Chapitre 4 : Les fonctions

▼ Introduction aux fonctions

Une `fonction` est un **bloc d'instructions** qui peut être exécuté à **différents endroits** du programme. Il s'agit donc d'un bout de **code réutilisable** à différents endroits dans le programme.

Les fonctions en Java sont en fait des **méthodes statiques** : ça veut dire qu'elles sont définies **obligatoirement** à l'intérieur d'une **class** sans pour autant être des fonctions liées aux **objets**.

C'est une **spécificité** de **Java** de devoir définir l'ensemble des fonctions à l'intérieur des class même si l'on ne souhaite pas faire de la `POO` (**Programmation Orientée Objet**). Cette spécificité technique est due au fonctionnement de la **compilation** Java : **javac** travaille sur l'ensemble des class nécessaires au fonctionnement du programme à compiler.

Nous verrons en détails comment définir des class dans le chapitre sur la **POO**. **Il est important de retenir qu'on ne peut pas définir des fonctions en dehors des class en Java**, ce qui est différent de la plupart des langages de programmation.

Déclarer une fonction

Voici des exemple de déclaration de fonctions :

```
public class ExempleDeFonctions {
    public static void uneFonctionPublique(){

    }

    private static void uneFonctionPrivee(){

    }
}
```

> **Rappel : ces fonctions sont forcément présentes au sein d'une `class`.**

Les mots-clés public et private

Une fonction `public` peut être appelée par n'importe quelle autre fonction dans le code.

Une fonction `private` ne peut être appelée que par des fonctions de la même `class` que celles de la fonction `private`.

Il faut obligatoirement préciser si les fonctions sont `public` ou `private`.

Le mot-clé static

Dans ce chapitre, nous définirons toujours des **fonctions** / **méthodes statiques**. Pour cela, il faut ajouter le mot-clé `static` à la définition de la fonction.

Sans le mot-clé `static`, cela correspond alors à des **méthodes pour les objets**. Nous verrons comment celles-ci fonctionnent dans le chapitre sur la `POO`.

Le mot-clé void

Il permet de dire que la fonction définie **ne retourne aucune valeur.**

Fonctions avec retour de valeurs

Sans le mot-clé `void`, la fonction doit o**bligatoirement retourner une valeur**. Dans ce cas, il est **obligatoire** de préciser le **type de retour** comme dans l'exemple suivant :

```
public static String assalamuAlaykum() {
    return "Assalamu alaykum";
}
```

Le type de retour est précisé avant le nom de la méthode, ici **String**. De plus, il est obligatoire de retourner une valeur au sein de la fonction : le mot clé `return` est alors utilisé.

> **Note : Il n'est possible de retourner qu'une seule valeur.**

Les paramètres

Une fonction peut accepter des **paramètres** d'entrée comme dans l'exemple suivante :

```
public static int addition(int x, int y) {
    return x + y;
}
```

Les **paramètres** d'entrée ont obligatoirement un type.

Conventions de nommage

Les **fonctions** suivent la convention de nommage du `camelCase`. **Tous les mots** après le **premier mot** commencent par une **majuscule** et le reste est en **minuscule**.

Les `class` suivent la convention de nommage du `PascalCase`. **Tous les mots** commencent par une **majuscule** (y compris le premier). Le reste est en **minuscule**.

Appeler une fonction

Une **fonction** peut être appelée **n'importe où dans le programme si celle-ci est** `public`.

Par contre, si elle est `private`, **il n'est possible de l'appeler que dans les fonctions de la** `class`.

Voici des exemples d'appels de fonctions :

```
uneFonction(); // Appel d'une fonction void
afficher("Assalamu", "Aleykoum!"); // Appel d'une fonction qui accepte des paramètres
var result = addition(12, 3); // Appel d'une fonction qui retourne une valeur
```

La fonction main

Tout programme **Java** possède un **point d'entrée** : **la fonction** `main`. Il s'agit d'une **fonction** qui sera toujours **appelée au début de l'exécution du programme**. Pour rappel, elle est définie ainsi :

```
public class Main {
    public static void main(String[] args) {

    }
}
```

C'est donc une fonction un peu particulière qui ne sera jamais appelée **explicitement**, mais qui sera appelée **automatiquement** au lancement du programme.

Elle est définie avec les **mots-clés** `static` et `void` car il s'agit d'une **méthode statique** qui ne retourne **aucune valeur**. De plus, elle est `public` puisqu'il s'agit du `point d'entrée` du programme et la fonction doit donc être appelée lors de l'utilisation de la commande `java`.

Le paramètre d'entrée args

La fonction `main` reçoit le **paramètre d'entrée** suivant : `String[] args`.

Il s'agit d'un **tableau de chaînes de caractères** qui possède **les valeurs d'entrée du programme**. Pour passer des **paramètres d'entrée** à un programme, il est possible de les définir lors du l'utilisation de la commande java à l'exécution du programme :

```
java Main argument1 argument2 argument3
```

Dans cet exemple, le tableau args est alors un tableau de taille 3 possédant les chaînes de caractères argument1, argument2 et argument3.

Pour rappel, il est alors possible d'itérer sur les arguments d'entrée de la manière suivante avec une boucle for :

```
for (String arg : args){
    System.out.println(arg);
}
```

▼ Les paramètres et arguments

Paramètres et arguments

Un peu de vocabulaire est nécessaire autour des **paramètres** et **arguments** pour bien comprendre comment les **fonctions** sont définies.

Les paramètres d'une fonction sont les **noms listés dans la définition** de la fonction. **Les arguments d'une fonction** sont les **valeurs réelles passées** à la fonction. Les **paramètres** sont **initialisés** avec les valeurs des **arguments** fournis.

Voici un exemple plus visuel :

```
//Définition des paramètres en entrée de la fonction
public static void afficher(String nom, String prenom){
    System.out.println(nom + prenom);
}

afficher("NDIAYE", "Babacar") //On passe les arguments "NDIAYE" et "Babacar"
// à la fonction. Cette ligne est utilisable dans une fonction
```

Arguments positionnels

En Java, l'initialisation des **paramètres** se fait forcément dans **le même ordre que celui des arguments**. Donc en utilisant la fonction **afficher** de l'exemple précédent :

```
afficher("NDIAYE", "Babacar"); //Initialise nom avec "NDIAYE" et prenom avec "Babacar"
afficher("Babacar", "NDIAYE"); //Initialise nom avec "Babacar" et prenom avec "NDIAYE"
```

Il faut donc faire attention à passer les arguments dans l'ordre attendu par la définition de la fonction.

Les types

Les **paramètres d'entrée** d'une fonction sont **forcément typés** :

```
public static int addition(int x, int y) {
    return x + y;
}
```

Dans l'exemple précédent, les types des paramètres sont des entiers. Il n'est donc pas possible d'utiliser autre chose que des entiers en tant qu'arguments lors des invocations de la fonction addition :

```
addition(3, 12); //OK

addition("Salut", 2); //Erreur de compilation car erreur de type
```

Les varargs (various arguments)

Il est possible de définir des paramètres indéfinis en taille en utilisant la notation `...` qui correspond aux `varargs` :

```
public static void afficher(String... chaines){
    for (String chaine : chaines) {
        System.out.println(chaine);
    }
}
//Exemple d'invocation à utiliser dans une fonction
afficher("Babacar", "Maman", "Awa");
```

Comme on peut le voir dans l'exemple précédent, il est alors possible d'itérer sur les éléments du `vararg` **chaines**. Un `vararg` est en fait un t**ableau techniquement parlant**. Il s'agit ici d'un **sucre syntaxique** qui permet d'éviter de définir un tableau lors de l'invocation (appel) de la fonction : on passe alors autant d'arguments que nécessaire plutôt qu'un tableau.

Limitation

Il est possible de définir des `varargs` avec d'autres **paramètres** lors de la définition de fonctions. Cependant, le `vararg` **doit obligatoirement être en dernière position** :

```
public static void afficher(String nom, String prenom, String... chaines){} //OK
public static void afficher(String nom, String... chaines, String prenom){} //Erreur
```

Il n'est d'ailleurs pas possible de définir **plusieurs** `varargs` au sein de la même fonction.

▼ Passage par valeur ou référence

Passage par valeurs

Dans l'exemple suivant :

```
public static void modify(int x) {
//Modification de la valeur local uniquement car il ne s'agit pas d'une référence
    x = 12;
}

public static void main(String[] args) {
    var x = 1;
    modify(x);
    System.out.println(x); //Affiche 1 car non modifié
}
```

Lors de l'utilisation des **types natifs** Java en tant qu'arguments d'une fonction, l'initialisation de la **variable locale** `x` de la fonction se fait par **valeur** et non pas par **référence**. Il n'est donc pas possible de modifier l'argument utilisé par la fonction `main` au sein de la **fonction** `modify`.

Passage par référence

Dans l'exemple suivant :

```
public static void modify(ArrayList<String> x) {
    x.add("c"); // Modifier la référence
}

public static void main(String[] args) {
    var x = new ArrayList<String>();
    x.add("a");
    x.add("b");
    modify(x); // Va modifier la liste car il s'agit d'une référence
    System.out.println(x); // Affiche [a, b, c] car modifié par la fonction
}
```

Contrairement au passage par **valeur**, on passe ici une **référence** car on utilise ici un **objet**. Il est alors possible de le modifier dans la fonction car il s'agit d'un **objet mutable**.

> **Attention** : Il n'est pas conseillé de modifier des objets dans des fonctions séparées comme dans l'exemple précédent car cela entraine un risque accru de bugs.

Portée des variables

Dans les exemples précédents, il est a noté que les variables s'appellent `x` dans la fonction main ainsi que dans la fonction `modify`.

Ce sont bien **deux variables différentes** à chaque fois car celles-ci sont locales aux fonctions dans lesquelles elles sont définies.

Pour résumer, au sein d'une fonction, **il est impossible de modifier les variables définies dans d'autres fonctions**. Il a été possible de modifier la liste dans le second exemple **car les variables x des deux fonctions référençaient la même liste**. En aucun cas, les variables n'ont été réassignées dans ces exemples. **Il faut faire particulièrement attention aux modifications quand on travaille avec des références**.

▼ Documentation des fonctions

La javadoc

Il est possible de créer de la documentation pour du code Java. On utilise pour cela la **javadoc**.

Pour les fonctions, la **javadoc** permet de décrire :

- **le fonctionnement de la fonction**
- **les paramètres d'entrée**
- **le retour de la fonction s'il y en a un**

Pour documenter une fonction, on utilise une syntaxe très proche des **commentaires multi-lignes**, comme le montre l'exemple ci-dessous :

```
/**
   * Cette fonction retourne l'addition de deux entiers passés en paramètres.
   * @param x  Le premier entier à additionner.
   * @param y  Le second paramètre à additionner.
   * @return int  Le résultat de l'addition.
*/
public static int addition(int x, int y) {
    return x + y;
}
```

On décrit le **fonctionnement** de la fonction dans la première partie de la documentation.

On **décrit** les **paramètres d'entrée** ensuite avec le mot-clé `@param`. Juste après le mot clé, on écrit le **nom du paramètre** pour mentionner quel paramètre est documenté suivi de la **description** du paramètre.

Enfin, on **décrit** le **retour de la fonction** avec le mot clé **@return**. On précise ensuite le **type de retour** de la fonction et enfin la **description** de la valeur retournée.

Quelle utilité ?

La javadoc est surtout utile dans les deux cas suivants:

- Lorsque l'on travaille sur une librairie utilisable par d'autres développeurs
- Lorsque l'on travaille en équipe sur un même logiciel

Il est conseillé de bien documenter toutes les fonctions qui utilisent le mot clé `public` car elles seront utilisables par d'autres développeurs. Cela permet de leur donner toutes les clés pour bien travailler avec les fonctions que vous aurez défini.

> **Note :** Nous verrons dans les chapitres suivants qu'il est également possible de documenter les `class` avec la javadoc.

Générer la javadoc

Pour générer la **javadoc**, il est possible d'utiliser la commande suivante :

```
javadoc -d javadoc src/*
```

Cette commande génère toute la **javadoc** des fichiers présents dans le dossier src. Le résultat de la génération se trouve alors dans le dossier **javadoc**.

> **Note :** Sans générer de javadoc manuellement, les éditeurs de code modernes sont capables de créer un visuel de la javadoc quand on essaie d'utiliser une fonction. Écrire la javadoc dans le code de vos programmes donnera déjà beaucoup d'aide aux autres développeurs même sans la générer.