MODULARISATION

PARAMÉTRER LES MÉTHODES

PARAMÈTRES DE MÉTHODE

Les **paramètres** vont nous permettre de créer des méthodes qui résolvent non plus *un* problème, mais toute une famille de problèmes

Écrire de telles méthodes requiert de généraliser, de voir au-delà d'une tâche simple pour modéliser la catégorie plus générale que dont cette tâche fait partie

La capacité de généralisation est l'une des plus grande qualité du développeur, et les **méthodes paramétrées** sont l'une des plus puissantes techniques à notre disposition.

INTUITION

« *Afficher 40 tirets* », « *Afficher 65 tirets* » : deux tâches très similaires pour lesquelles on doit pour l'instant écrire deux méthodes au comportement presque identique

On aimerait pouvoir extraire dans une variable ce qui change entre ces deux algorithmes : le nombre de tirets

- on pourrait alors utiliser ce nombre dans une boucle pour afficher le nombre de tirets souhaités

EXEMPLE

```java
public static void main(String arg[]) {
  afficher40Tirets();
  System.out.println();
  afficher65Tirets();
  System.out.println();
}

public static void afficher40Tirets() {
  for (int i = 1; i <= 40; i++) {
    System.out.print("-");
  }
}

public static void afficher65Tirets() {
  for (int i = 1; i <= 65; i++) {
    System.out.print("-");
  }
```

ON AIMERAIT BIEN POUVOIR...

```
public static void main(String arg[]) {
  int nbTirets = 80;
  afficherTirets();   // On voudrait « transmettre » "nbTirets" à la méthode...
}

public static void afficherTirets() {
  // Cette boucle ne passe pas la compilation :
  // "nbTirets" n'est pas visible dans cette méthode...
  for (int i = 1; i <= nbTirets; i++) {
    System.out.print("-");
  }
}
```

Ce qu'on veut, c'est **paramétrer** la méthode pour qu'elle puisse s'adapter au besoin et modéliser la fonctionnalité générique « *afficher n tirets* »

PARAMÈTRES

Paramètre : n'importe quelle caractéristique qui distingue différentes membres d'une famille de traitements similaires

En pratique, c'est une **valeur** que l'appelant passe à la méthode appelée

Un paramètre `nbTirets` permettrait de distinguer toutes les méthodes capables d'afficher un certain nombre de tirets. On n'a donc besoin que d'une seule méthode **paramétrée**

Quand on *appelle* la méthode, on spécifie combien de tirets on veut afficher

DÉCLARATION D'UNE MÉTHODE PARAMÉTRÉE

Un paramètre est indiqué dans l'entête de la méthode, entre les parenthèses qui, pour l'instant, étaient toujours restées vides

Comme pour une déclaration de variable, le nom **et** le type doivent être précisés

- Convention de nommage : **camelCase**

```
public static void maMéthode(<type> <nomParamètre>) {
  // instructions pouvant utiliser le paramètre
}
```

```
public static void afficherCode(int code) {
  System.out.println("Le code est : " + code);
}
```

APPEL D'UNE MÉTHODE PARAMÉTRÉE

```
maMéthode(<expression>);
```

```
// Exemple d'appel pour la méthode 'afficherCode'
afficherCode(1234);
```

```
______
SORTIE

Le code est : 1234
```

AFFICHERTIRETS - VERSION PARAMÉTRÉE

```java
public static void main(String arg[]) {
  afficherTirets(10);   // La valeur 10 est transmise au paramètre de la méthode
  System.out.println();
  afficherTirets(25);   // La valeur 25 est transmise au paramètre de la méthode
}

public static void afficherTirets(int nb) {
  // nb est initialisée à la valeur passée lors de l'appel
  // (ici c'est d'abord 10, puis 25)
  for (int i = 1; i <= nb; i++) {
    System.out.print("-");
  }
}
```

```
______
SORTIE

----------
-------------------------
```

MÉCANIQUE DU PASSAGE DE PARAMÈTRE

Lors de l'**appel** de méthode, ce qui est entre parenthèses est une **expression** qui va être **complètement évaluée** *avant* l'appel effectif

- exactement comme ce qui est entre parenthèses du `println` est évalué avant d'être affiché (`println` est elle-même une méthode paramétrée)
- comme d'habitude, l'expression peut être arbitrairement complexe

```
System.out.println("IMC : " + poids / (taille * taille)); // "IMC : 25.95..." est passée à printl
afficherCode(30 * 10);                    // la valeur 300 est passée
afficherTirets(1050 % 100 / 2);           // la valeur 25 est passée
```

ARGUMENT VS. PARAMÈTRE

On appelle **paramètre** la *variable* qui apparaît dans l'entête de la méthode

On appelle **argument** la *valeur* qui est passée lors de l'appel de la méthode

On dit aussi parfois **paramètre formel** pour paramètre et **paramètre effectif** pour argument

```
public static void main(String[] args) {
  afficherTripleDe(100 * 4);   // argument (ou paramètre effectif) : 400
}

public static void afficherTripleDe(int nb) { // paramètre (ou paramètre formel): 'nb'
  int triple = 3 * nb;
  System.out.println("Triple de " + nb + " = " + triple);
}
```

MÉCANIQUE DE LA MÉTHODE APPELÉE

La méthode appelée traite le paramètre **comme si c'était une variable locale**

- elle fait exactement comme si la première instruction de la méthode était une déclaration/initialisation d'une variable dont le nom est celui du paramètre
- elle initialise cette variable avec la valeur passée en argument
- conséquence : la variable éventuellement utilisée comme argument **n'est pas affectée** par d'éventuelles modification du paramètre dans la méthode appelée (on y revient plus loin)

MÉCANIQUE DU PASSAGE - EXEMPLE

```java
public static void main(String[] args) {
  afficherEspaces(20);
}

public static void afficherEspaces(int nb) {
  for (int i = 1; i <= nb; i++) {
    System.out.print(" ");
  }
}
```

```java
// C'est comme si "afficherEspaces" se comportait ainsi :
public static void afficherEspaces() {
  int nb = 20;                    // valeur passée en argument
  for (int i = 1; i <= nb; i++) {
    System.out.print(" ");
  }
}
```

ERREUR COMMUNE - INDICATION DU TYPE DANS L'APPEL

NE PAS inclure le type dans l'appel de méthode

```
int nbEspacesSouhaité = 20;
afficherEspaces(int nbEspacesSouhaité);  // NON : on ne doit pas préciser le type
```

ERREUR COMMUNE - TYPE INCOMPATIBLE

- C’est dans la définition de la méthode que le type du paramètre est indiqué
- Le type effectif de l’argument passé doit correspondre

```
public static void afficherEspaces(int nb) { // La méthode s'attend à recevoir un int
  for (int i = 1; i <= nb; i++) {
    System.out.print(" ");
  }
}

public static void main(String[] args) {
  afficherEspaces(20.0); // NON: un double ne peut pas être automatiquement converti en int
}
```

ERREUR COMMUNE - PAS LE BON NOMBRE DE PARAMÈTRES

Si la méthode est paramétrée, il est interdit de l'appeler sans préciser l'argument

```java
public static void afficherEspaces(int nb) { // La méthode s'attend à recevoir un int
  for (int i = 1; i <= nb; i++) {
    System.out.print(" ");
  }
}

public static void main(String[] args) {
  afficherEspaces(); // NON : un argument entier doit obligatoirement être passé
}
```

ERREUR COMMUNE - RÉCUPÉRER UNE VALEUR PAR LE PARAMÈTRE

Rappel : **un paramètre est une variable locale à la méthode** ⇒ c'est une **copie** de la valeur passée en argument

Conséquence : modifier un paramètre formel dans une méthode n'a aucune conséquence sur les variables déclarées chez l'appelant

EXEMPLE

```java
public static void modifier(int nombre) {
  System.out.println("dans modifier : " + nombre);
  nombre = nombre + 10;
  System.out.println("dans modifier : " + nombre);
}

public static void main(String[] args) {
  int nombre = 33;
  System.out.println("dans main : " + nombre);
  modifier(nombre);
  System.out.println("dans main : " + nombre);
}
```

```
dans main : 33
dans modifier : 33
dans modifier : 43
dans main : 33
```

QUELLE EST LA SORTIE ?

```java
public static class Youplaboum {
  public static void main(String[] args) {
    int a = 9;
    int b = 2;
    int c = 5;

    mystere(c, b, a);
    mystere(b, a, c);
  }

  public static void mystere(int a, int c, int b) {
    System.out.println(c + " et " + (b - a));
  }
}
```

```
______
SORTIE

2 et 4
9 et 3
```

```java
public class QuelleEstLaSortie {
  public static void main(String[] args) {
    int nb = 17;
    doublerNombre(nb);
    System.out.println("nb = " + nb);

    int nombre = 42;
    doublerNombre(nombre);
    System.out.println("nombre = " + nombre);
  }

  public static void doublerNombre(int nombre) {
    System.out.println("Valeur initiale = " + nombre);
    nombre = nombre * 2;
    System.out.println("Valeur finale = " + nombre);
  }
}
```

```
Valeur initiale = 17
Valeur finale = 34
nb = 17
Valeur initiale = 42
Valeur finale = 84
nombre = 42
```

CONSÉQUENCE DE CETTE GESTION DES PARAMÈTRES

On est sûrs que les variables de la méthode appelante sont protégées des modifications, puisque les paramètres effectifs sont des copies des arguments

Du coup, les paramètres nous permettent *d'envoyer* des valeurs à une méthode, mais ne nous permettent pas de *recevoir* des valeurs en retour

C'est le rôle de la *valeur de retour* (chapitre suivant)

SYNTAXE GÉNÉRALE D'UNE MÉTHODE PARAMÉTRÉE

```
public static void nomDeMéthode(<type> <nom>, <type> <nom>, ..., <type> <nom>) {
  <instruction>;
  <instruction>;
  ...
  <instruction>;
}
```

On peut déclarer **autant de paramètres que l'on veut** entre les parenthèses

- on les sépare par des virgules
- le type est obligatoire pour chaque paramètre (même si c'est le même à chaque fois)
- chaque paramètre est traité comme une variable locale à la méthode

SYNTAXE DE L'APPEL

Lors de l'appel, il faut préciser autant de paramètres que spécifiés dans la méthode

- Le type de chaque valeur passée doit être compatible avec le type du paramètre correspondant

```
nomDeMéthode(<expression>, <expression>, ..., <expression>);
```

VARIATION DE "AFFICHERTIRETS"

On veut afficher une série de caractères identiques, mais pas seulement des tirets (`'A'`, `'#'`, `'*'`...).

On peut faire du caractère à afficher un deuxième paramètre

```
public static void répéterCaractère(char car, int nbRépétitions) {
  for (int i = 1, i <= nbRépétitions; i++) {
    System.out.print(car);
  }
}

public static void main(String[] args) {
  répéterCaractère('#', 20);
  répéterCaractère('-', 10);
  répéterCaractère('#', 20);
}
```

```
####################----------####################
```