Méthodes

Modularisation d’un programme

Rappel - Structure d’un programme Java

```
public class NomDeClasse {
  public static void main(String[] args) {
    instruction;
    instruction;
    ...
    instruction;
  }
}
```

- Tout programme Java contient (au moins) une **classe**
  - qui contient une **méthode** appelée `main`
    - qui contient des **instructions** à exécuter

Rappel - Algorithme

- ⇒ description de la résolution étape par étape d’un problème
- Ex : *cuisiner des cookies*

```
- Préchauffer four 180°c
- Mélanger les ingrédients secs
- Mélanger beurre + sucre
- Intégrer beurre + sucre
- Battre les blancs en neige
- Intégrer blancs en neige
- Faire des boules aplaties
- Les placer dans le four
- Attendre 15 minutes
- Sortir les cookies
- Mélanger les ingrédients pour la garniture
- Appliquer la garniture sur les cookies
```

Structure

- Cet algorithme :
  - **manque de structure** : juste une longue liste d'étapes
  - **est redondant** : que se passe-t-il si on veut faire la cuisson en 2 fournées ?
    - ⇒ on doit recopier le code de cuisson
- La redondance induit des problèmes (**duplication**) :
  - la cuisson change ? ⇒ il faut modifier les deux parties
  - une troisième fournée ? ⇒ il faut encore recopier tout le traitement

Algorithme structuré

- Pour régler ces problèmes de structure et de duplication, on va **modulariser** l'algorithme, c'est-à-dire le découper en sous-problèmes
- Pour cela, il faut appliquer la procédure suivante :
  - Identifier les sous-problèmes
  - Nommer les sous-problèmes
  - Extraire les sous-problèmes

Identification des sous-problèmes

- Il faut **décomposer** l'algorithme de manière cohérente en gardant ensemble les parties qui sont au même niveau d'abstraction et qui constituent un sous-problème complet

```
- Préchauffer four 180°c
- Mélanger les ingrédients secs
- Mélanger beurre + sucre
- Intégrer beurre + sucre
- Battre les blancs en neige
- Intégrer blancs en neige
- Faire des boules aplaties
- Les placer dans le four
- Attendre 15 minutes
- Sortir les cookies
- Mélanger les ingrédients pour la garniture
- Appliquer la garniture sur les cookies
```

```
- Préchauffer four 180°c
-----------------------------------------
- Mélanger les ingrédients secs
- Mélanger beurre + sucre
- Intégrer beurre + sucre
- Battre les blancs en neige
- Intégrer blancs en neige
- Faire des boules aplaties
-----------------------------------------
- Les placer dans le four
- Attendre 15 minutes
- Sortir les cookies
-----------------------------------------
- Mélanger les ingrédients pour la garniture
- Appliquer la garniture sur les cookies
```

Nommage des sous-problèmes

- On **nomme** chaque sous-problème (le choix des noms est **très important** pour la lisibilité et la compréhension)

```
- Préchauffer four 180°c
-----------------------------------------
- Mélanger les ingrédients secs
- Mélanger beurre + sucre
- Intégrer beurre + sucre
- Battre les blancs en neige
- Intégrer blancs en neige
- Faire des boules aplaties
-----------------------------------------
- Les placer dans le four
- Attendre 15 minutes
- Sortir les cookies
-----------------------------------------
- Mélanger les ingrédients pour la garniture
- Appliquer la garniture sur les cookies
```

```
- Préchauffer four 180°c
- RÉALISER L'APPAREIL
  - Mélanger les ingrédients secs
  - Mélanger beurre + sucre
  - Intégrer beurre + sucre
  - Battre les blancs en neige
  - Intégrer blancs en neige
  - Faire des boules aplaties
- CUIRE LES COOKIES
  - Les placer dans le four
  - Attendre 15 minutes
  - Sortir les cookies
- DÉCORER LES COOKIES
  - Mélanger les ingrédients pour la garnitu
  - Appliquer la garniture sur les cookies
```

Extraction des sous-problèmes

- On **extrait** chaque sous-problème en dehors de l’algorithme principal

```
- Préchauffer four 180°c
- RÉALISER L'APPAREIL
  - Mélanger les ingrédients secs
  - Mélanger beurre + sucre
  - Intégrer beurre + sucre
  - Battre les blancs en neige
  - Intégrer blancs en neige
  - Faire des boules aplaties
- CUIRE LES COOKIES
  - Les placer dans le four
  - Attendre 15 minutes
  - Sortir les cookies
- DÉCORER LES COOKIES
  - Mélanger les ingrédients pour la garnitu
  - Appliquer la garniture sur les cookies
```

```
Sous-problème "RÉALISER L'APPAREIL"
- Mélanger les ingrédients secs
- ...

Sous-problème "CUIRE LES COOKIES"
- Les placer dans le four
- ...

Sous-problème "DÉCORER LES COOKIES"
- Mélanger les ingrédients pour la garniture
- ...

Problème principal "CUISINER DES COOKIES"
- Préchauffer four 180°c
- Appel de "RÉALISER L'APPAREIL"
- Appel de "CUIRE LES COOKIES"
- Appel de "DÉCORER LES COOKIES"
```

Modularisation

```
- Problème "CUISINER DES COOKIES"
  - Préchauffer four 180°c
  - Appel de "RÉALISER L'APPAREIL"
  - Appel de "CUIRE LES COOKIES"
  - Appel de "DÉCORER LES COOKIES"
```

- L’algorithme général est maintenant bien plus lisible
  - on a une vision de plus haut niveau de ce que fait l’algorithme
  - si on a besoin des détails, on examine les algorithmes correspondants aux sous-problèmes
- Qu’en est-il des problèmes de duplication ?

Exemple de duplication

```
 1 - Préchauffer four 180°c
 2 - Mélanger les ingrédients secs
 3 - Mélanger beurre + sucre
 4 - Intégrer beurre + sucre
 5 - Battre les blancs en neige
 6 - Intégrer blancs en neige
 7 - Faire des boules aplaties
 8 - Placer le 1er fournée dans le four
 9 - Attendre 15 minutes
10 - Sortir les cookies
11 - Placer le 2ème fournée dans le four
12 - Attendre 15 minutes
13 - Sortir les cookies
14 - Mélanger les ingrédients pour la garnit
15 - Appliquer la garniture sur les cookies
```

```
 1 - Préchauffer four 180°c
 2 - Mélanger les ingrédients secs
 3 - Mélanger beurre + sucre
 4 - Intégrer beurre + sucre
 5 - Battre les blancs en neige
 6 - Intégrer blancs en neige
 7 - Faire des boules aplaties
 8 - Cuire les cookies (1ère fournée)
 9 - Cuire les cookies (2ème fournée)
10 - Mélanger les ingrédients pour la garnit
11 - Appliquer la garniture sur les cookies
```

La décomposition élimine la duplication

- La duplication est quasiment éliminée
  - ici, une boucle de 2 itérations appelant **Cuire les cookies** éliminerait complètement la duplication
  - mais les boucles ne permettent pas d'éliminer la duplication systématiquement (ex. : code dupliqué se trouvant à des endroits différents dans le code)
- Avec la décomposition :
  - la cuisson change ? ⇒ un seul endroit à modifier
  - une troisième fournée ? ⇒ on ajoute un appel à **Cuire les cookies**

Décomposition procédurale

- Ce mécanisme s’appelle la **décomposition procédurale**
  - ⇒ séparation d’une problème en plusieurs sous-problèmes
  - ⇒ chaque sous-problème est plus simple que le problème englobant
- Ces sous-problèmes peuvent eux-mêmes être décomposés en d’autres sous-problèmes au besoin
- Ex : **Battre les blancs en neige** peut être décomposé :

```
Sous-problème "BATTRE LES BLANCS EN NEIGE"
- Séparer les jaunes des blancs
- Battre 30 secondes au fouet
- Ajouter du sel
- Battre jusqu'à consistance ferme
```

Développement itératif

- Le **développement itératif** consiste à écrire un programme par étapes, en ajoutant une fonctionnalité à la fois
- L'un des grands bénéfices que l'on en tire est la capacité à **tester chaque partie au fur et à mesure** pour s'assurer que tout fonctionne bien avant de continuer
- Java permet d'appliquer la **décomposition procédurale** et le **développement itératif** grâce notamment aux **méthodes**

Méthodes

- Une **méthode** désigne un bloc d'instructions auquel on donne un nom
  - un peu comme une « commande » Java qu'on écrit nous-mêmes
  - ⇒ sous-problème qu'on a identifié et qu'on va isoler
  - permet la décomposition (**Faire l'appareil**, **Cuire les cookies**, **Décorer les cookies**…)
  - supprime la redondance (chaque « commande » pourra être répétée à volonté sans duplication de code)

Processus de décomposition

1. **Concevoir l'algorithme**
   - Repérer la structure de la résolution du problème
     - Quels sont les sous-problèmes importants ? Peuvent-ils eux-mêmes être décomposés en sous-problèmes ?
     - Dans tout cela, y a-t-il des choses qui sont répétées ?
2. **Définir (écrire) les méthodes**
   - Chaque sous-problème identifié sera dans sa propre méthode bien nommée
3. **Appeler (invoquer) les méthodes**
   - Dans la méthode `main` mais aussi dans n'importe quelle autre méthode si celle-ci est elle-même décomposée

Définir une méthode

```
public static void maMéthodeSuperBienNommée() {
  // instructions
}
```

- Convention de nommage : `camelCase` (commence par une minuscule)
- `void` est le **type de retour** : cette méthode ne renvoie rien à l'appelant (*void* = vide)
  - on appelle *appelant* la méthode qui a appelé la méthode courante
  - on verra qu'il est souvent utile de renvoyer une valeur à l'appelant
- L'ensemble d'instructions correspondantes à la méthode se trouve dans le bloc d'accolades { … }
- Même définie, **une méthode n'est pas exécutée tant qu'elle n'est pas appelée**

Appeler (invoquer) la méthode

```
// Appel de méthode (exécution)
maMéthodeSuperBienNommée();
```

- **Appel de méthode** : instruction qui provoque l'*exécution* de la méthode appelée
  - ce qui cause l'exécution de la *{ séquence d'instructions }* de cette méthode
- Quand la méthode est terminée, le contrôle est rendu à l'appelant
- C'est comme si on commandait à l'ordinateur : « *À chaque fois que je te dis* `maMéthodeSuperBienNommée` *, je veux que tu exécutes la séquence d'instructions qui se trouve dans cette méthode* »
- Une fois définie, une méthode peut être appelée autant de fois qu'on veut
- Appeler une méthode non définie entraîne une erreur de compilation

Appeler la méthode - Syntaxe

```
// Appel de méthode (exécution)
maMéthodeSuperBienNommée();
```

- Parenthèses obligatoires
- Point-virgule obligatoire (un appel de méthode est une instruction)
- Comme toujours, attention à la casse
- Faites confiance à l'*auto-complétion* de votre IDE pour faire respecter la casse et l’orthographe

Définition et appels de méthodes - Exemples

```java
public class WeWillRockYou {
  public static void main(String[] args) {
    System.out.println("We will, we will, rock you!");
    System.out.println();
    System.out.println("We will, we will, rock you!");
  }
}
```

```java
public class WeWillRockYou {
  public static void main(String[] args) {
    sing();
    System.out.println();
    sing();
  }

  public static void sing() {
    System.out.println("We will, we will, rock you!");
  }
}
```

```java
public class CuisinerCookies {

  public static void main(String[] args) {

    System.out.println("Mélanger les ingrédients secs");
    System.out.println("Mélanger beurre + sucre");
    System.out.println("Intégrer beurre + sucre");
    System.out.println("Battre les blancs en neige");
    System.out.println("Intégrer blancs en neige");
    System.out.println("Faire des boules aplaties");
    System.out.println("Placer la 1ère fournée dans le four");
    System.out.println("Attendre 15 minutes");
    System.out.println("Sortir les cookies");
    System.out.println("Placer la 2ème fournée dans le four");
    System.out.println("Attendre 15 minutes");
    System.out.println("Sortir les cookies");
    System.out.println("Mélanger les ingrédients pour la garniture");
    System.out.println("Appliquer la garniture sur les cookies");

  }
}
```

```java
public class CuisinerCookies {

  public static void main(String[] args) {

    // Réaliser l'appareil
    System.out.println("Mélanger les ingrédients secs");
    System.out.println("Mélanger beurre + sucre");
    System.out.println("Intégrer beurre + sucre");
    System.out.println("Battre les blancs en neige");
    System.out.println("Intégrer blancs en neige");
    System.out.println("Faire des boules aplaties");

    // Cuire fournée 1
    System.out.println("Placer la 1ère fournée dans le four");
    System.out.println("Attendre 15 minutes");
    System.out.println("Sortir les cookies");

    // Cuire fournée 2
    System.out.println("Placer la 2ème fournée dans le four");
    System.out.println("Attendre 15 minutes");
    System.out.println("Sortir les cookies");

    // Décorer les cookies
    System.out.println("Mélanger les ingrédients pour la garniture");
    System.out.println("Appliquer la garniture sur les cookies");

  }
}
```

```java
public class CuisinerCookies {
  public static void main(String[] args) {
    RéaliserAppareil();
    CuireCookies();
    CuireCookies();
    DécorerCookies();
  }
}
public static void RéaliserAppareil() {
  System.out.println("Mélanger les ingrédients secs");
  System.out.println("Mélanger beurre + sucre");
  System.out.println("Intégrer beurre + sucre");
  System.out.println("Battre les blancs en neige");
  System.out.println("Intégrer blancs en neige");
  System.out.println("Faire des boules aplaties");
}
public static void CuireCookies() {
  System.out.println("Placer les cookies dans le four");
  System.out.println("Attendre 15 minutes");
  System.out.println("Sortir les cookies");
}
public static void DécorerCookies() {
  System.out.println("Mélanger les ingrédients pour la garniture");
  System.out.println("Appliquer la garniture sur les cookies");
}
```

Méthodes - synthèse (1)

- Les méthodes rendent le code **plus facile à lire** : elles permettent de **décomposer le programme** et d'en **capturer la structure**
- Les **noms de méthodes efficaces** renforcent cette qualité
- La méthode `main` devrait être un **bon résumé** de ce qui se passe dans le programme
- Plus généralement, *toute méthode* devrait se lire relativement rapidement
  - les détails sont relégués aux méthodes de plus bas niveau (les sous-problèmes)

Méthodes - synthèse (2)

- Modulariser un code redondant permet :
  - d'avoir **un seul point de modification** pour ce code (et facile à retrouver)
  - de pouvoir **répéter** à volonté l'action décrite (pas de copier/coller !)
- Parfois, décomposer en méthodes rend le code plus long, surtout s'il n'y avait pas de redondance à éliminer
  - Mais **un code plus long ne signifie pas nécessairement un code de plus mauvaise qualité**

Quand NE PAS utiliser de méthodes

- Ne créez pas de méthodes pour :
  - exécuter des instructions ultra-simples (genre un `println` vide)
  - regrouper des traitements qui ne sont pas vraiment en relation (mauvaise décomposition)

Flux d’exécution

- Le flux d’exécution est comme un « curseur » qui se déplace d’instruction en instruction
- Ce curseur pointe toujours sur la prochaine instruction qui va être exécutée
- Au lancement du programme, l’environnement d’exécution place le curseur sur la première instruction de la méthode `main`
- Quand une instruction est terminée, le curseur passe à la suivante
- Le programme se termine lorsque le flux d’exécution atteint la fin de la méthode `main`

Flux d’exécution - Appels de méthodes

- Quand une méthode est appelée, le curseur :
  - « saute » dans la méthode concernée, sur la première instruction
  - suit le flux des instructions de la méthode
  - et « re-saute » au point où la méthode a été invoquée (*retour à l’appelant*)
- Le flux d’exécution peut ainsi suivre un tunnel d’appels imbriqués de méthodes, potentiellement très profond
  - mais le curseur sait toujours d’où il vient

Comprendre le flux d’exécution

```
public class Omelette {
  public static void main(String[] args) {
    BattreLesOeufs();
    CuireOmelette();
  }

  public static void BattreLesOeufs() {
    PrendreLesOeufs();
    CasserLesOeufsDansBol();
    Assaisonner();
    System.out.println("Battre les oeufs avec une fourchette");
  }

  public static void PrendreLesOeufs() {
    System.out.println("Ouvrir le placard en haut à droite");
    System.out.println("Prendre la boîte à oeufs");
    System.out.println("Fermer le placard en haut à droite");
  }

  // Autres méthodes non incluses...
}
```

Quelle est la sortie ?

```java
public class QuelleEstLaSortie {
  public static void foo() {
    System.out.println("chouette");
    baz();
  }

  public static void bar() {
    baz();
    System.out.print("machin ");
    foo();
  }

  public static void baz() {
    System.out.println("truc");
  }

  public static void main(String[] args) {
    foo();
    bar();
    System.out.println("bidule");
  }
}
```

Solution

- Voici l’ordre dans lequel les méthodes sont appelées :
  - *main, foo, baz, bar, baz, foo, baz*
- Et voici la sortie :

```
chouette
truc
truc
machin chouette
truc
bidule
```

Méthodes paramétrées

Rendre les méthodes plus génériques

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 grandes qualités du développeur, et les **méthodes paramétrées** sont l'une des plus puissantes techniques à notre disposition.

Intuition

- «*Afficher 10 tirets*», «*Afficher 25 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

```
public static void main(String arg[]) {
  afficher10Tirets();
  System.out.println();
  afficher25Tirets();
  System.out.println();
}

public static void afficher10Tirets() {
  for (int i = 1; i <= 10; i++) {
    System.out.print("-");
  }
}

public static void afficher25Tirets() {
  for (int i = 1; i <= 25; i++) {
    System.out.print("-");
  }
}
```

On aimerait bien pouvoir...

```
public static void main(String arg[]) {
  int nbTirets = 25;
  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

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

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

- Quand on appelle `afficherCode`, l'appelant **doit** spécifier un entier pour le paramètre `code`
- **C'est cette valeur *passée* qui va être utilisée pour initialiser le paramètre dans la méthode**

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 prin
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

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

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

```
// 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

```
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);
}
```

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

Type de retour de méthode

Permettre aux méthodes de renvoyer un résultat

Méthode - Renvoyer un résultat

- Les méthodes que nous avons vues jusqu'à présent étaient plutôt « orientées action » : `afficherCeci`, `dessinerCela`…
- Les paramètres permettent de les rendre plus génériques : **afficher un rectangle de taille `longueur x largeur`**…
- Les paramètres permettent de *transmettre* quelque chose à la méthode, mais pas vraiment de recevoir un résultat en retour
- On va voir comment écrire des méthodes qui **renvoient** quelque chose à l'appelant
- Ces méthodes répondent à des questions :
  - *Quelle est la racine carrée de 12.5 ?*
  - *Quel est le nom du client qui correspond à la référence de cette facture ?*

Exemple

- Chez l'appelant, on voudrait ça :

```
int racine = calculerRacine(9.0);  // affectation du résultat à la variable 'racine'
```

- Mais pour l'instant, on ne sait définir de méthodes qui ne font que des actions, comme afficher :

```
public static void calculerRacine(double n) {
  double racine = ...;
  System.out.println(racine);
}
```

- On voudrait **renvoyer** la valeur calculée à l'appelant, pour qu'il puisse utiliser le résultat, et non pas simplement l'afficher dans notre méthode

Type de retour

- Dans une définition de méthode, le mot juste avant le nom de la méthode définit le **type de retour** de la méthode
- Jusqu'à présent, c'était toujours `void` (vide) : nos méthodes ne retournaient rien
- On va remplacer ce mot-clé dans l'entête de la méthode par le type de la valeur qu'on veut retourner. La racine carré d'un double est aussi un double :
- L'instruction `return` est utilisée pour renvoyer la valeur à l'appelant

```
public static double calculerRacine(double n) {
  double racine = ...;
  return racine;
}
```

L’instruction return

```
return <expression>;
```

- Quand Java rencontre l’instruction `return`, il **évalue l’expression** puis la **renvoie à l’appelant**
- Conséquence : `return` **termine immédiatement la méthode** (retour à l’appelant)
- `return` est **obligatoire** dans toute méthode dont le type de retour n’est pas `void`
- Tous les types sont possibles comme type de retour d’une méthode

Exemple

- Écrivons une méthode `sommeDesEntiersJusque(int n)` qui calcule et retourne la somme des n premiers entiers : `1 + 2 + 3 + … + n`

```java
public static int sommeDesEntiersJusque(int n) {
  int somme = 0;
  for (int i = 1; i <= n; i++) {
    somme += i;
  }
  return somme;
}
```

```java
int somme = sommeDesEntiersJusque(100);
System.out.println("Somme : " + somme);


______
SORTIE

5050
```

Erreur commune - Ignorer la valeur de retour

- Si vous utilisez une méthode avec un type de retour, en général vous voudrez récupérer cette valeur (chez l'appelant)
- Il est fréquent (et même souhaitable) que ces méthodes ne fassent rien d'autre que calculer la valeur de retour
- Donc, si vous ne la récupérez pas, le traitement est perdu :

```
sommeDesEntiersJusque(1000); // résultat dans la nature
Math.round(2.6);             // résultat dans la nature
```

Récupérer et utiliser le résultat

- Vous pouvez utiliser un appel de méthode non-`void` dans n'importe quelle expression : à droite d'une affectation, au milieu d'une expression, comme argument dans un autre appel de méthode, etc.

```
int resultat = sommeDesEntiersJusque(1000); // OK : résultat pas perdu

double unFlottant = 2.5;
// Expression impliquant des appels de méthodes
double nb = Math.ceil(unFlottant) + Math.floor(unFlottant);

// Expression impliquant des appels de méthodes imbriqués
double racine = Math.sqrt(sommeDesEntiersJusque(Math.round(98.76)));

// Utilisation dans un println : toujours pareil,
// le résultat est utilisé comme valeur pour l'opérande.
// L'expression est totalement évaluée avant d'être affichée.
System.out.println("Un nombre aléatoire entre 0 et 1 : " + Math.random());
```