Chapitre IV : Fonctions et Méthodes

I. Fonctions et Méthodes

On appelle *fonction* un sous-programme qui permet d'effectuer un ensemble d'instruction pouvant être réutilisé. C'est un moyen de simplifier le code et de réduire le programme principal au minimum.

On appelle *méthode* toute fonction faisant partie d'une classe. Elle permet d'effectuer des traitements sur ou avec les données membres de la classe. Elle se définit donc dans la classe.

1. Déclaration et appel d'une méthode

Avant d'être utilisée, une méthode doit être définie afin d'être **appelée**. Définir une méthode revient à déclarer son **nom**, ses **arguments** et les **instructions** qu'elle contient. La syntaxe de déclaration de méthode est la suivante.

```
1. Type_De_Valeur_De_Retour Nom_De_La_Methode(type1 argument1,type2 argument2, ...)
2. {
3.     Liste d'instructions
4. }
```

1. Appel de méthode

Pour exécuter une méthode, il suffit de faire appel à elle en écrivant son nom, tout en respectant la casse, suivie de ses arguments, s'ils existent.

```
1. //Appel de méthode sans valeur de retour (void)
2. Nom_De_La_méthode (argument1, argument2, …, argumentN);
```

Ou

```
1. //Appel de méthode avec valeur de retour
2. type_de_donnees_de_retour variable_de_retour = Nom_De_La_méthode (argument1,
   argument2, …, argumentN);
```

Exemple

```
1. int[] tab = new int[10];
2. remplir(tab);                // Méthode sans valeur de retour
3. int mx = maximum(tab);       // Méthode avec valeur de retour
4. trier(tab);                  // Méthode sans valeur de retour
5. afficher(tab);               // Méthode sans valeur de retour
```

Remarques

- Lors d'appel d'une méthode avec arguments, il faut veiller à ce qu'ils soient dans le même ordre et avec les mêmes types.
- Un argument peut être :
    - une constante, ou une variable
    - une expression
    - une autre méthode retournant une valeur

2. Valeur de retour

La méthode peut renvoyer une valeur grâce au mot-clé **return**, qui met fin à l'exécution de la méthode. L'instruction return peut renfermer :

- une valeur ou une constante ou une variable
- une expression
- une autre méthode retournant une valeur
- la méthode elle-même et dans ce cas on aura une méthode récursive.

Lorsque l'instruction return est rencontrée, la méthode évalue la valeur qui la suit, puis la renvoie au programme appelant, c'est la classe à partir de laquelle la méthode a été appelée. Une méthode **peut contenir plusieurs instructions return.** En voici un exemple.

```
1. public int max(int x, int y, int z) {
2.  if (x >= y)
3.  { if (z >= x)  { return z; }
4.    else         { return x; }
5.  }
6.  else { if (z >= y)    { return z; }
7.         else           { return y; }
8.  } }
```

Remarques

- L'instruction return doit être la **dernière** instruction exécutée.
- Le programme doit être conçu et écrit de sorte qu'il exécute toujours une instruction return à la fin.
- Le type de valeur de retour doit correspondre à celui qui a été précisé dans la définition de la méthode.

3. Exercice d'application

Dans chacun des cas suivants, qu'affiche le code donné ?

a)

```
1. public double lire()
2. { System.out.print("Entrez un nombre: ");
```

```
3.    Scanner keyb = new Scanner(System.in);
4.    double n = keyb.nextDouble();
5.    if (n > 0) { return n; }
6.  }
```

b)

```
1.  public void afficheRacine(double a)
2.  {
3.    if (a < 0.0) { return; }
4.   System.out.println(Math.sqrt(a));
5.  }
```

c)

```
1.  private static Scanner clavier = new Scanner(System.in);
2.  public static void main(String[] args)
3.  {
4.   int val = saisieEntier();
5.   System.out.println(val);
6.  }
7.  public int saisieEntier()
8.  {
9.   int i;
10.  System.out.println("entrez un entier: ");
11.  i = clavier.nextInt();
12.  return i;
13. }
```

4. Passage d'arguments

En programmation, et de façon générale, on dispose de deux types de passage de paramètres ou d'arguments :

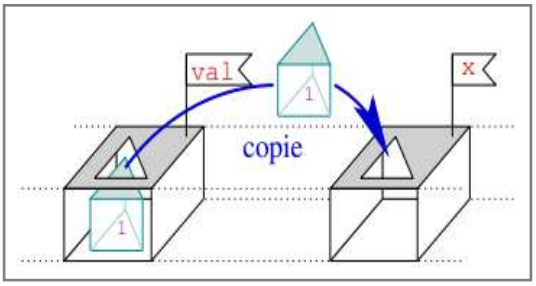
- Passage par valeur : si la méthode ne peut pas modifier la valeur de l'argument et elle travaille avec une **copie locale** de cet argument.
- Passage par adresse ou par référence si la méthode peut modifier la valeur de l'argument en question.

En Java, il n'existe qu'un seul type de passage d'arguments, c'est le **passage par valeur**. En fait, lors du passage par adresse, cas des types évolués, par exemple, la méthode reçoit une copie de la référence au paramètre, donc c'est un passage par valeur aussi.

a. Passage par valeur

```
public static void main(String[] args)
{ int val = 1;
  m(val);
  System.out.println(" val=" + val);
}
public void m(int x)
{  x = x + 1;

  System.out.print("x=" + x);
}
```



x=2 val=1

Au début de l'exécution de la méthode, les valeurs des paramètres sont recopiées dans l'espace mémoire de la méthode. Puis les traitements seront effectués sur ces copies !

Exemple

b. Passage par adresse ou référence : Type évolué

Au début de l'exécution de la méthode, l'adresse en mémoire du paramètre est passée à la méthode. Dans ce cas, la méthode peut modifier la valeur qui est enregistrée à cet endroit, et qui pourra être utilisée par le code appelant (main en général).

Exemple

```
1. public static void main(String[] args) {
2. int[] tab = {1};
3. m(tab);   // tab (référence) => copie de la référence dans x
4.           // PASSAGE PAR VALEUR AUSSI
5. System.out.println(" tab[0]= " + tab[0]);
6. }
7. public void m(int[] x) {
8. int [] t ={100};
9. x = t;    // Modification de la reference de l'objet,
10.          // la référence originale reste inchangée
11.System.out.println("x[0]= " + x[0]);
12.}
```

c. Exercice d'application

Dans chacun des cas suivants qu'affiche le code donné ?

a)

```
1. public void echanger(int a, int b) {
2. int temp = a;
3.  a = b;
4. b = temp;
5. }
6. public static void main(String[] args) {
7. int x = 10, y = 20;
8. echanger(x, y);
9. System.out.println(x + " ; " + y);
10. }
```

b)

```
1. public static void main(String[] args) {
2. int[] tab = {1};
3. m(tab);
4. System.out.println(" tab[0]= " + tab[0]);
5. }
6. static void m(int[] x) {
7. x[0] = 100;
8. System.out.print("x[0]= " + x[0]);
9. }
```

5. Surcharge de Méthodes : Overloading

Un des apports les plus intéressants du Java est la possibilité d'appeler plusieurs méthodes avec le même **nom**, à condition que leurs **arguments diffèrent** (en type et/ou en nombre). Ce principe est appelé **surcharge** de méthode. Il permet de donner le même nom à des méthodes comportant des paramètres différents et simplifie donc l'écriture de méthodes sémantiquement similaires sur des paramètres de type différent. On dit que les méthodes surchargées ont des ***signatures*** (nom et arguments) différentes.

Exemple

```
1. int somme( int p1, int p2){
2. return (p1 + p2);
3. }
4. float somme( float p1, float p2){
```

```
5. return (p1 + p2);
6. }
7. float somme( float p1, float p2, float p3){
8. return (p1 + p2 + p3);
9. }
10. int somme( float p1, int p2){
11. return ((int)(p1) + p2);
12. }
```

Remarques

- En Java, on **ne peut pas** avoir deux méthodes de **même nom** et de **mêmes paramètres** (nombre et types) mais avec **un type de retour est différent**. Par exemple, on ne peut pas avoir deux méthodes int f(int) et double f(int).
- Lors de la définition de méthodes surchargées, il est obligatoire de les définir de façon non ambiguë.

Exemple

```
1. public class Surcharge {
2.     void ajouter(double a, int b){
3.           System.out.println("1er ajouter appelée.");
4.     }
5.     void ajouter(long a, long b){
6.           System.out.println("2ème ajouter appelée.");
7.     }
8.     public static void main(String[] args) {
9.           ajouter(5, 5); // Quelle Méthode « ajouter » sera invoquée ??!
10.    }
11. }
```

⇨ `Erreur de compilation : The method ajouter(double, int) is ambiguous for the type …`