

Méthodes de classes en JAVA

Introduction

Le problème :

Lorsque l'on progresse dans la conception d'un algorithme ou d'un programme, ce dernier peut prendre une taille et une complexité croissante. En outre, des séquences d'instructions peuvent se répéter à plusieurs endroits. Un algorithme ou programme écrit d'un seul tenant devient difficile à comprendre et à maintenir.

La solution :

Une solution consiste à découper le programme en plusieurs parties plus petites. Chaque partie réalise une tâche particulière que lui sous-traite le programme principal. Ces parties sont appelées ***fonctions*** ou ***procédures***.

- Dans le langage Java, on les appelle des ***méthodes***.

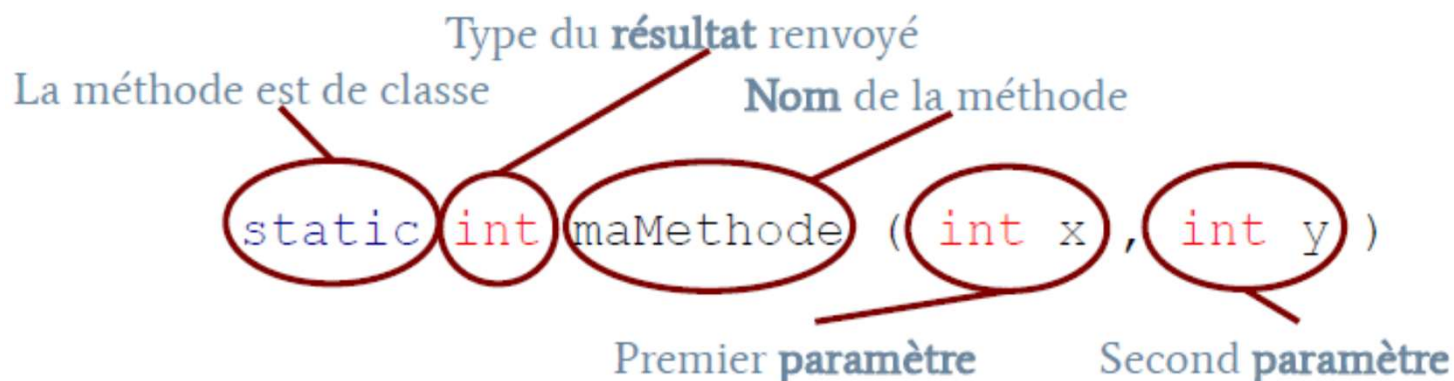
Qu'est-ce qu'une méthode ?

- Une méthode est un groupement d'instructions spécialisés dans la réalisation d'une tâche particulière ;
- Il permet de réutiliser du code ; par exemple si l'on souhaite afficher plusieurs tableaux, il suffit d'écrire une méthode qui permet de le faire pour n'importe quel tableau et de l'appeler au besoin.
- Une méthode peut prendre des arguments (paramètres) ou pas.
- Une méthode peut **renvoyer un résultat** (fonction) ou **pas** (procédure)

Déclaration d'une méthode de classe

En JAVA, une méthode est obligatoirement déclarée dans une classe.

- Une méthode de classe possède :
 - Un nom (C'est le nom de la macro-instruction)
 - Une liste de paramètres d'entrée sous la forme **type** symbol
 - Le type de résultat renvoyé (**void** qui signifie vide si la méthode ne retourne pas de résultat)



Définition d'une méthode de classe

- Une méthode de classe possède un corps délimité par { et }
 - Le corps contient une suite d'instructions
 - Termine avec **return** résultat ; (**return**; si pas de résultat)

Exemple : valeur absolue d'un réel

```
static double valAbs (double x) {  
    if (x >= 0) {  
        return x;  
    }  
    else { return -x; }  
}
```

Corps de la
méthode

Le return termine la méthode

Cas de la méthode main (point de départ d'un programme)

- La méthode main est une méthode spéciale qui ne retourne pas de résultat qui indique le point de départ d'un programme.

```
public static void main ( String [] args )
```

Diagram illustrating the components of the `main` method signature:

- `public`: Déclaration de visibilité
- `static`: Ne retourne pas de résultat
- `void`: Ne retourne pas de résultat
- `main`: Nom spécial pour indiquer Le point de départ du programme
- `(String [] args)`: `args` est un argument de la méthode Sous la forme d'un tableau de `String`

Appel d'une méthode (On parle aussi d'invocation d'une méthode)

- Pour utiliser une méthode, il suffit de l'invoquer (de l'appeler) :

```
nomMéthode (arg1, arg2, ....) //méthode ne retournant pas de résultat
```



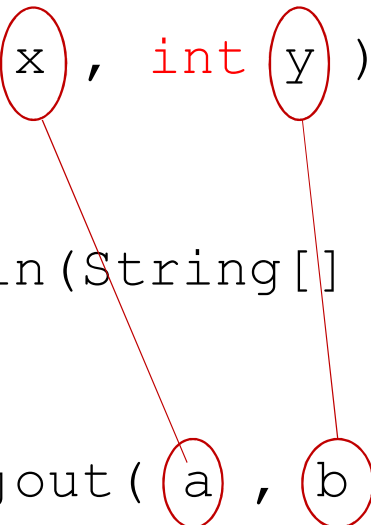
Liste de paramètres

```
type x = nomMéthode (arg1, arg2, ....) //le résultat est stocké dans  
la variable x
```

Attention : une méthode est toujours déclarée dans une classe mais peut être appelée par une autre méthode

Exemple

- Class Maclasse {
 static int ajout(int x, int y) {
 return x+y ;
 }
 public static void main(String[] args) {
 int a = 5;
 int b = 7;
 int resultat = ajout(a, b);
 System.out.println(a + " + " + b + " = " + resultat);
 }
}



The diagram illustrates the flow of arguments in the provided Java code. Two red circles are drawn around the variables 'a' and 'b' in the 'ajout' method call within the 'main' method. Two red lines originate from these circles: one line connects 'a' to the 'x' parameter in the 'ajout' method signature, and the other line connects 'b' to the 'y' parameter. This visualizes how the values of 'a' and 'b' are passed to the 'ajout' method.

La surcharge de méthode

- On parle de **surcharge** de méthode lorsque deux ou plusieurs méthodes portent le **même nom** mais **pas les mêmes types** ou **pas le même nombre de paramètres**.
- Lors de l'appel, la méthode invoquée est sélectionnée en fonction de ses arguments (types ou nombre)

Exemple :

On souhaite écrire une méthode qui calcule la somme de deux entiers quel que soit leur type.

Exemple

```
public class Operation {  
    static int somme(int x, int y) {  
        return x + y;  
    }  
    static int somme(int x, int y, int z) {  
        return x + y + z;  
    }  
    static double somme(int x, double y) {  
        return x + y ;  
    }  
  
    public static void main(String args[]) {  
        System.out.println(somme(1,2));  
        System.out.println(somme(1,2,3));  
        System.out.println(somme(1,2.5));  
    }  
}
```

Remarque

- Si une méthode ne retourne rien, Java ajoute implicitement un return à la fin du corps de la méthode :

```
public static void main(String[] args) {  
    int res = ajout(1, 2);  
    System.out.println("1 + 2 = " + res);  
}
```



```
public static void main(String[] args) {  
    int resultat = ajout(1, 2);  
    System.out.println("1 + 2 = " + res);  
    return;  
}
```

Localité des variables

- Variable locale = variable définie dans une méthode
 - N'existe que le temps de l'invocation de la méthode
 - Il en va de même des paramètres de la méthode
- Lors d'une invocation de méthode, l'environnement d'exécution :
 - Crée un cadre d'appel pour accueillir les variables locales et les paramètres ;
 - Crée les variables locales/paramètres dans le cadre ;
 - Affecte les paramètres.
- À la fin de l'invocation, l'environnement d'exécution
 - Détruit le cadre, ce qui détruit les variables locales/paramètres

Exemple

```
public class Localité {  
    static void augmenter(int val)  
    {val=val+5;  
    System.out.println("valeur de val dans la procédure  
"+val);}  
    public static void main(String[] args) {  
        int val = 10;  
        augmenter(val);  
        System.out.println("valeur de val en sortie : "+val);  
    }  
}
```

Résultat :

```
valeur de val dans la procédure 15  
valeur de val en sortie : 10
```

Passage des paramètres par valeur et par référence

- Le passage des arguments peut se faire par
 - Valeur : l'appelé reçoit une copie d'une valeur
 - la copie et l'originale **sont différentes**
 - Référence : l'appelée reçoit une référence vers une valeur
 - la valeur est **partagée** entre l'appelant et l'appelé
- En Java :
 - Passage par valeur pour les 8 types primitifs
 - (**boolean, byte, char, short, int, long, float et double**)
 - Passage par référence pour les autres types
 - (**String, tableaux, ...**)

Exemple

```
import java.util.Arrays;
public class Localité {
    public static void main(String[] args) {
        int [] tab =new int [10];
        System.out.println("avant " +Arrays.toString(tab));
        passPar(tab);
        System.out.println("après " +Arrays.toString(tab));
    }
    static void passPar(int [] tab) {
        for(int i=0;i<tab.length;i++) tab[i]=i+1;
    }
}
```

Résultat :

- avant [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- après [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]