

## 7.7 Erreurs fréquentes

— 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 ici sans raison
  return x*x;
 }
```

— Une méthode n'affiche pas son résultat ; elle le renvoie ;

— Les paramètres formels sont locaux à la méthode et sont donc indépendants des variables utilisées dans la méthode appelante.