FORMATION JAVA POUR DÉBUTANTS

Apprendre le Java par Jonathan

Formation complète et pratique pour débutants

De zéro à vos premiers projets · Java 21 LTS

Un véritable livre de formation moderne — variables, boucles, objets, projets réels

ÉDITION 2026

Avant-propos

Bienvenue ! Vous tenez entre les mains un cours pensé pour une seule chose : vous faire passer de « je n'ai jamais programmé » à « je crée mes propres programmes Java », sans sauter aucune étape.

Ce livre ne suppose **aucune** connaissance préalable. Vous n'avez pas besoin de savoir ce qu'est une variable, une boucle ou un objet. Chaque mot technique est expliqué la première fois qu'il apparaît : ce qu'il signifie, à quoi il sert, pourquoi il est utile, un exemple concret, et une comparaison avec la vie de tous les jours.

> 📌 **À retenir**
>
> La programmation ne s'apprend pas en lisant, mais en **tapant du code**. Ouvrez votre ordinateur, recopiez les exemples, cassez-les, réparez-les. C'est ainsi que l'on apprend vraiment.

Comment lire ce livre ?

- **Dans l'ordre.** Chaque chapitre s'appuie sur le précédent.
- **Avec un clavier.** Faites tous les exercices avant de regarder les corrections (elles sont regroupées à la fin de chaque chapitre).
- **Sans se décourager.** Les erreurs font partie du métier. Un développeur passe une grande partie de son temps à corriger des bugs — vous apprendrez à les aimer.

Les encadrés que vous rencontrerez

> 📌 **À retenir**
>
> L'essentiel à mémoriser.

> 💡 **Conseil**
>
> Une astuce pour aller plus vite ou mieux comprendre.

⚠️ **Attention**

Un point délicat où l'on se trompe souvent.

🐞 **Erreur fréquente**

Une erreur classique de débutant et comment l'éviter.

✅ **Bonne pratique**

La façon propre et professionnelle de faire.

🎯 **Défi**

Un petit challenge pour aller plus loin.

📖 **Vocabulaire**

La définition simple d'un mot technique.

💡 **Conseil — la version de Java utilisée**

Ce cours utilise **Java 21 LTS**. « LTS » signifie *Long Term Support* (support à long terme) : c'est une version stable, maintenue plusieurs années, et compatible avec tous les outils. Une version encore plus récente existe (Java 25 LTS, sortie fin 2025) et tout le code de ce livre y fonctionne aussi. Pour débuter, Java 21 est le choix le plus sûr : documentation abondante, tutoriels partout, aucun risque.

Table des matières

1. **Introduction à la programmation orientée objet**

   Classes, objets, attributs, méthodes, constructeurs, encapsulation

2. **Héritage et polymorphisme**

   `extends`, `super`, `@Override` · Animal, Chien, Chat

3. **Les collections**

   `ArrayList`, `HashSet`, `HashMap`

ALLER PLUS LOIN

1. **La gestion des erreurs**

   Exceptions, `try`, `catch`, `finally`, `throw`

2. **Lire et écrire dans des fichiers**
3. **Les bonnes pratiques**
4. **Utiliser Git et GitHub**
5. **Déboguer un programme**

PRATIQUE

1. **Mini-projets guidés**

   Calculatrice · Nombre mystère · Convertisseur · Gestionnaire de tâches · Gestionnaire de contacts

2. **Projet final : gestionnaire de budget personnel**

RESSOURCES

1. **Évaluations et grille d'auto-évaluation**
2. **Plan d'apprentissage sur 8 semaines**
3. **Glossaire**
4. **Conclusion et feuille de route**

Chapitre 1 — Découvrir la programmation

🎓 **Objectifs d'apprentissage**

À la fin de ce chapitre, vous saurez :

- expliquer ce qu'est la programmation avec vos propres mots ;
- dire ce qu'est un langage de programmation et ce qu'est Java ;
- citer les domaines où Java est utilisé ;
- faire la différence entre Java et JavaScript ;
- comprendre le rôle du JDK, de la JVM et du compilateur ;
- distinguer le code source du programme compilé.

1.1 Qu'est-ce que la programmation ?

Programmer, c'est écrire une suite d'instructions très précises qu'un ordinateur va exécuter. L'ordinateur est extrêmement rapide et obéissant, mais il n'a aucune imagination : il fait exactement ce qu'on lui dit, ni plus ni moins. Si vous vous trompez d'une virgule, il ne « devine » pas ce que vous vouliez dire — il s'arrête ou fait n'importe quoi.

📖 **Vocabulaire — Programme**

Un **programme** est un ensemble d'instructions qui accomplit une tâche : une calculatrice, un jeu, une application bancaire ou un site web sont tous des programmes.

Analogie du quotidien : programmer, c'est comme écrire une recette de cuisine pour quelqu'un qui n'a jamais cuisiné et qui suit chaque phrase au pied de la lettre. « Casser 2 œufs » doit venir *avant* « battre les œufs ». Si vous inversez, le résultat est raté. L'ordinateur est ce cuisinier ultra-rapide mais ultra-littéral.

> 📖 **Vocabulaire — Algorithme**
>
> Un **algorithme** est la *logique* d'une solution : la suite d'étapes à suivre pour résoudre un problème, indépendamment du langage. « Pour trouver le plus grand nombre d'une liste, je regarde chaque nombre et je garde le plus grand vu jusqu'ici » est un algorithme. Le *code* est la traduction de cet algorithme dans un langage que la machine comprend.

1.2 Qu'est-ce qu'un langage de programmation ?

Un ordinateur ne comprend, au fond, que des **0 et des 1** (le langage binaire). Écrire directement en binaire serait insupportable pour un humain. On a donc inventé des **langages de programmation** : des langages intermédiaires, plus proches du langage humain, que l'on traduit ensuite pour la machine.

Il en existe des centaines : Java, Python, C, C++, JavaScript, C#… Chacun a ses forces. **Java** est l'un des plus utilisés au monde, réputé pour sa robustesse et sa portabilité.

> 📖 **Vocabulaire — Syntaxe**
>
> La **syntaxe** est l'ensemble des règles d'écriture d'un langage (où mettre les points-virgules, les accolades, etc.). Comme la grammaire d'une langue : si on ne la respecte pas, le message n'est pas compris.

1.3 Qu'est-ce que Java ?

Java est un langage de programmation créé en 1995 par la société Sun Microsystems (aujourd'hui propriété d'Oracle). Sa promesse historique tient en une phrase : « *Write once, run anywhere* » — « écrivez une fois, exécutez partout ». Un même programme Java peut tourner sur Windows, macOS, Linux, un serveur ou un téléphone Android, sans être réécrit.

Où utilise-t-on Java ?

Domaine	Exemples concrets
Applications d'entreprise	Banques, assurances, systèmes de gestion (le back-end de nombreuses grandes entreprises)
Applications Android	Une grande partie des applications mobiles Android
Sites et services web	Serveurs web avec Spring, Jakarta EE…
Big Data	Outils comme Hadoop, Kafka, Elasticsearch
Logiciels de bureau	Applications multiplateformes
Jeux vidéo	Minecraft (la version originale) est écrit en Java

1.4 Java n'est pas JavaScript !

> ⚠️ **Attention — une confusion très fréquente**
>
> **Java et JavaScript sont deux langages totalement différents**, malgré leurs noms voisins. C'est comme « carotte » et « carte » : proche à l'oreille, sans rapport. JavaScript a été nommé ainsi pour surfer sur la popularité de Java, mais ce sont deux technologies distinctes, avec des créateurs, des usages et des syntaxes différents.

	Java	JavaScript
Usage principal	Applications, serveurs, Android	Sites web interactifs (dans le navigateur)
Exécution	Compilé puis exécuté par la JVM	Interprété par le navigateur
Typage	Strict (on déclare le type)	Souple

1.5 Comment fonctionne Java ? (JDK, compilateur, JVM)

Voici le cœur du chapitre. Quand vous écrivez du Java, votre texte n'est pas directement compris par l'ordinateur. Il passe par plusieurs étapes.

📖 **Vocabulaire — Code source**

Le **code source** est le texte que vous écrivez, lisible par un humain. En Java, il est enregistré dans un fichier portant l'extension `.java`.

📖 **Vocabulaire — Compilateur**

Le **compilateur** (en Java : la commande `javac`) est un programme qui **traduit** votre code source en un langage intermédiaire appelé *bytecode*. Comme un traducteur qui transforme un texte français en une langue universelle intermédiaire.

📖 **Vocabulaire — Bytecode**

Le **bytecode** est le résultat de la compilation, enregistré dans un fichier `.class`. Ce n'est ni du texte lisible ni du binaire de votre machine : c'est un code intermédiaire que la JVM sait exécuter partout.

📖 **Vocabulaire — JVM (Java Virtual Machine)**

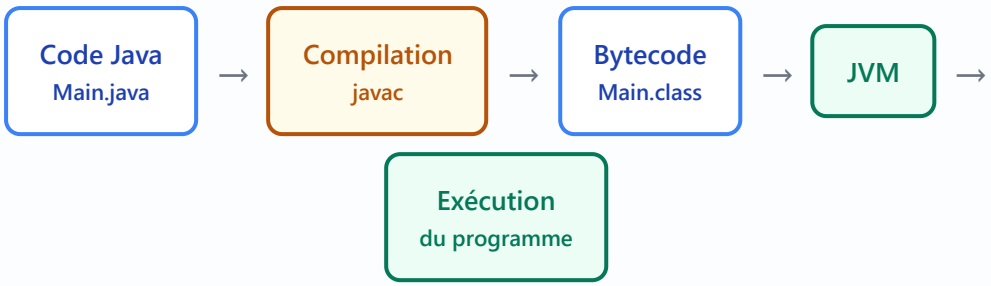
La **JVM**, ou « machine virtuelle Java », est un programme qui lit le bytecode et l'exécute sur *votre* machine. C'est elle qui rend Java portable : il existe une JVM pour Windows, une pour macOS, une pour Linux. Le même bytecode tourne sur toutes.

Analogie : le bytecode est un film numérique universel ; la JVM est le lecteur vidéo adapté à chaque appareil. Le film est le même, seul le lecteur change.

📖 **Vocabulaire — JDK (Java Development Kit)**

Le **JDK** est la « boîte à outils » complète du développeur Java. Il contient le compilateur (`javac`), la JVM (pour exécuter), et de nombreuses bibliothèques toutes prêtes. **Pour programmer en Java, il faut installer le JDK** (nous le ferons au chapitre 2).

Le parcours d'un programme Java



Vous écrivez le code → le compilateur le traduit en bytecode → la JVM lit le bytecode et fait tourner le programme.

📌 À retenir

- Vous écrivez du **code source** dans un fichier `.java` .
- Le **compilateur** (`javac`) le transforme en **bytecode** (`.class`).
- La **JVM** exécute ce bytecode sur n'importe quel système.
- Le **JDK** réunit tous ces outils. C'est ce qu'on installe.

Quiz — Chapitre 1

1. (QCM) Que fait le compilateur `javac` ?

a) Il exécute le programme b) Il traduit le code source en bytecode c) Il installe Java

2. (Vrai/Faux) Java et JavaScript sont le même langage.

3. (QCM) Quel fichier contient le bytecode ? a) `.java` b) `.txt` c) `.class`

4. (QCM) Qu'est-ce qui rend Java portable (« run anywhere ») ? a) Le compilateur b) La JVM c) Le fichier `.java`

5. (Avec vos mots) Expliquez en une phrase la différence entre code source et bytecode.

6. (Vrai/Faux) Le JDK contient le compilateur et la JVM.

💡 Réponses

1-b · 2-Faux · 3-c · 4-b · 5 : le code source est le texte lisible que j'écris (`.java`) ; le bytecode est sa version traduite (`.class`) que la JVM exécute · 6-Vrai.

Chapitre 2 — Installer son environnement de travail

🎓 **Objectifs d'apprentissage**

- installer le JDK (Java 21) sur Windows, macOS ou Linux ;
- vérifier l'installation avec `java --version` et `javac --version` ;
- installer un éditeur de code (IntelliJ IDEA, ou VS Code en alternative) ;
- créer un projet, un package, une classe, et lancer un programme ;
- lire la console et repérer la ligne d'une erreur.

📖 **Vocabulaire — IDE**

Un **IDE** (*Integrated Development Environment*, « environnement de développement intégré ») est un logiciel qui réunit tout ce qu'il faut pour programmer : un éditeur de texte intelligent, un bouton pour lancer le programme, un détecteur d'erreurs, un débogueur… C'est l'atelier du développeur. Nous utiliserons **IntelliJ IDEA Community Edition** (gratuit).

2.1 Étape 1 — Installer le JDK (Java 21)

Le JDK est indispensable. Nous prenons la distribution **Eclipse Temurin (Adoptium)**, gratuite, open source et très simple. Alternative : le JDK Oracle ou `jdk.java.net`.

Sur Windows

1. Rendez-vous sur **adoptium.net**.
2. Choisissez **Temurin 21 (LTS)**, système **Windows x64**, format **.msi**.
3. Lancez le fichier `.msi` téléchargé.
4. Pendant l'installation, cochez l'option **« Set JAVA_HOME variable »** et **« Add to PATH »** si elles sont proposées. C'est important : cela permet d'utiliser Java depuis le terminal.
5. Terminez l'installation (Suivant → Suivant → Installer).

Sur macOS

1. Sur **adoptium.net**, choisissez **Temurin 21 (LTS)**, système **macOS**. Prenez **aarch64** si votre Mac est récent (puce Apple M1/M2/M3…) ou **x64** pour un Mac Intel plus ancien.
2. Téléchargez le `.pkg` et lancez-le, puis suivez l'assistant.
3. *Alternative pratique* avec Homebrew : dans le Terminal, tapez `brew install --cask temurin@21`.

Sur Linux

Selon votre distribution, ouvrez un terminal :

```
# Ubuntu / Debian
sudo apt update
sudo apt install openjdk-21-jdk

# Fedora
sudo dnf install java-21-openjdk-devel

# Arch Linux
sudo pacman -S jdk21-openjdk
```

2.2 Étape 2 — Vérifier l'installation

Ouvrez un **terminal** (voir encadré ci-dessous) et tapez ces deux commandes, l'une après l'autre :

```
java --version
javac --version
```

```
RÉSULTAT ATTENDU (EXEMPLE)
openjdk 21.0.5 2024-10-15 LTS
OpenJDK Runtime Environment Temurin-21.0.5+11 (build 21.0.5+11-LTS)
javac 21.0.5
```

📖 **Vocabulaire — Terminal / invite de commandes**

Le **terminal** (appelé « invite de commandes » ou « PowerShell » sous Windows, « Terminal » sous macOS et Linux) est une fenêtre où l'on tape des commandes texte au lieu de cliquer. Pour l'ouvrir :

- **Windows :** touche Windows, tapez « PowerShell », Entrée.
- **macOS :** Cmd + Espace, tapez « Terminal », Entrée.
- **Linux :** Ctrl + Alt + T.

🐞 **Erreur fréquente**

Si le terminal répond `'java' n'est pas reconnu…` (Windows) ou `command not found: java` (macOS/Linux), c'est que le JDK n'est pas dans le PATH. **Solutions :** fermez et rouvrez le terminal ; réinstallez le JDK en cochant « Add to PATH » ; ou redémarrez l'ordinateur. Le *PATH* est la liste des dossiers où le système cherche les commandes.

2.3 Étape 3 — Installer IntelliJ IDEA Community Edition

1. Allez sur **jetbrains.com/idea**.
2. Téléchargez la version **Community Edition** (gratuite — attention à ne pas prendre « Ultimate » qui est payant).
3. Installez-la comme un logiciel classique (sur Windows, cochez « Add … to PATH » et l'association des fichiers `.java` si proposé).
4. Au premier lancement, acceptez les réglages par défaut. IntelliJ détecte souvent le JDK automatiquement.

Alternative : Visual Studio Code

Si vous préférez un éditeur plus léger :

1. Installez **VS Code** depuis **code.visualstudio.com**.
2. Dans l'onglet Extensions (icône en forme de carrés, à gauche), recherchez et installez le pack **« Extension Pack for Java »** de Microsoft. Il regroupe tout le nécessaire (langage, débogueur, gestion de projet).

> 💡 **Conseil**
>
> Pour ce cours, **IntelliJ IDEA est recommandé** : il guide davantage le débutant (création de projet, exécution en un clic, débogueur visuel). Nous décrirons IntelliJ, mais tout fonctionne aussi dans VS Code.

2.4 Étape 4 — Créer votre premier projet

Dans IntelliJ IDEA :

1. Écran d'accueil → **New Project**.
2. À gauche, sélectionnez **New Project** (type « Java »).
3. Champ **Name** : tapez `PremierProjet`.
4. Champ **JDK** : choisissez **21** (ou cliquez « Download JDK » si vide).
5. Décochez « Add sample code » pour partir d'une page blanche, puis **Create**.

> 📖 **Vocabulaire — Projet, package, classe**
>
> - Un **projet** est le dossier qui contient tout votre travail (code, réglages).
> - Un **package** est un sous-dossier qui range vos classes par thème, comme des dossiers dans une armoire. Exemple : `com.jonathan.calculatrice`.
> - Une **classe** est un fichier de code (`.java`). Pour l'instant, retenez : « une classe = un fichier ».

Créer un package et une classe

1. Dans le panneau de gauche, ouvrez le dossier `src`.
2. Clic droit sur `src` → **New** → **Package** → nommez-le `fr.jonathan.debut`.
3. Clic droit sur ce package → **New** → **Java Class** → nommez-la `Main` (avec un M majuscule).

> ⚠️ **Attention — nom de fichier = nom de classe**
>
> En Java, une classe `public` **doit** être dans un fichier portant exactement son nom. La classe `Main` vit dans `Main.java`. Java respecte la casse : `Main` ≠ `main`. IntelliJ crée le fichier pour vous, donc laissez-le faire.

2.5 Étape 5 — Lancer et lire la console

Nous écrirons notre premier vrai programme au chapitre 3. Pour l'instant, sachez que dans IntelliJ, on lance un programme en cliquant sur la **flèche verte** ▶ à côté de la méthode `main`, ou avec le raccourci **Maj + F10**.

> 📖 **Vocabulaire — Console**
>
> La **console** (ou « terminal de sortie ») est la zone, en bas de l'IDE, où le programme affiche ses messages et où apparaissent les erreurs. C'est votre principal moyen de « parler » avec le programme au début.

Reconnaître une erreur et trouver sa ligne

Quand un programme plante, Java affiche un message en rouge dans la console. Exemple :

```
CONSOLE — ERREUR
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at fr.jonathan.debut.Main.main(Main.java:7)
```

Ce message est votre **ami**, pas votre ennemi. Il dit :

- **Le type d'erreur :** `ArithmeticException: / by zero` → une division par zéro.
- **Le fichier et la ligne :** `Main.java:7` → l'erreur est à la **ligne 7** du fichier `Main.java`. Double-cliquez sur cette ligne bleue dans IntelliJ : l'éditeur vous y emmène directement.

> 💡 **Conseil — la règle d'or face à une erreur**
>
> Lisez toujours **la première ligne** du message (le type d'erreur) et **cherchez le numéro de ligne** de *votre* fichier. 90 % des débutants paniquent sans lire ; les 10 % qui lisent résolvent le problème en une minute.

> ✅ **Bonne pratique**
>
> Affichez les numéros de ligne dans l'éditeur (dans IntelliJ, ils sont visibles par défaut à gauche). Ils sont essentiels pour retrouver une erreur.

Quiz — Chapitre 2

1. (QCM) Quelle commande affiche la version du compilateur ? a) `java --version` b) `javac --version` c) `version java`

2. (Vrai/Faux) La classe publique `Main` doit être enregistrée dans un fichier `Main.java`.

3. (QCM) Dans `at ...Main.main(Main.java:7)`, que signifie `7` ? a) 7 erreurs b) la ligne de l'erreur c) la version

4. (Avec vos mots) Qu'est-ce qu'un IDE ?

5. (Vrai/Faux) IntelliJ IDEA Community Edition est payant.

💡 **Réponses**

1-b · 2-Vrai · 3-b · 4 : un logiciel tout-en-un pour écrire, lancer et déboguer du code · 5-Faux (il est gratuit ; c'est « Ultimate » qui est payant).

Chapitre 3 — Votre premier programme Java

🎓 Objectifs d'apprentissage

- écrire, comprendre et exécuter le programme « Hello World » ;
- reconnaître le rôle de chaque mot : `public`, `class`, `static`, `void`, `main` … ;
- utiliser `System.out.println` pour afficher du texte ;
- comprendre le rôle des accolades `{ }` et du point-virgule `;`.

3.1 Le programme « Hello World »

Par tradition, le tout premier programme que l'on écrit dans n'importe quel langage affiche simplement un message. Voici le nôtre. Recopiez-le dans votre classe `Main` :

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Bonjour Java !");
    }
}
```

Lancez-le (flèche verte ▶). Dans la console, vous devez voir :

```
CONSOLE
Bonjour Java !
```

💡 Conseil — ne mémorisez pas tout de suite

Ce petit code contient beaucoup de mots nouveaux. **Vous n'avez pas à les retenir aujourd'hui.** Pour l'instant, recopiez-les mécaniquement : c'est la « formule magique » de départ de tout programme Java. Nous allons quand même expliquer chaque mot, pour lever le mystère.

3.2 Le code expliqué, mot par mot

Élément	Rôle expliqué simplement
`public`	Signifie « accessible de partout ». C'est un *niveau de visibilité*. Ici, on rend la classe et la méthode utilisables par la machine qui lance le programme.
`class`	Mot-clé qui déclare une **classe**. En Java, tout code vit à l'intérieur d'une classe. Voyez-la comme la « boîte » qui contient votre programme.
`Main`	Le **nom** que nous avons donné à la classe. Il doit correspondre au nom du fichier (`Main.java`). Par convention, un nom de classe commence par une majuscule.
`static`	Signifie que la méthode `main` appartient à la classe elle-même et peut être lancée sans créer d'objet. Nous comprendrons pleinement ce mot au chapitre 13. Pour `main`, il est obligatoire.
`void`	Signifie « ne renvoie rien ». La méthode `main` exécute des actions mais ne rend aucune valeur. (Chapitre 10.)
`main`	Le nom de la méthode de **démarrage**. C'est *le point d'entrée* : quand vous lancez le programme, la JVM cherche `main` et exécute ce qu'il y a dedans. Sans elle, rien ne démarre.
`String[] args`	Les **arguments** que l'on peut passer au programme au lancement (une liste de mots). On ne s'en sert pas au début, mais cette partie doit être présente pour que `main` soit valide.
`System.out.println(...)`	L'instruction qui **affiche du texte** dans la console, puis passe à la ligne. `System.out` représente la sortie standard (l'écran) ; `println` = « print line » = « afficher une ligne ».

📖 **Vocabulaire — Méthode et point d'entrée**

Une **méthode** est un bloc d'instructions qui porte un nom (ici `main`). Le **point d'entrée** est la méthode par laquelle le programme commence. En Java, c'est toujours `main` .

3.3 Explication ligne par ligne

Code	Ce qu'il se passe
`public class Main {`	« Je crée une classe nommée Main. » L'accolade `{` ouvre son contenu.
`public static void main(String[] args) {`	« Voici la méthode de démarrage. » L'accolade ouvre son contenu. Tout ce qui est entre ses accolades s'exécute au lancement.
`System.out.println("Bonjour Java !");`	« Affiche le texte *Bonjour Java !* puis va à la ligne. » Le texte entre `"guillemets"` s'appelle une **chaîne de caractères**. Le `;` termine l'instruction.
`}`	Ferme la méthode `main`.
`}`	Ferme la classe `Main`.

📌 **À retenir — les deux signes de ponctuation clés**

- Les **accolades** `{ }` délimitent un bloc (le contenu d'une classe, d'une méthode…). Elles vont toujours par paire.
- Le **point-virgule** `;` termine une *instruction* (une action). Il se met à la fin de chaque ligne d'action, comme le point à la fin d'une phrase.

3.4 Afficher plusieurs lignes : `println` vs `print`

```
System.out.println("Ligne 1");   // affiche puis passe à la ligne
System.out.println("Ligne 2");
System.out.print("A");            // affiche SANS passer à la ligne
System.out.print("B");
```

```
CONSOLE
Ligne 1
Ligne 2
AB
```

`println` = affiche *et* passe à la ligne. `print` = affiche *sans* passer à la ligne.

3.5 Erreurs fréquentes du débutant

🐞 **Erreur fréquente — le point-virgule oublié**

`System.out.println("Salut")` sans `;` → erreur « ';' *expected* ». Ajoutez le point-virgule à la fin.

🐞 **Erreur fréquente — les guillemets manquants**

Le texte à afficher doit être entre guillemets doubles : `"Bonjour"`. Sans eux, Java croit que `Bonjour` est le nom d'une variable et se plaint de ne pas la connaître.

🐞 **Erreur fréquente — accolade non fermée**

Chaque `{` doit avoir son `}`. Un oubli déclenche « *reached end of file while parsing* ». IntelliJ colore les accolades appariées : cliquez à côté d'une accolade pour voir sa jumelle.

🐞 **Erreur fréquente — `Println` ou `PrintLn`**

Java respecte les majuscules/minuscules (la *casse*). Ce doit être exactement `println`, tout en minuscules. `System`, en revanche, prend une majuscule.

3.6 Exercice

Exercice 3.1 NIVEAU 1

Objectif : vous approprier `println`.

Connaissances nécessaires : structure d'un programme, `System.out.println`.

Consignes : écrivez un programme qui affiche votre prénom sur la première ligne, votre ville sur la deuxième, et « Je débute en Java ! » sur la troisième.

Résultat attendu (exemple) :

```
Jonathan
Genève
Je débute en Java !
```

Indice : il faut trois instructions `System.out.println(...)`, chacune terminée par un `;`.

Correction de l'exercice 3.1

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Jonathan");
        System.out.println("Genève");
        System.out.println("Je débute en Java !");
    }
}
```

Explication : chaque appel à `println` affiche une chaîne puis passe à la ligne, ce qui empile les trois textes l'un sous l'autre. Les guillemets délimitent le texte ; le point-virgule termine chaque instruction.

Quiz — Chapitre 3

1. (QCM) Quel est le point d'entrée d'un programme Java ? a) `class` b) `main` c) `System`

2. (Complétez) `System.out._____("Salut");` pour afficher et passer à la ligne.

3. (Vrai/Faux) Le point-virgule termine une instruction.

4. (Trouvez l'erreur) `System.out.println(Bonjour);`

5. (Prédire) Qu'affiche `print("A"); print("B");` ?

💡 **Réponses**

1-b · 2 : `println` · 3-Vrai · 4 : `Bonjour` n'est pas entre guillemets → il faut `"Bonjour"` · 5 : `AB` sur la même ligne.

Chapitre 4 — Les commentaires

🎓 **Objectifs d'apprentissage**

- écrire des commentaires sur une ligne, sur plusieurs lignes, et de documentation ;
- savoir quand un commentaire est utile… et quand il est inutile.

4.1 À quoi servent les commentaires ?

Un **commentaire** est un texte que vous écrivez *pour vous ou pour d'autres humains*, et que Java **ignore complètement** à l'exécution. Il sert à expliquer, rappeler, ou désactiver temporairement du code.

Analogie : ce sont les notes que l'on écrit dans la marge d'un cahier de recette (« attention, four déjà chaud ») : elles aident le lecteur mais ne font pas partie du plat.

4.2 Les trois types de commentaires

```
// Ceci est un commentaire sur une seule ligne.
System.out.println("Salut"); // on peut aussi commenter en fin de ligne

/* Ceci est un commentaire
   sur plusieurs lignes.
   Pratique pour une explication longue. */

/**
 * Ceci est un commentaire de DOCUMENTATION (Javadoc).
 * Il décrit ce que fait une classe ou une méthode.
 * @param nom le prénom de l'utilisateur
 */
```

Type	Syntaxe	Usage
Une ligne	`// texte`	Petite note rapide
Plusieurs lignes	`/* texte */`	Explication longue, ou désactiver un bloc
Documentation	`/** texte */`	Décrire une classe/méthode ; outils et IDE l'affichent

4.3 Bons et mauvais commentaires

✅ Bonne pratique

Un bon commentaire explique le **pourquoi**, pas le *quoi*. Le code dit déjà *ce qu'il fait* ; le commentaire dit *pourquoi* on le fait ainsi.

```
// Taux de TVA suisse standard en 2026
double tva = 0.081;
```

🐞 Erreur fréquente — le commentaire inutile

Ne paraphrasez pas le code. Ceci n'apporte rien :

```
int age = 18; // on met 18 dans age  ← inutile, on le voit déjà
```

💡 Conseil — désactiver du code

Pendant vos tests, vous pouvez « commenter » une ligne pour la désactiver sans la supprimer. Dans IntelliJ : sélectionnez la ligne et pressez **Ctrl + /** (Cmd + / sur Mac).

Exercice & correction

Exercice 4.1 NIVEAU 1

Consignes : reprenez votre programme du chapitre 3 et ajoutez : un commentaire d'une ligne au-dessus de la classe indiquant son but, et un commentaire de fin de ligne sur l'un des `println`.

```java
// Programme de présentation personnelle
public class Main {
    public static void main(String[] args) {
        System.out.println("Jonathan"); // mon prénom
        System.out.println("Genève");
    }
}
```

Explication : les deux commentaires sont ignorés par Java ; le résultat affiché reste identique. Ils servent uniquement au lecteur.

Quiz — Chapitre 4

1. (Vrai/Faux) Un commentaire est exécuté par Java.

2. (QCM) Quelle syntaxe pour un commentaire sur plusieurs lignes ? a) `// ... //` b) `/* ... */` c) `<!-- ... -->`

3. (Avec vos mots) Un bon commentaire explique le « quoi » ou le « pourquoi » ?

💡 **Réponses**

1-Faux (il est ignoré) · 2-b · 3 : le **pourquoi**.

Chapitre 5 — Les variables et les types

🎓 **Objectifs d'apprentissage**

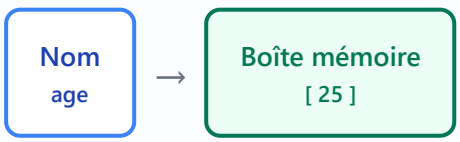
- comprendre ce qu'est une variable, la déclarer et l'initialiser ;
- modifier la valeur d'une variable ;
- connaître les principaux types : `int`, `long`, `double`, `float`, `boolean`, `char`, `String` ;
- nommer correctement une variable et créer une constante avec `final` ;
- distinguer types primitifs et objets.

5.1 Qu'est-ce qu'une variable ?

Une **variable** est une boîte nommée dans la mémoire de l'ordinateur, dans laquelle on range une valeur pour la réutiliser plus tard.

📖 **Vocabulaire — Variable**

Ce que c'est : un espace mémoire nommé qui contient une valeur.
À quoi ça sert : mémoriser et réutiliser une information (un âge, un prix, un prénom).
Pourquoi c'est utile : sans variables, impossible de garder la moindre donnée pendant l'exécution.
Exemple : `int age = 25;`
Analogie : une variable est comme un tiroir étiqueté. L'étiquette est le nom (`age`), le contenu est la valeur (`25`). On peut ouvrir le tiroir pour lire ou changer ce qu'il contient.



Une variable relie un **nom** à une **valeur** rangée en mémoire.

5.2 Déclarer, initialiser, modifier

Trois gestes à distinguer :

```
int age;              // 1) DÉCLARATION : je crée la boîte "age" de type int
age = 25;             // 2) INITIALISATION : je range 25 dedans
age = 26;             // 3) MODIFICATION : je remplace par 26

int annee = 2026;      // déclaration + initialisation en une seule ligne (courant)
```

📌 **À retenir — l'anatomie d'une déclaration**

`int age = 25;` se lit : **type** (`int`) + **nom** (`age`) + `=` + **valeur** (`25`) + `;` . Le `=` signifie « reçoit la valeur », pas « égal » au sens mathématique.

🐞 **Erreur fréquente — utiliser une variable non initialisée**

Si vous déclarez `int age;` et tentez de l'afficher sans lui donner de valeur, Java refuse : « *variable age might not have been initialized* ». Donnez toujours une valeur avant d'utiliser une variable.

5.3 Les types de base

Le **type** indique la nature de la valeur (nombre entier, nombre à virgule, texte…). Il détermine ce qu'on peut ranger dans la boîte et ce qu'on peut en faire.

Tableau comparatif des principaux types

Type	Contient	Exemple	À utiliser pour
`int`	Nombre entier (≈ ±2 milliards)	`int n = 42;`	Âge, quantité, compteur
`long`	Très grand entier	`long p = 9000000000L;`	Grandes valeurs (population mondiale, millisecondes)
`double`	Nombre à virgule (précis)	`double prix = 9.99;`	Prix, mesures, moyennes
`float`	Nombre à virgule (moins précis)	`float f = 1.5f;`	Rare pour un débutant ; préférez `double`
`boolean`	`true` ou `false`	`boolean ok = true;`	Vrai/faux, oui/non, actif/inactif
`char`	Un seul caractère	`char lettre = 'A';`	Une lettre, un symbole
`String`	Une chaîne de texte	`String nom = "Jonathan";`	Mots, phrases, prénoms

⚠️ **Attention — guillemets simples ou doubles ?**

- `char` : **un seul** caractère entre **apostrophes** → `'A'`.
- `String` : du texte entre **guillemets doubles** → `"Bonjour"`.

`'A'` (char) et `"A"` (String) ne sont pas la même chose pour Java !

⚠️ **Attention — les suffixes `L` et `f`**

Un `long` supérieur à 2 milliards s'écrit avec un `L` à la fin : `9000000000L`. Un `float` s'écrit avec un `f` : `1.5f`. Sans ces suffixes, Java interprète mal la valeur.

Un exemple complet

```java
public class Main {
    public static void main(String[] args) {
        String prenom = "Jonathan";
        int age = 30;
        double taille = 1.78;
        boolean majeur = true;
        char initiale = 'J';

        System.out.println("Prénom : " + prenom);
        System.out.println("Âge : " + age);
        System.out.println("Taille : " + taille + " m");
        System.out.println("Majeur : " + majeur);
        System.out.println("Initiale : " + initiale);
    }
}
```

```
CONSOLE
Prénom : Jonathan
Âge : 30
Taille : 1.78 m
Majeur : true
Initiale : J
```

Explication ligne par ligne : chaque ligne déclare une variable d'un type adapté à sa donnée. Le signe `+`, ici, **colle du texte** (on appelle cela la *concaténation*, chapitre 6). On affiche une étiquette suivie de la valeur.

5.4 Bien nommer ses variables

✅ **Bonne pratique — conventions de nommage**

- Un nom **parlant** : `ageUtilisateur` plutôt que `a` ou `x`.
- **camelCase** : première lettre minuscule, puis une majuscule à chaque nouveau mot : `nombreDeTentatives`.
- Pas d'espace, pas d'accent, pas de tiret. Les lettres, chiffres et `_` sont permis, mais un nom ne commence jamais par un chiffre.
- Interdits : les mots réservés (`int`, `class`, `public`…).

🐞 **Erreur fréquente — noms invalides**

`int 2age;` (commence par un chiffre), `int mon age;` (espace), `int prix€;` (symbole) → tous refusés.

5.5 Les constantes avec `final`

Parfois, une valeur ne doit **jamais** changer (une TVA, le nombre PI, un plafond). On la déclare avec `final` : c'est une **constante**.

```
final double TVA = 0.081;
final int AGE_MAJORITE = 18;

TVA = 0.1; // ❌ ERREUR : impossible de modifier une constante
```

✅ **Bonne pratique**

Par convention, les constantes s'écrivent en **MAJUSCULES** avec des `_` entre les mots : `AGE_MAJORITE`. Cela signale immédiatement « valeur fixe ».

5.6 Types primitifs vs objets (survol)

📖 **Vocabulaire — Type primitif vs objet**

Les **types primitifs** (`int` , `double` , `boolean` , `char` , `long` , `float` ...) sont les briques de base : ils rangent directement une valeur simple, s'écrivent en minuscules.

`String` , lui, commence par une majuscule : c'est un **objet** (une valeur plus riche, dotée de « capacités » — des méthodes comme `.length()`). Nous approfondirons les objets au chapitre 13. Pour l'instant, retenez simplement : *minuscule = primitif, Majuscule = objet.*

Exercices — Chapitre 5

Exercice 5.1 NIVEAU 1

Objectif : déclarer des variables de bons types.

Consignes : créez des variables pour : un nom (texte), un nombre de frères et sœurs (entier), une note sur 20 (à virgule), et « aime le café » (vrai/faux). Affichez-les.

Exercice 5.2 NIVEAU 2

Consignes : déclarez une constante `PRIX_UNITAIRE = 4.5` et une variable `quantite = 3`. Calculez et affichez le total.

Exercice 5.3 NIVEAU 3 — DÉFI

Consignes : vous avez `int a = 5;` et `int b = 8;`. Sans utiliser de troisième variable « temporaire » évidente… trouvez tout de même un moyen d'**échanger** leurs valeurs (astuce : une variable temporaire *est* la solution la plus claire — implémentez-la proprement).

Corrections — Chapitre 5

5.1 —

```
String nom = "Léa";
int fratrie = 2;
double note = 14.5;
boolean aimeCafe = true;
System.out.println(nom + ", " + fratrie +  " frères/sœurs, note " + note + ", 
```

On choisit le type adapté à chaque donnée : texte → `String`, entier → `int`, virgule → `double`, oui/non → `boolean`.

5.2 —

```
final double PRIX_UNITAIRE = 4.5;
int quantite = 3;
double total = PRIX_UNITAIRE * quantite;
System.out.println("Total : " + total + " €"); // Total : 13.5 €
```

Le prix est fixe → `final`. Le total est un calcul (`*` = multiplication).

5.3 —

```
int a = 5, b = 8;
int temp = a;  // on met de côté a (5)
a = b;         // a devient 8
b = temp;      // b devient l'ancien a, soit 5
System.out.println("a=" + a + " b=" + b); // a=8 b=5
```

La variable temporaire évite d'écraser une valeur avant de l'avoir copiée. C'est le geste propre et lisible — à préférer aux astuces obscures.

Quiz — Chapitre 5

1. (QCM) Quel type pour un prix comme 9.99 ? a) `int` b) `double` c) `boolean`

2. (Vrai/Faux) `'A'` et `"A"` sont identiques pour Java.

3. (QCM) Que fait `final` ? a) termine le programme b) empêche de modifier la variable c) affiche la variable

4. (Trouvez l'erreur) `int 3notes = 15;`

5. (Complétez) Pour un vrai/faux, on utilise le type `_____` .

6. (Prédire) Qu'affiche `int x = 7; x = 10; System.out.println(x);` ?

💡 **Réponses**

1-b · 2-Faux (`char` vs `String`) · 3-b · 4 : un nom ne peut pas commencer par un chiffre → `notes3` · 5 : `boolean` · 6 : `10` .

Chapitre 6 — Les opérateurs

🎓 Objectifs d'apprentissage

- faire des calculs : `+ - * / %` ;
- incrémenter et décrémenter (`++` , `--`) ;
- comparer des valeurs et combiner des conditions (`&& || !`) ;
- ne plus jamais confondre `=` , `==` et `!=` .

📖 Vocabulaire — Opérateur

Un **opérateur** est un symbole qui effectue une opération sur des valeurs : addition, comparaison, etc. `3 + 5` utilise l'opérateur `+` .

6.1 Les opérateurs arithmétiques

Opérateur	Rôle	Exemple	Résultat
`+`	Addition	`7 + 3`	10
`-`	Soustraction	`7 - 3`	4
`*`	Multiplication	`7 * 3`	21
`/`	Division	`7 / 2`	3 (voir piège)
`%`	Modulo (reste de la division)	`7 % 2`	1

⚠️ **Attention — le grand piège de la division entière**

`7 / 2` donne **3** et non 3.5 ! Quand on divise deux `int`, Java fait une division *entière* et jette la partie après la virgule. Pour obtenir 3.5, il faut au moins un nombre à virgule :

```
int a = 7, b = 2;
System.out.println(a / b);        // 3   (division entière)
System.out.println(7.0 / 2);      // 3.5 (un double suffit)
System.out.println((double) a / b); // 3.5 (on "convertit" a en double)
```

📖 **Vocabulaire — Modulo**

Le **modulo** (`%`) donne le *reste* d'une division. `17 % 5 = 2` (car 17 = 3×5 + 2). Très utile pour savoir si un nombre est pair (`n % 2 == 0`) ou pour « boucler » (heures, jours…).

6.2 Incrémentation et décrémentation

```
int compteur = 5;
compteur++;   // ajoute 1  → 6   (équivaut à compteur = compteur + 1)
compteur--;   // enlève 1  → 5
compteur += 10; // ajoute 10 → 15  (raccourci de compteur = compteur + 10)
compteur -= 3;  // enlève 3  → 12
compteur *= 2;  // double    → 24
```

Ces raccourcis sont omniprésents, surtout dans les boucles (chapitre 9).

6.3 Les opérateurs d'affectation et de comparaison

📌 **À retenir — la différence CAPITALE entre `=`, `==` et `!=`**

Symbole	Signification	Exemple
`=`	**Affectation** : « range la valeur dans »	`age = 18;` (met 18 dans age)
`==`	**Comparaison d'égalité** : « est-il égal à ? » (renvoie true/false)	`age == 18` (est-ce que age vaut 18 ?)
`!=`	**Différent de** : « est-il différent de ? »	`age != 18`

🐞 **Erreur fréquente — `=` au lieu de `==` dans une condition**

Écrire `if (age = 18)` est une erreur classique : c'est une affectation, pas un test. Il faut `if (age == 18)`. Bonne nouvelle : en Java, ce cas précis provoque souvent une erreur de compilation, ce qui vous protège.

Comparaison	Signifie
`a == b`	a égal à b ?
`a != b`	a différent de b ?
`a > b`	a strictement supérieur à b ?
`a < b`	a strictement inférieur à b ?
`a >= b`	a supérieur ou égal à b ?
`a <= b`	a inférieur ou égal à b ?

Une comparaison renvoie toujours un `boolean` (`true` ou `false`).

6.4 Les opérateurs logiques : `&&`, `||`, `!`

Ils permettent de **combiner** plusieurs conditions.

Opérateur	Nom	Vrai quand…	Analogie
`&&`	ET	les **deux** conditions sont vraies	« majeur ET permis » pour conduire
`||`	OU	**au moins une** est vraie	« carte OU espèces » pour payer
`!`	NON	on **inverse** la condition	« NON connecté » = déconnecté

```java
int age = 20;
boolean aPermis = true;

System.out.println(age >= 18 && aPermis);  // true  (les deux sont vrais)
System.out.println(age < 18 || aPermis);   // true  (au moins un vrai)
System.out.println(!aPermis);                // false (inverse de true)
```

> 💡 **Conseil — table de vérité minute**
>
> `true && false` → false. `true || false` → true. `!true` → false. Retenez : ET exige tout le monde, OU se contente d'un seul, NON retourne la réponse.

Exercices — Chapitre 6

Exercice 6.1 NIVEAU 1

Consignes : déclarez `a = 17` et `b = 5`. Affichez leur somme, différence, produit, division entière et modulo.

Exercice 6.2 NIVEAU 2

Consignes : déclarez un entier `n = 24`. Affichez `true` ou `false` selon que `n` est pair, à l'aide du modulo.

Exercice 6.3 NIVEAU 3 — DÉFI

Consignes : une personne peut voter si elle a au moins 18 ans **et** est inscrite sur les listes. Avec `int age = 17;` et `boolean inscrit = true;`, affichez si elle peut voter.

Corrections — Chapitre 6

6.1 —

```
int a = 17, b = 5;
System.out.println(a + b); // 22
System.out.println(a - b); // 12
System.out.println(a * b); // 85
System.out.println(a / b); // 3  (division entière)
System.out.println(a % b); // 2  (reste)
```

6.2 —

```
int n = 24;
System.out.println(n % 2 == 0); // true : un nombre pair a un reste de 0 dans l
```

6.3 —

```
int age = 17;
boolean inscrit = true;
System.out.println(age >= 18 && inscrit); // false : la 1re condition échoue, c
```

Avec `&&`, il suffit qu'*une* condition soit fausse pour que le tout soit faux.

Quiz — Chapitre 6

1. (Prédire) Que vaut `9 / 2` (deux int) ? Et `9 % 2` ?
2. (QCM) Quel opérateur teste l'égalité ? a) `=` b) `==` c) `=>`
3. (Prédire) `true && false` vaut ? `true || false` vaut ?
4. (Vrai/Faux) `x += 3` équivaut à `x = x + 3`.
5. (Complétez) Pour tester si `note` vaut 20 : `if (note ___ 20)`.

Réponses

1 : `9/2 = 4` et `9%2 = 1` · 2-b · 3 : `false` puis `true` · 4-Vrai · 5 : `==`.

Chapitre 7 — Les entrées utilisateur (Scanner)

🎓 **Objectifs d'apprentissage**

- demander une information à l'utilisateur avec la classe `Scanner` ;
- lire du texte (`nextLine`) et des nombres (`nextInt` , `nextDouble`) ;
- comprendre et éviter le piège `nextInt()` / `nextLine()` ;
- fermer le scanner et valider simplement une saisie.

7.1 Lire une saisie avec `Scanner`

Jusqu'ici nos programmes affichaient du texte. Pour les rendre *interactifs*, on utilise **Scanner**, un outil livré avec Java qui lit ce que l'utilisateur tape au clavier.

📖 **Vocabulaire — import**

`import java.util.Scanner;` indique à Java d'aller chercher l'outil `Scanner` dans sa bibliothèque. On écrit les `import` tout en haut du fichier, avant la classe.
Analogie : c'est comme sortir un outil de la boîte avant de s'en servir.

```
import java.util.Scanner;   // 1) on importe l'outil (tout en haut)

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // 2) on crée le scanner (lit le clavie

        System.out.print("Quel est ton prénom ? ");
        String prenom = sc.nextLine();              // 3) on lit une ligne de texte

        System.out.println("Bonjour " + prenom + " !");

        sc.close();                                 // 4) on ferme le scanner
    }
}
```

```
CONSOLE (L'UTILISATEUR TAPE « JONATHAN »)
Quel est ton prénom ? Jonathan
Bonjour Jonathan !
```

Explication ligne par ligne :

`import ... Scanner;`	rend l'outil `Scanner` disponible.
`new Scanner(System.in)`	crée un scanner branché sur `System.in` (le clavier).
`sc.nextLine()`	attend que l'utilisateur tape une ligne + Entrée, et renvoie ce texte.
`sc.close()`	libère le scanner quand on n'en a plus besoin.

7.2 Lire des nombres

Méthode	Lit…	Exemple de saisie
`nextLine()`	une ligne de texte (String)	`Jonathan Dupont`
`nextInt()`	un nombre entier	`30`
`nextDouble()`	un nombre à virgule	`1.78`

⚠️ **Attention — la virgule décimale**

Selon la configuration de votre système, `nextDouble()` peut attendre `1,78` (virgule) plutôt que `1.78` (point). Si vous obtenez une erreur, essayez l'autre séparateur.

7.3 LE piège à connaître : `nextInt()` puis `nextLine()`

🐞 **Erreur fréquente — la ligne « sautée »**

Quand on fait `nextInt()` puis `nextLine()`, la deuxième lecture semble « avalée » : le programme ne s'arrête pas pour vous laisser taper.

Pourquoi ? `nextInt()` lit le nombre mais **laisse le « Entrée »** (le retour à la ligne) dans le tampon. Le `nextLine()` suivant lit alors cette ligne vide restante et passe son tour.

La solution : ajouter un `sc.nextLine();` « vide » pour consommer le retour à la ligne après un `nextInt()` / `nextDouble()`.

```
System.out.print("Ton âge ? ");
int age = sc.nextInt();
sc.nextLine();   // ✅ on "avale" le retour à la ligne laissé par nextInt()

System.out.print("Ta ville ? ");
String ville = sc.nextLine(); // maintenant ceci fonctionne
```

7.4 Programme complet : présentation personnalisée

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Quel est ton prénom ? ");
        String prenom = sc.nextLine();

        System.out.print("Quel est ton âge ? ");
        int age = sc.nextInt();
        sc.nextLine(); // on avale le retour à la ligne

        System.out.print("Dans quelle ville habites-tu ? ");
        String ville = sc.nextLine();

        int ageDansDixAns = age + 10;

        System.out.println("---");
        System.out.println("Salut " + prenom + " ! Tu habites à " + ville + ".");
        System.out.println("Tu as " + age + " ans, tu en auras " + ageDansDixAns + "

        sc.close();
    }
}
```

```
CONSOLE (EXEMPLE)
Quel est ton prénom ? Jonathan
Quel est ton âge ? 30
Dans quelle ville habites-tu ? Genève
---
```

```
Salut Jonathan ! Tu habites à Genève.
Tu as 30 ans, tu en auras 40 dans 10 ans.
```

7.5 Valider simplement une saisie

Un utilisateur peut se tromper. `hasNextInt()` permet de vérifier *avant* de lire si ce qui a été tapé est bien un entier :

```java
System.out.print("Entre un entier : ");
if (sc.hasNextInt()) {
    int n = sc.nextInt();
    System.out.println("Merci, tu as tapé " + n);
} else {
    System.out.println("Ce n'est pas un entier valide.");
}
```

(Nous verrons une validation plus robuste avec les boucles au chapitre 9 et les exceptions au chapitre 16.)

> ✅ **Bonne pratique**
>
> Un seul `Scanner` par programme suffit, créé une fois au début et fermé à la fin. Ne créez pas plusieurs scanners sur `System.in`.

Exercices — Chapitre 7

Exercice 7.1 NIVEAU 1

Consignes : demandez le prénom et le pays de l'utilisateur, puis affichez « Bonjour [prénom], bienvenue depuis [pays] ! ».

Exercice 7.2 NIVEAU 2

Consignes : demandez deux nombres entiers et affichez leur somme.
Entrée exemple : 8 puis 5 — **Sortie exemple :** `La somme est 13`

Exercice 7.3 NIVEAU 3 — DÉFI

Consignes : demandez un prénom (texte) puis une année de naissance (entier), et affichez « [prénom], tu as environ [âge] ans. » (on est en 2026). Gérez correctement le piège `nextInt/nextLine`.

Corrections — Chapitre 7

7.1 —

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Prénom : ");
        String prenom = sc.nextLine();
        System.out.print("Pays : ");
        String pays = sc.nextLine();
        System.out.println("Bonjour " + prenom + ", bienvenue depuis " + pays +
        sc.close();
    }
}
```

7.2 —

```
int a = sc.nextInt();
int b = sc.nextInt();
System.out.println("La somme est " + (a + b));
```

Notez les parenthèses autour de `(a + b)` : sans elles, Java collerait les chiffres au texte au lieu de les additionner (voir encadré ci-dessous).

7.3 —

```
System.out.print("Prénom : ");
String prenom = sc.nextLine();
System.out.print("Année de naissance : ");
int annee = sc.nextInt();
int age = 2026 - annee;
System.out.println(prenom + ", tu as environ " + age + " ans.");
```

Ici `nextLine()` vient *avant* le `nextInt()` : pas de piège. Si l'ordre était inversé, il faudrait un `sc.nextLine();` intercalé.

🐞 **Erreur fréquente — `+` qui colle au lieu d'additionner**

`"Somme : " + a + b` avec a=8, b=5 affiche `Somme : 85` (Java colle) et non 13 ! Dès qu'un `String` est présent, `+` devient de la concaténation, de gauche à droite. Mettez le calcul entre parenthèses : `"Somme : " + (a + b)` → `Somme : 13`.

Quiz — Chapitre 7

1. (QCM) Quelle méthode lit un entier ? a) `nextLine()` b) `nextInt()` c) `readInt()`

2. (Vrai/Faux) Il faut importer `Scanner` avant de l'utiliser.

3. (Complétez) Après `sc.nextInt()`, on ajoute souvent `sc._____();` pour vider le tampon.

4. (Prédire) Avec a=3, b=4 : qu'affiche `"R:" + a + b` ? Et `"R:" + (a + b)` ?

💡 **Réponses**

1-b · 2-Vrai · 3 : `nextLine` · 4 : `R:34` puis `R:7`.

📌 📊 **Évaluation intermédiaire n°1 — après les bases**

Vous avez terminé les fondations (chapitres 1 à 7) ! Une **évaluation complète « après les bases »** vous attend au chapitre 23. Faites-la une fois les conditions et boucles vues, pour valider vos acquis.

Chapitre 8 — Les conditions

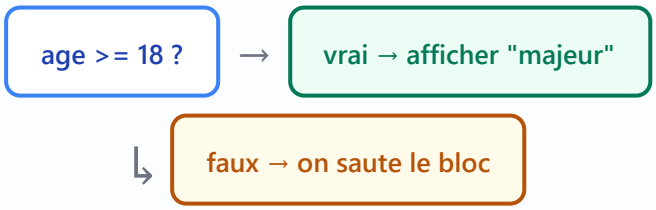
> 🎓 **Objectifs d'apprentissage**
>
> - faire prendre des décisions à un programme avec `if`, `else`, `else if` ;
> - imbriquer et combiner des conditions ;
> - utiliser `switch` pour choisir parmi plusieurs cas ;
> - écrire des cas concrets : majorité, mot de passe, appréciation, menu.

8.1 La condition `if`

Une **condition** permet d'exécuter du code *seulement si* quelque chose est vrai. **Analogie :** « *Si* il pleut, je prends un parapluie. »

```
int age = 20;

if (age >= 18) {
    System.out.println("Tu es majeur.");
}
```

Entre parenthèses : une condition qui vaut `true` ou `false`. Si elle est vraie, le bloc entre accolades s'exécute ; sinon, il est ignoré.



8.2 `if ... else`

`else` (« sinon ») gère le cas contraire :

```java
int age = 15;

if (age >= 18) {
    System.out.println("Tu es majeur.");
} else {
    System.out.println("Tu es mineur.");
}
```

```
CONSOLE
Tu es mineur.
```

8.3 `else if` : plusieurs cas

Pour tester plusieurs possibilités à la suite, on enchaîne des `else if`. Exemple : attribuer une appréciation à une note sur 20.

```java
int note = 14;

if (note >= 16) {
    System.out.println("Très bien");
} else if (note >= 14) {
    System.out.println("Bien");
} else if (note >= 10) {
    System.out.println("Passable");
} else {
    System.out.println("Insuffisant");
}
```

```
CONSOLE
Bien
```

> ⚠️ **Attention — l'ordre compte**
>
> Java teste les conditions **de haut en bas** et s'arrête à la première vraie. Si vous mettiez `note >= 10` en premier, une note de 18 afficherait « Passable » à tort. Allez du cas le plus exigeant au moins exigeant.

8.4 Conditions combinées et imbriquées

Combinées avec `&&` / `||` (chapitre 6) :

```
int age = 25;
boolean aPermis = true;

if (age >= 18 && aPermis) {
    System.out.println("Tu peux conduire.");
}
```

Imbriquées (un `if` dans un `if`) :

```
if (age >= 18) {
    if (aPermis) {
        System.out.println("Majeur et titulaire du permis.");
    } else {
        System.out.println("Majeur mais sans permis.");
    }
}
```

> 💡 **Conseil**
>
> Quand c'est possible, préférez des conditions combinées (`&&`) à des imbrications profondes : c'est plus lisible.

8.5 Vérifier un mot de passe

```java
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final String MOT_DE_PASSE = "java2026";

        System.out.print("Mot de passe : ");
        String saisie = sc.nextLine();

        if (saisie.equals(MOT_DE_PASSE)) {
            System.out.println("Accès autorisé ✅");
        } else {
            System.out.println("Accès refusé ❌");
        }
        sc.close();
    }
}
```

> ⚠️ **Attention — comparer du texte avec `.equals()`, pas `==`**
>
> Pour comparer deux `String`, on utilise `saisie.equals(MOT_DE_PASSE)` et **jamais** `==`. La raison (technique) est expliquée en détail au chapitre 12. Pour l'instant, retenez la règle : *texte* → `.equals()`.

8.6 Le `switch` : choisir parmi plusieurs valeurs

Quand on compare une même variable à plusieurs valeurs précises, `switch` est plus clair qu'une longue série de `else if`. Exemple : un menu.

```java
int choix = 2;

switch (choix) {
    case 1:
        System.out.println("Nouvelle partie");
        break;
    case 2:
        System.out.println("Charger une partie");
        break;
    case 3:
        System.out.println("Quitter");
        break;
    default:
        System.out.println("Choix invalide");
}
```

```
CONSOLE
Charger une partie
```

🐞 **Erreur fréquente — oublier `break`**

Sans `break`, l'exécution « déborde » sur le cas suivant. `break` dit « j'ai trouvé mon cas, je sors du switch ». `default` gère toutes les autres valeurs.

💡 **Conseil — la syntaxe moderne (Java 14+)**

Java propose une écriture plus courte avec `->`, sans `break` :

```java
switch (choix) {
    case 1 -> System.out.println("Nouvelle partie");
    case 2 -> System.out.println("Charger");
    default -> System.out.println("Invalide");
}
```

Exercices — Chapitre 8

Exercice 8.1 NIVEAU 1

Consignes : demandez un âge et affichez « Majeur » ou « Mineur ».

Exercice 8.2 NIVEAU 2

Consignes : demandez une note sur 20 et affichez : « Excellent » (≥16), « Bien » (≥12), « Moyen » (≥10), sinon « À revoir ».

Exercice 8.3 NIVEAU 2

Consignes : demandez un nombre et indiquez s'il est positif, négatif ou nul.

Exercice 8.4 NIVEAU 3 — DÉFI

Consignes : avec un `switch`, affichez le nom du jour selon un numéro de 1 à 7 (1 = Lundi …). Gérez le cas invalide avec `default`.

Corrections — Chapitre 8

8.1 —

```
int age = sc.nextInt();
if (age >= 18) System.out.println("Majeur");
else System.out.println("Mineur");
```

(Quand un bloc ne contient qu'une instruction, les accolades sont facultatives — mais les garder reste une bonne habitude.)

8.2 —

```
int note = sc.nextInt();
if (note >= 16) System.out.println("Excellent");
else if (note >= 12) System.out.println("Bien");
else if (note >= 10) System.out.println("Moyen");
else System.out.println("À revoir");
```

On teste du plus exigeant au moins exigeant, pour que le bon message sorte.

8.3 —

```
int n = sc.nextInt();
if (n > 0) System.out.println("Positif");
else if (n < 0) System.out.println("Négatif");
else System.out.println("Nul");
```

8.4 —

```
int jour = sc.nextInt();
switch (jour) {
    case 1 -> System.out.println("Lundi");
    case 2 -> System.out.println("Mardi");
    case 3 -> System.out.println("Mercredi");
    case 4 -> System.out.println("Jeudi");
    case 5 -> System.out.println("Vendredi");
    case 6 -> System.out.println("Samedi");
    case 7 -> System.out.println("Dimanche");
    default -> System.out.println("Numéro invalide");
}
```

Quiz — Chapitre 8

1. (QCM) `else` s'exécute quand… a) la condition du `if` est vraie b) elle est fausse c) toujours

2. (Vrai/Faux) On compare deux `String` avec `==`.

3. (Trouvez l'erreur) Un `switch` où chaque `case` classique n'a pas de `break`.

4. (Prédire) Avec `note = 11`, que donne l'exercice 8.2 ?

5. (Complétez) Le mot-clé pour « toutes les autres valeurs » dans un switch est `_____`.

💡 **Réponses**

1-b · 2-Faux (on utilise `.equals()`) · 3 : l'exécution déborde sur les cas suivants → il manque `break` · 4 : `Moyen` · 5 : `default`.

Chapitre 9 — Les boucles

> 🎓 **Objectifs d'apprentissage**
>
> - répéter des actions avec `for`, `while` et `do while` ;
> - choisir la bonne boucle selon la situation ;
> - maîtriser `break` et `continue` ;
> - écrire des boucles imbriquées (table de multiplication).

9.1 Pourquoi des boucles ?

Une **boucle** répète un bloc d'instructions plusieurs fois, sans le réécrire. **Analogie :** « répète 10 pompes » plutôt que « fais une pompe » écrit 10 fois. Les boucles évitent la répétition et gèrent les cas où l'on ignore d'avance combien de fois répéter.

9.2 La boucle `for`

À utiliser quand on **connaît le nombre de répétitions** (ou une plage de nombres).

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

Décomposition de `for (int i = 1; i <= 10; i++)` :

`int i = 1`	**Initialisation** : point de départ, exécutée une seule fois.
`i <= 10`	**Condition** : tant qu'elle est vraie, on continue.
`i++`	**Incrément** : exécuté après chaque tour (ici, +1).

```
CONSOLE
1
2
3
...
10
```

On revient tester la condition après chaque tour. Quand elle devient fausse, on sort.

📖 **Vocabulaire — Itération**

Une **itération** est un « tour » de boucle. La variable de contrôle (souvent nommée `i`) est le **compteur**.

9.3 La boucle `while`

À utiliser quand on **ne sait pas d'avance combien de fois** répéter : on répète « tant que » une condition reste vraie.

```
int compte = 5;
while (compte > 0) {
    System.out.println(compte);
    compte--;          // ⚠️ sans ceci, boucle infinie !
}
System.out.println("Décollage !");
```

```
CONSOLE
5
4
3
2
1
Décollage !
```

🐞 **Erreur fréquente — la boucle infinie**

Si la condition ne devient jamais fausse, la boucle tourne pour toujours et le programme se fige. Vérifiez toujours qu'une variable évolue vers la sortie (ici `compte--`). Pour arrêter un programme figé dans IntelliJ : bouton **Stop** rouge ■.

9.4 La boucle `do ... while`

Comme `while`, mais la condition est testée **à la fin** : le bloc s'exécute donc **au moins une fois**. Idéale pour un menu ou une question qu'on pose au moins une fois.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int reponse;
        do {
            System.out.print("Combien font 3 x 4 ? ");
            reponse = sc.nextInt();
        } while (reponse != 12);   // on redemande TANT QUE la réponse est fausse
        System.out.println("Bravo !");
        sc.close();
    }
}
```

Quelle boucle choisir ?

Situation	Boucle conseillée
Je connais le nombre de tours (1 à 10, parcourir un tableau...)	`for`
Je répète tant qu'une condition est vraie, peut-être 0 fois	`while`
Je veux exécuter au moins une fois (menu, question)	`do ... while`

9.5 `break` et `continue`

- `break` : **sort** immédiatement de la boucle.
- `continue` : **saute** au tour suivant sans finir le tour courant.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;      // on arrête tout à 5
    System.out.println(i);   // affiche 1 2 3 4
}

for (int i = 1; i <= 5; i++) {
    if (i % 2 == 0) continue; // on saute les pairs
    System.out.println(i);      // affiche 1 3 5
}
```

9.6 Boucles imbriquées : la table de multiplication

Une boucle dans une boucle. La boucle interne s'exécute entièrement à chaque tour de la boucle externe.

```java
for (int i = 1; i <= 3; i++) {          // tables 1 à 3
    for (int j = 1; j <= 10; j++) {      // de x1 à x10
        System.out.println(i + " x " + j + " = " + (i * j));
    }
    System.out.println("---");
}
```

```
CONSOLE (EXTRAIT)
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
---
2 x 1 = 2
...
```

Exercices — Chapitre 9

Exercice 9.1 NIVEAU 1

Consignes : affichez les nombres de 1 à 20 avec une boucle `for`.

Exercice 9.2 NIVEAU 1

Consignes : affichez la table de multiplication de 7 (de 7×1 à 7×10).

Exercice 9.3 NIVEAU 2

Consignes : créez un compte à rebours de 10 à 1 avec un `while`, puis affichez « Partez ! ».

Exercice 9.4 NIVEAU 2

Consignes : demandez un mot de passe en boucle *jusqu'à* ce que l'utilisateur tape « java » (utilisez `do while`).

Exercice 9.5 NIVEAU 3 — DÉFI

Consignes : calculez et affichez la somme des nombres de 1 à 100. (Résultat attendu : 5050.)

Corrections — Chapitre 9

9.1 —

```java
for (int i = 1; i <= 20; i++) System.out.println(i);
```

9.2 —

```java
for (int j = 1; j <= 10; j++) {
    System.out.println("7 x " + j + " = " + (7 * j));
}
```

9.3 —

```java
int t = 10;
while (t >= 1) {
    System.out.println(t);
    t--;
}
System.out.println("Partez !");
```

9.4 —

```java
String mdp;
do {
    System.out.print("Mot de passe : ");
    mdp = sc.nextLine();
} while (!mdp.equals("java"));   // on répète tant que ce n'est PAS "java"
System.out.println("Bienvenue !");
```

Le `!` inverse la condition : « tant que ce n'est pas java ». On compare du texte avec `.equals()`.

9.5 —

```java
int somme = 0;
for (int i = 1; i <= 100; i++) {
    somme += i;      // on accumule
}
System.out.println(somme); // 5050
```

> On part de 0, puis on ajoute chaque nombre : c'est le motif de l'**accumulateur**, très courant.

Quiz — Chapitre 9

1. (QCM) Quelle boucle s'exécute au moins une fois ? a) `for` b) `while` c) `do while`

2. (Prédire) Que fait `for (int i=0; i<3; i++)` comme nombre de tours ?

3. (QCM) `break` sert à… a) sauter un tour b) sortir de la boucle c) recommencer

4. (Vrai/Faux) Oublier de faire évoluer la condition d'un `while` peut créer une boucle infinie.

5. (Prédire) Combien de lignes affiche une boucle `i` de 1 à 3 contenant une boucle `j` de 1 à 4 ?

💡 **Réponses**

1-c · 2 : 3 tours (i = 0, 1, 2) · 3-b · 4-Vrai · 5 : 3 × 4 = 12 lignes.

Chapitre 10 — Les méthodes

🎓 **Objectifs d'apprentissage**

- comprendre ce qu'est une méthode et pourquoi découper un programme ;
- déclarer et appeler une méthode ;
- utiliser des paramètres, des arguments et une valeur de retour ;
- comprendre `void`, la portée des variables et la surcharge.

10.1 Qu'est-ce qu'une méthode ?

Une **méthode** est un bloc d'instructions qui porte un nom et que l'on peut « appeler » (déclencher) autant de fois qu'on veut. C'est une petite machine : on lui donne éventuellement des *entrées*, elle fait un travail, et elle peut renvoyer un *résultat*.

📖 **Vocabulaire — Méthode**

Analogie : une méthode est comme un appareil électroménager. Un mixeur : vous mettez des fruits (les *paramètres*), vous appuyez sur le bouton (l'*appel*), il rend un smoothie (la *valeur de retour*). Vous n'avez pas besoin de savoir comment il fonctionne à l'intérieur pour l'utiliser à nouveau.

Vous en connaissez déjà : `main` est une méthode, `System.out.println(...)` appelle une méthode toute prête.

10.2 Pourquoi découper un programme ?

- **Éviter les répétitions** : on écrit le code une fois, on l'appelle partout.
- **Lisibilité** : un grand problème devient une suite de petites tâches nommées.
- **Réutilisation et correction** : on corrige à un seul endroit.

10.3 Déclarer et appeler une méthode simple

```java
public class Main {

    // Déclaration d'une méthode qui affiche un message
    static void direBonjour() {
        System.out.println("Bonjour à tous !");
    }

    public static void main(String[] args) {
        direBonjour();   // appel n°1
        direBonjour();   // appel n°2 : on réutilise sans réécrire
    }
}
```

```
CONSOLE
Bonjour à tous !
Bonjour à tous !
```

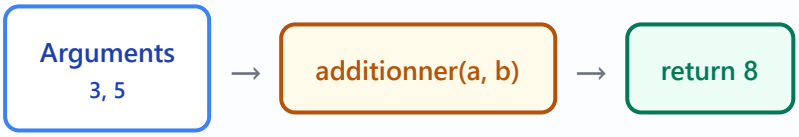
> ⚠️ **Attention — le mot `static`**
>
> Nos méthodes sont `static` car elles sont appelées depuis `main` (elle-même `static`) sans créer d'objet. Ce mot prendra tout son sens au chapitre 13. Pour l'instant, écrivez `static` pour vos méthodes utilitaires.

10.4 Paramètres, arguments et retour

> 📖 **Vocabulaire — Paramètre vs argument**
>
> Le **paramètre** est la variable annoncée dans la *déclaration* de la méthode (« il me faut deux nombres »). L'**argument** est la valeur réelle qu'on *passe* lors de l'appel (« voici 3 et 5 »).

```java
public class Main {

    // a et b sont des PARAMÈTRES ; la méthode RENVOIE un int
    static int additionner(int a, int b) {
        return a + b;   // renvoie le résultat à l'appelant
    }

    public static void main(String[] args) {
        int resultat = additionner(3, 5); // 3 et 5 sont les ARGUMENTS
        System.out.println("Somme : " + resultat); // Somme : 8
    }
}
```

`static int additionner(int a, int b)`	`int` avant le nom = **type de retour** ; `(int a, int b)` = deux paramètres.
`return a + b;`	calcule et **renvoie** la valeur ; l'exécution de la méthode s'arrête ici.
`additionner(3, 5)`	appel avec les arguments 3 et 5 ; renvoie 8, stocké dans `resultat`.



On entre des valeurs, la méthode travaille, elle renvoie un résultat.

10.5 `void` ou valeur de retour ?

- `void` : la méthode **fait** quelque chose (afficher) mais ne renvoie rien.
- Un type (`int` , `double` , `String` , `boolean` ...) : la méthode **calcule et renvoie** une valeur, récupérable avec `=` .

```java
static boolean estMajeur(int age) {
    return age >= 18;   // renvoie true ou false
}
// utilisation :
if (estMajeur(20)) System.out.println("Majeur");
```

10.6 La portée des variables

📖 **Vocabulaire — Portée (scope)**

La **portée** d'une variable est la zone où elle existe. Une variable déclarée dans une méthode (ou un bloc) n'existe que là : on parle de variable **locale**. En dehors, elle est inconnue.

```
static void demo() {
    int x = 10;   // x existe seulement dans demo()
}
public static void main(String[] args) {
    demo();
    // System.out.println(x); ❌ ERREUR : x n'existe pas ici
}
```

10.7 La surcharge de méthodes

📖 **Vocabulaire — Surcharge (overloading)**

La **surcharge** consiste à créer plusieurs méthodes de *même nom* mais avec des *paramètres différents*. Java choisit la bonne selon les arguments fournis.

```
static int additionner(int a, int b) { return a + b; }
static double additionner(double a, double b) { return a + b; }
static int additionner(int a, int b, int c) { return a + b + c; }

additionner(3, 5);        // utilise la 1re → 8
additionner(2.5, 1.5);    // utilise la 2e → 4.0
additionner(1, 2, 3);     // utilise la 3e → 6
```

Exercices — Chapitre 10

Exercice 10.1 NIVEAU 1

Consignes : écrivez une méthode `afficherLigne()` qui affiche « ---------- », et appelez-la trois fois dans `main`.

Exercice 10.2 NIVEAU 2

Consignes : écrivez une méthode `moyenne(double a, double b, double c)` qui renvoie la moyenne des trois nombres. Testez avec 12, 15, 18.

Exercice 10.3 NIVEAU 2

Consignes : écrivez `celsiusVersFahrenheit(double c)` qui renvoie `c * 9 / 5 + 32`. Testez avec 25 (attendu : 77.0).

Exercice 10.4 NIVEAU 3 — DÉFI

Consignes : écrivez `estPair(int n)` qui renvoie un `boolean`, puis utilisez-la dans une boucle pour afficher les nombres pairs de 1 à 20.

Corrections — Chapitre 10

10.1 —

```
static void afficherLigne() {
    System.out.println("----------");
}
// dans main :
afficherLigne(); afficherLigne(); afficherLigne();
```

10.2 —

```
static double moyenne(double a, double b, double c) {
    return (a + b + c) / 3;
}
// main : System.out.println(moyenne(12, 15, 18)); → 15.0
```

Les parenthèses autour de la somme sont indispensables : sinon seule `c` serait divisée.

10.3 —

```
static double celsiusVersFahrenheit(double c) {
    return c * 9 / 5 + 32;
}
// celsiusVersFahrenheit(25) → 77.0
```

10.4 —

```
static boolean estPair(int n) {
    return n % 2 == 0;
}
// main :
for (int i = 1; i <= 20; i++) {
    if (estPair(i)) System.out.println(i);
}
```

On sépare la *logique* (est-ce pair ?) de l'*action* (afficher) : le code devient lisible et réutilisable.

Quiz — Chapitre 10

1. (QCM) Que signifie `void` ? a) renvoie un entier b) ne renvoie rien c) vide la mémoire

2. (Avec vos mots) Différence entre paramètre et argument ?

3. (Vrai/Faux) Une variable locale existe en dehors de sa méthode.

4. (QCM) La surcharge, c'est… a) même nom, paramètres différents b) noms différents c) une erreur

5. (Prédire) Que renvoie `additionner(4, 6)` pour `return a + b;` ?

💡 **Réponses**

1-b · 2 : le paramètre est déclaré dans la méthode, l'argument est la valeur passée à l'appel · 3-Faux · 4-a · 5 : `10`.

Chapitre 11 — Les tableaux

🎓 **Objectifs d'apprentissage**

- créer un tableau et comprendre les index ;
- lire, modifier une case, connaître la taille avec `length` ;
- parcourir un tableau avec `for` et `for-each` ;
- manipuler un tableau à deux dimensions (grille).

11.1 Qu'est-ce qu'un tableau ?

Un **tableau** est une variable qui contient *plusieurs* valeurs du même type, rangées dans des cases numérotées. Au lieu de `note1`, `note2`, `note3`… on a un seul tableau `notes`.

📖 **Vocabulaire — Tableau et index**

Analogie : un tableau est une rangée de casiers numérotés. Chaque casier (case) contient une valeur, et son numéro s'appelle l'**index**. **Point crucial : on compte à partir de 0.** Le 1er casier a l'index 0, le 2e l'index 1, etc.

Un tableau `notes` de 4 cases. `notes[0]` vaut 12, `notes[3]` vaut 18.

11.2 Créer et remplir un tableau

```
// Méthode 1 : on donne directement les valeurs
int[] notes = {12, 15, 9, 18};

// Méthode 2 : on crée un tableau vide de taille 4, puis on remplit
int[] scores = new int[4];  // 4 cases, remplies de 0 par défaut
scores[0] = 100;
scores[1] = 80;
```

11.3 Lire, modifier, taille

```
int[] notes = {12, 15, 9, 18};

System.out.println(notes[0]);   // lit la 1re case → 12
notes[2] = 11;              // modifie la 3e case (9 → 11)
System.out.println(notes.length); // nombre de cases → 4
```

⚠️ **Attention — `length` sans parenthèses (pour un tableau)**

Pour un tableau, on écrit `notes.length` (sans `()`). Pour une chaîne `String`, ce sera `texte.length()` (avec `()`, chapitre 12). Ne les confondez pas.

🐞 **Erreur fréquente — index hors limites**

Un tableau de 4 cases a les index 0, 1, 2, 3. Accéder à `notes[4]` déclenche `ArrayIndexOutOfBoundsException`. Le dernier index valide est toujours `length - 1`.

11.4 Parcourir un tableau

Avec une boucle `for` classique (on a accès à l'index) :

```
for (int i = 0; i < notes.length; i++) {
    System.out.println("Note n°" + i + " : " + notes[i]);
}
```

Avec une boucle `for-each` (plus simple quand on n'a pas besoin de l'index) :

```
for (int note : notes) {   // "pour chaque note dans notes"
    System.out.println(note);
}
```

💡 **Conseil**

Le `for-each` se lit « pour chaque élément de la collection ». Utilisez-le quand vous voulez juste *lire* chaque valeur. Utilisez le `for` classique quand vous avez besoin de l'index (pour modifier, ou connaître la position).

11.5 Les tableaux à deux dimensions (grilles)

Un tableau à deux dimensions est un tableau *de tableaux* : une grille avec des lignes et des colonnes (comme un damier ou une feuille de calcul).

```
int[][] grille = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(grille[1][2]); // ligne 1, colonne 2 → 6

// Parcours complet
for (int i = 0; i < grille.length; i++) {        // lignes
    for (int j = 0; j < grille[i].length; j++) {  // colonnes
        System.out.print(grille[i][j] + " ");
    }
    System.out.println(); // saut de ligne après chaque rangée
}
```

```
CONSOLE
1 2 3
4 5 6
```

Exercices — Chapitre 11

Exercice 11.1 NIVEAU 1

Consignes : créez un tableau contenant 5 prénoms et affichez-les avec un `for-each`.

Exercice 11.2 NIVEAU 2

Consignes : soit `int[] notes = {12, 8, 15, 10, 18};`. Calculez et affichez la moyenne.

Exercice 11.3 NIVEAU 2

Consignes : pour le même tableau, trouvez et affichez la **plus grande** note.

Exercice 11.4 NIVEAU 3 — DÉFI

Consignes : pour le même tableau, trouvez la plus **petite** note ET sa position (index) dans le tableau.

Corrections — Chapitre 11

11.1 —

```
String[] prenoms = {"Léa", "Tom", "Sara", "Max", "Nour"};
for (String p : prenoms) System.out.println(p);
```

11.2 —

```
int[] notes = {12, 8, 15, 10, 18};
int somme = 0;
for (int n : notes) somme += n;
double moyenne = (double) somme / notes.length; // (double) pour éviter la divi
System.out.println("Moyenne : " + moyenne); // 12.6
```

Le `(double)` force une division à virgule : sinon `63 / 5` donnerait 12 au lieu de 12.6.

11.3 —

```
int max = notes[0];         // on suppose que la 1re est la plus grande
for (int n : notes) {
    if (n > max) max = n;    // si on trouve mieux, on met à jour
}
System.out.println("Max : " + max); // 18
```

11.4 —

```
int min = notes[0];
int position = 0;
for (int i = 1; i < notes.length; i++) {
    if (notes[i] < min) {
        min = notes[i];
        position = i;      // on mémorise l'index du minimum
    }
}
System.out.println("Min " + min + " à l'index " + position); // Min 8 à l'index
```

Ici on a besoin de l'index → boucle `for` classique. On garde en mémoire la valeur *et* sa position.

Quiz — Chapitre 11

1. (QCM) Le premier élément d'un tableau a l'index… a) 1 b) 0 c) -1

2. (QCM) La taille d'un tableau `t` se lit… a) `t.size()` b) `t.length` c) `length(t)`

3. (Prédire) Pour `int[] t = {5, 10, 15};`, que vaut `t[2]` ?

4. (Vrai/Faux) `t[t.length]` est un accès valide.

5. (Complétez) La boucle simplifiée pour parcourir sans index s'appelle `for-____`.

💡 **Réponses**

1-b · 2-b · 3 : `15` · 4-Faux (hors limites ; le dernier index est `length-1`) · 5 : `for-each`.

Chapitre 12 — Les chaînes de caractères (String)

> 🎓 **Objectifs d'apprentissage**
>
> - manipuler du texte avec la classe `String` et ses méthodes ;
> - concaténer, transformer, chercher, découper une chaîne ;
> - comprendre **pourquoi** on compare avec `.equals()` et non `==`.

12.1 La classe `String`

Une **chaîne de caractères** (`String`) est une suite de caractères : un mot, une phrase, un texte. C'est un *objet* (chapitre 13) doté de nombreuses **méthodes** pour le manipuler.

```
String phrase = "Bonjour Java";
```

12.2 La concaténation

Concaténer, c'est coller des chaînes bout à bout avec `+` :

```
String prenom = "Jonathan";
String message = "Bonjour " + prenom + " !";
System.out.println(message); // Bonjour Jonathan !
```

12.3 Les méthodes essentielles

Méthode	Rôle	Exemple → résultat
`length()`	nombre de caractères	`"Java".length()` → 4
`toUpperCase()`	tout en majuscules	`"java".toUpperCase()` → "JAVA"
`toLowerCase()`	tout en minuscules	`"JAVA".toLowerCase()` → "java"
`contains()`	contient-elle ce texte ?	`"Bonjour".contains("jour")` → true
`equals()`	égale à une autre chaîne ?	`"a".equals("a")` → true
`equalsIgnoreCase()`	égale, sans tenir compte de la casse	`"OUI".equalsIgnoreCase("oui")` → true
`substring()`	extrait une portion	`"Bonjour".substring(0, 3)` → "Bon"
`replace()`	remplace un texte par un autre	`"chat".replace("h", "")` → "cat"
`trim()`	enlève les espaces au début/fin	`" hi ".trim()` → "hi"
`isEmpty()`	la chaîne est-elle vide ?	`"".isEmpty()` → true

```
String s = "  Bonjour Java  ";
System.out.println(s.trim());                  // "Bonjour Java"
System.out.println(s.trim().length());         // 12
System.out.println(s.toUpperCase());           // "  BONJOUR JAVA  "
System.out.println(s.contains("Java"));       // true
System.out.println("Bonjour".substring(0, 3)); // "Bon" (de l'index 0 inclus à 3 exc
```

💡 **Conseil — on peut enchaîner les méthodes**

`s.trim().toUpperCase()` applique d'abord `trim()`, puis `toUpperCase()` sur le résultat. On lit de gauche à droite.

⚠️ **Attention — `substring(debut, fin)`**

Le premier index est **inclus**, le second **exclu**. `"Bonjour".substring(0, 3)` prend les caractères aux index 0, 1, 2 → "Bon".

12.4 LE point crucial : comparer avec `.equals()`, pas `==`

> 📌 **À retenir — la règle d'or des chaînes**
>
> Pour comparer le **contenu** de deux chaînes, utilisez `a.equals(b)`. **N'utilisez jamais** `==` pour comparer des `String`.

Pourquoi ? Une explication simple :

- `==` vérifie si deux variables pointent vers **le même objet en mémoire** (la même « boîte »).
- `.equals()` vérifie si elles ont **le même contenu** (les mêmes lettres).

Deux chaînes peuvent contenir exactement le même texte tout en étant deux objets distincts en mémoire. `==` répondrait alors « false » à tort.

```
String a = new String("java");
String b = new String("java");

System.out.println(a == b);         // false ❌ (deux objets différents)
System.out.println(a.equals(b));    // true  ✅ (même contenu)
```

> 📖 **Vocabulaire — Référence**
>
> Une variable objet ne contient pas l'objet lui-même, mais une **référence** : une « adresse » qui dit où trouver l'objet. `==` compare les adresses ; `.equals()` compare les contenus. (Détaillé au chapitre 13.)

> 💡 **Conseil — astuce anti-erreur**
>
> Pour comparer une saisie à une valeur fixe, mettez la constante à gauche : `"oui".equals(saisie)`. Ainsi, même si `saisie` est nulle (vide/inexistante), le programme ne plante pas.

Exercices — Chapitre 12

Exercice 12.1 NIVEAU 1

Consignes : demandez un mot et affichez-le en majuscules ainsi que son nombre de lettres.

Exercice 12.2 NIVEAU 2

Consignes : demandez une réponse et affichez « Oui reconnu » si l'utilisateur a tapé « oui », « OUI », « Oui »… (insensible à la casse).

Exercice 12.3 NIVEAU 3 — DÉFI

Consignes : demandez une phrase et affichez-la sans les espaces de début/fin, puis dites si elle contient le mot « Java ».

Corrections — Chapitre 12

12.1 —

```
String mot = sc.nextLine();
System.out.println(mot.toUpperCase());
System.out.println("Longueur : " + mot.length());
```

12.2 —

```
String rep = sc.nextLine();
if (rep.equalsIgnoreCase("oui")) {
    System.out.println("Oui reconnu");
}
```

`equalsIgnoreCase` ignore majuscules/minuscules : « OUI », « Oui » et « oui » sont acceptés.

12.3 —

```
String phrase = sc.nextLine().trim();
System.out.println(phrase);
System.out.println("Contient Java : " + phrase.contains("Java"));
```

Quiz — Chapitre 12

1. (QCM) Comment comparer deux chaînes ? a) `==` b) `.equals()` c) `=`

2. (Prédire) Que renvoie `"Java".length()` ?

3. (Prédire) Que renvoie `"Bonjour".substring(0, 3)` ?

4. (Vrai/Faux) `"Oui".equalsIgnoreCase("oui")` renvoie true.

5. (Avec vos mots) Pourquoi `==` peut renvoyer false pour deux chaînes identiques ?

💡 **Réponses**

1-b · 2 : `4` · 3 : `"Bon"` · 4-Vrai · 5 : `==` compare les adresses mémoire, pas le contenu ; deux objets différents au même texte donnent false.

📌 **🏁 Bravo — les fondamentaux sont acquis !**

Vous maîtrisez variables, opérateurs, entrées, conditions, boucles, méthodes, tableaux et chaînes. C'est déjà de quoi écrire de vrais petits programmes. Faites l'**évaluation « après les bases » (chapitre 23)** et attaquez les mini-projets 1 à 3 (chapitre 21) avant de découvrir la programmation orientée objet.

Chapitre 13 — Introduction à la programmation orientée objet

🎓 **Objectifs d'apprentissage**

- comprendre la différence entre une classe et un objet ;
- créer des attributs et des méthodes ;
- utiliser un constructeur, `new` et `this` ;
- protéger les données avec l'encapsulation (`private` , getters, setters).

13.1 L'idée centrale : classe et objet

📖 **Vocabulaire — Classe et objet**

Une **classe** est un *modèle*, un plan. Un **objet** est une *réalisation concrète* fabriquée à partir de ce plan (on dit une **instance**).

Analogie : la classe est le *plan d'architecte* d'une maison ; chaque maison réellement construite à partir de ce plan est un *objet*. Un seul plan, plusieurs maisons — chacune avec sa couleur, ses habitants, mais la même structure.

Autre image : la classe `Voiture` décrit ce qu'est une voiture (une couleur, une vitesse, la capacité de démarrer). Votre voiture rouge et celle bleue du voisin sont deux *objets* de la classe `Voiture` .

Une classe sert de moule ; chaque `new` fabrique un objet distinct.

📖 **Vocabulaire — Attribut et méthode**

- Un **attribut** (ou champ) est une *donnée* de l'objet : le prénom, l'âge… Ce que l'objet *possède*.
- Une **méthode** est une *action* que l'objet sait faire : se présenter, vieillir… Ce que l'objet *fait*.

13.2 Notre première classe : `Personne`

Créez une nouvelle classe `Personne` (fichier `Personne.java`) :

```
public class Personne {
    // Les ATTRIBUTS (les données de chaque personne)
    String prenom;
    int age;

    // Une MÉTHODE (une action que la personne sait faire)
    void sePresenter() {
        System.out.println("Bonjour, je m'appelle " + prenom + " et j'ai " + age + "
    }
}
```

Puis, dans `Main`, on **crée** et on utilise des objets :

```
public class Main {
    public static void main(String[] args) {
        Personne p1 = new Personne();   // on fabrique un objet
        p1.prenom = "Jonathan";            // on remplit ses attributs
        p1.age = 30;

        Personne p2 = new Personne();   // un deuxième objet, indépendant
        p2.prenom = "Léa";
        p2.age = 25;

        p1.sePresenter();  // Bonjour, je m'appelle Jonathan et j'ai 30 ans.
        p2.sePresenter();  // Bonjour, je m'appelle Léa et j'ai 25 ans.
    }
}
```

> 📖 **Vocabulaire — `new`**
>
> Le mot-clé `new` **fabrique un nouvel objet** à partir d'une classe et réserve sa place en mémoire. `new Personne()` = « construis-moi une nouvelle personne ».

On accède à un attribut ou une méthode avec un **point** : `p1.prenom`, `p1.sePresenter()`.

13.3 Le constructeur et `this`

Remplir les attributs un par un est fastidieux. Un **constructeur** permet de créer un objet *déjà rempli*, en une ligne.

> 📖 **Vocabulaire — Constructeur**
>
> Le **constructeur** est une méthode spéciale, portant le *même nom que la classe*, appelée automatiquement à la création (`new`). Il sert à initialiser l'objet.

```
public class Personne {
    String prenom;
    int age;

    // Le CONSTRUCTEUR (même nom que la classe, pas de type de retour)
    Personne(String prenom, int age) {
        this.prenom = prenom;  // this.prenom = l'attribut ; prenom = le paramètre
        this.age = age;
    }

    void sePresenter() {
        System.out.println("Je suis " + prenom + ", " + age + " ans.");
    }
}
```

```
// Création en une seule ligne, propre :
Personne p1 = new Personne("Jonathan", 30);
p1.sePresenter(); // Je suis Jonathan, 30 ans.
```

📖 **Vocabulaire — `this`**

`this` désigne « l'objet courant », celui sur lequel on travaille. `this.prenom = prenom;` se lit : « range le *paramètre* `prenom` dans l'*attribut* `prenom` de cet objet ». `this` lève l'ambiguïté quand l'attribut et le paramètre portent le même nom.

13.4 L'encapsulation : protéger les données

📖 **Vocabulaire — Encapsulation**

L'**encapsulation** consiste à *cacher* les attributs (les rendre `private`) et à n'y donner accès que par des méthodes contrôlées (**getters** pour lire, **setters** pour modifier). Cela protège l'objet contre les valeurs incohérentes.

Analogie : un distributeur de billets. Vous ne plongez pas la main dans le coffre (attributs `private`) ; vous passez par l'écran et les boutons (méthodes `public`) qui vérifient tout.

📖 **Vocabulaire — `public` et `private`**

- `public` : accessible depuis n'importe où (l'interface publique).
- `private` : accessible *seulement à l'intérieur de la classe* (protégé de l'extérieur).

```java
public class CompteBancaire {
    private double solde;   // caché : on ne peut pas le modifier n'importe comment

    public CompteBancaire(double soldeInitial) {
        this.solde = soldeInitial;
    }

    // GETTER : permet de LIRE le solde
    public double getSolde() {
        return solde;
    }

    // Méthode contrôlée : on vérifie avant de modifier
    public void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
        } else {
            System.out.println("Montant invalide.");
        }
    }
}
```

```java
CompteBancaire c = new CompteBancaire(100);
c.deposer(50);
System.out.println(c.getSolde()); // 150.0
// c.solde = -9999; ❌ IMPOSSIBLE : solde est private
```

📖 **Vocabulaire — Getter et setter**

- Un **getter** (`getSolde()`) renvoie la valeur d'un attribut privé.
- Un **setter** (`setNom(...)`) modifie un attribut, souvent après vérification.

13.5 Retour sur `static`

Maintenant que les objets sont clairs : un membre `static` appartient à la *classe*, pas à un objet. On l'appelle sans `new`. C'est pourquoi `main` et nos méthodes utilitaires sont `static` : elles ne dépendent d'aucun objet précis. À l'inverse, `sePresenter()` (non static) a besoin d'un objet `Personne` pour savoir *quel* prénom afficher.

Exercices — Chapitre 13

Exercice 13.1 NIVEAU 1

Consignes : créez une classe `Livre` avec les attributs `titre` et `auteur`, et une méthode `afficher()`. Créez deux livres dans `main` et affichez-les.

Exercice 13.2 NIVEAU 2

Consignes : ajoutez à `Livre` un constructeur qui reçoit le titre et l'auteur. Créez les livres en une ligne avec `new`.

Exercice 13.3 NIVEAU 3 — DÉFI

Consignes : créez une classe `Etudiant` avec un attribut `private double moyenne`, un constructeur, un getter, et une méthode `estAdmis()` qui renvoie `true` si la moyenne ≥ 10.

Corrections — Chapitre 13

13.1 —

```
public class Livre {
    String titre;
    String auteur;
    void afficher() {
        System.out.println(titre + " — " + auteur);
    }
}
// main :
Livre l1 = new Livre();
l1.titre = "Java facile"; l1.auteur = "Jonathan";
l1.afficher();
```

13.2 —

```
public class Livre {
    String titre, auteur;
    Livre(String titre, String auteur) {
        this.titre = titre;
        this.auteur = auteur;
    }
    void afficher() { System.out.println(titre + " — " + auteur); }
}
// main :
Livre l1 = new Livre("Java facile", "Jonathan");
l1.afficher();
```

13.3 —

```
public class Etudiant {
    private double moyenne;
    public Etudiant(double moyenne) { this.moyenne = moyenne; }
    public double getMoyenne() { return moyenne; }
    public boolean estAdmis() { return moyenne >= 10; }
}
// main :
Etudiant e = new Etudiant(12.5);
System.out.println("Admis : " + e.estAdmis()); // Admis : true
```

L'attribut est `private` : on ne peut le lire que via `getMoyenne()`, ce qui protège les données.

Quiz — Chapitre 13

1. (Avec vos mots) Quelle est la différence entre une classe et un objet ?

2. (QCM) Le constructeur porte le nom… a) `main` b) de la classe c) `construct`

3. (QCM) Que fait `new Personne()` ? a) supprime un objet b) crée un objet c) affiche

4. (Vrai/Faux) Un attribut `private` est accessible directement depuis une autre classe.

5. (Complétez) Le mot-clé qui désigne l'objet courant est `_____`.

💡 **Réponses**

1 : la classe est le modèle/plan, l'objet est une réalisation concrète · 2-b · 3-b · 4-Faux · 5 : `this`.

Chapitre 14 — Héritage et polymorphisme

🎓 **Objectifs d'apprentissage**

- comprendre l'héritage entre une classe parente et une classe enfant ;
- utiliser `extends`, `super` et `@Override` ;
- découvrir le polymorphisme sur l'exemple Animal / Chien / Chat.

💡 **Conseil — rappel avant de commencer**

Ce chapitre s'appuie sur le précédent. Assurez-vous d'être à l'aise avec *classe*, *objet*, *attribut*, *méthode* et *constructeur*. On avance doucement.

14.1 L'héritage : partager du code

📖 **Vocabulaire — Héritage**

L'**héritage** permet à une classe (l'*enfant*) de récupérer automatiquement les attributs et méthodes d'une autre (le *parent*), puis d'ajouter ou de modifier ce qui lui est propre.

Analogie : tous les animaux mangent et dorment. Un chien *est* un animal : il hérite de « manger » et « dormir », et ajoute « aboyer ». On n'a pas à réécrire « manger » pour chaque espèce.

14.2 La classe parente `Animal`

```java
public class Animal {
    String nom;

    Animal(String nom) {
        this.nom = nom;
    }

    void manger() {
        System.out.println(nom + " mange.");
    }

    void faireDuBruit() {
        System.out.println(nom + " fait un bruit.");
    }
}
```

14.3 Les classes enfants avec `extends`

📖 **Vocabulaire — `extends` et `super`**

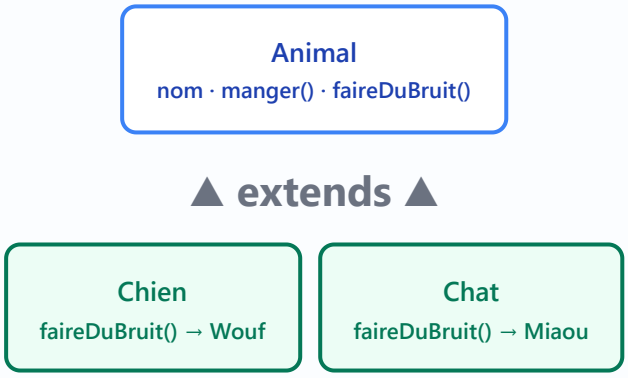
- `extends` établit l'héritage : `class Chien extends Animal` signifie « Chien est un Animal ».
- `super(...)` appelle le constructeur du parent, pour initialiser la partie héritée.

```java
public class Chien extends Animal {
    Chien(String nom) {
        super(nom);   // appelle le constructeur d'Animal pour remplir "nom"
    }

    // On REDÉFINIT la méthode héritée
    @Override
    void faireDuBruit() {
        System.out.println(nom + " aboie : Wouf !");
    }
}
```

```java
public class Chat extends Animal {
    Chat(String nom) {
        super(nom);
    }

    @Override
    void faireDuBruit() {
        System.out.println(nom + " miaule : Miaou !");
    }
}
```

📖 **Vocabulaire — Redéfinition et `@Override`**

Redéfinir (override), c'est fournir dans l'enfant une nouvelle version d'une méthode héritée. L'annotation `@Override` n'est pas obligatoire mais fortement recommandée : elle indique votre intention et Java vérifie que vous redéfinissez bien une méthode existante (protection contre les fautes de frappe).



Chien et Chat héritent de `manger()`, mais chacun a sa version de `faireDuBruit()`.

14.4 Utilisation et polymorphisme

```java
public class Main {
    public static void main(String[] args) {
        Chien rex = new Chien("Rex");
        Chat felix = new Chat("Félix");

        rex.manger();        // Rex mange.       (méthode héritée d'Animal)
        rex.faireDuBruit();  // Rex aboie : Wouf ! (version de Chien)
        felix.faireDuBruit();// Félix miaule : Miaou ! (version de Chat)

        // POLYMORPHISME : on manipule des Chien/Chat comme des Animal
        Animal[] animaux = { new Chien("Médor"), new Chat("Minou") };
        for (Animal a : animaux) {
            a.faireDuBruit(); // chacun réagit selon son vrai type !
        }
    }
}
```

CONSOLE

```
Rex mange.
Rex aboie : Wouf !
Félix miaule : Miaou !
Médor aboie : Wouf !
Minou miaule : Miaou !
```

📖 Vocabulaire — Polymorphisme

Polymorphisme = « plusieurs formes ». On peut traiter un `Chien` et un `Chat` comme des `Animal` (même type commun), et pourtant, à l'appel de `faireDuBruit()`, Java exécute *automatiquement* la bonne version selon le vrai type de l'objet.

Analogie : vous dites « faites du bruit » à un groupe d'animaux ; le chien aboie, le chat miaule. Une même consigne, des réponses adaptées.

📌 À retenir

- `extends` : l'enfant hérite du parent.
- `super(...)` : appelle le constructeur du parent.
- `@Override` : redéfinit une méthode héritée.
- Polymorphisme : le bon comportement est choisi selon le type réel de l'objet.

Exercices — Chapitre 14

Exercice 14.1 NIVEAU 2

Consignes : ajoutez une classe `Vache extends Animal` qui redéfinit `faireDuBruit()` pour afficher « Meuh ! ». Testez-la.

Exercice 14.2 NIVEAU 3 — DÉFI

Consignes : créez une classe parente `Forme` avec une méthode `double aire()` renvoyant 0. Créez `Carre` (attribut côté) et `Cercle` (attribut rayon) qui redéfinissent `aire()`. Mettez plusieurs formes dans un tableau `Forme[]` et affichez toutes leurs aires.

Corrections — Chapitre 14

14.1 —

```
public class Vache extends Animal {
    Vache(String nom) { super(nom); }
    @Override
    void faireDuBruit() { System.out.println(nom + " : Meuh !"); }
}
```

14.2 —

```
public class Forme {
    double aire() { return 0; }
}
public class Carre extends Forme {
    double cote;
    Carre(double cote) { this.cote = cote; }
    @Override double aire() { return cote * cote; }
}
public class Cercle extends Forme {
    double rayon;
    Cercle(double rayon) { this.rayon = rayon; }
    @Override double aire() { return Math.PI * rayon * rayon; }
}
// main :
Forme[] formes = { new Carre(3), new Cercle(2) };
for (Forme f : formes) System.out.println(f.aire()); // 9.0 puis 12.566...
```

`Math.PI` est la constante π fournie par Java. Le polymorphisme fait que chaque forme calcule son aire à sa façon.

Quiz — Chapitre 14

1. (QCM) Quel mot-clé crée l'héritage ? a) `inherits` b) `extends` c) `super`

2. (QCM) `super(nom)` sert à… a) appeler le constructeur parent b) créer un objet c) supprimer

3. (Vrai/Faux) `@Override` aide à éviter les fautes de frappe dans le nom d'une méthode redéfinie.

4. (Avec vos mots) Qu'est-ce que le polymorphisme ?

💡 **Réponses**

1-b · 2-a · 3-Vrai · 4 : la capacité de traiter des objets de types différents via un type commun, chacun exécutant sa propre version d'une méthode.

Chapitre 15 — Les collections

🎓 **Objectifs d'apprentissage**

- comprendre la différence entre un tableau et une collection ;
- maîtriser `ArrayList` : ajouter, supprimer, chercher, parcourir, taille ;
- découvrir `HashSet` (valeurs uniques) et `HashMap` (clé → valeur).

15.1 Pourquoi des collections ?

Un tableau (`int[]`) a une **taille fixe** : décidée à la création, impossible d'ajouter une case ensuite. Or, souvent, on ne connaît pas le nombre d'éléments à l'avance (une liste de tâches qui grandit…). Les **collections** sont des « tableaux intelligents » à taille variable.

📖 **Vocabulaire — Collection**

Une **collection** est un objet qui contient un groupe d'éléments, avec des méthodes toutes prêtes pour les gérer (ajouter, retirer, compter…). La plus utilisée pour débuter est `ArrayList` .

15.2 `ArrayList` : la liste dynamique

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> courses = new ArrayList<>();

        courses.add("Pain");        // ajouter
        courses.add("Lait");
        courses.add("Œufs");

        System.out.println(courses.size());   // taille → 3
        System.out.println(courses.get(0));  // lire l'index 0 → Pain

        courses.remove("Lait");              // supprimer par valeur
        System.out.println(courses.contains("Œufs")); // chercher → true

        // parcourir
        for (String article : courses) {
            System.out.println("- " + article);
        }
    }
}
```

```
CONSOLE
3
Pain
true
- Pain
- Œufs
```

📖 **Vocabulaire — les chevrons `<String>` (généricité)**

`ArrayList<String>` signifie « une liste *de String* ». Les chevrons précisent le type des éléments : Java refusera d'y mettre autre chose, ce qui évite les erreurs. On peut faire `ArrayList<Integer>`, `ArrayList<Personne>`, etc.

⚠️ **Attention — `Integer`, pas `int`**

Dans une collection, on utilise les versions « objet » des types primitifs : `Integer` (pour `int`), `Double` (pour `double`), `Boolean` (pour `boolean`). Java convertit automatiquement dans la plupart des cas : `liste.add(5)` fonctionne.

Les méthodes clés d'`ArrayList`

Méthode	Rôle
`add(x)`	ajoute un élément à la fin
`get(i)`	lit l'élément à l'index i
`set(i, x)`	remplace l'élément à l'index i
`remove(x)` / `remove(i)`	supprime par valeur ou par index
`size()`	nombre d'éléments
`contains(x)`	l'élément est-il présent ? (true/false)
`isEmpty()`	la liste est-elle vide ?
`clear()`	vide la liste

Tableau vs ArrayList

	Tableau `int[]`	`ArrayList`
Taille	Fixe	Variable (grandit/rétrécit)
Ajouter/retirer	Impossible directement	`add()` / `remove()`
Taille	`.length`	`.size()`
Contenu	types primitifs ou objets	objets uniquement

15.3 `HashSet` : des valeurs uniques

Un **HashSet** est une collection *sans doublons* et *sans ordre garanti*. Idéal pour une liste où chaque valeur ne doit apparaître qu'une fois.

```
import java.util.HashSet;

HashSet<String> villes = new HashSet<>();
villes.add("Paris");
villes.add("Genève");
villes.add("Paris");   // doublon ignoré
System.out.println(villes.size()); // 2, pas 3
```

15.4 `HashMap` : associer une clé à une valeur

Une **HashMap** range des paires **clé → valeur**, comme un dictionnaire (un mot → sa définition) ou un annuaire (un nom → un numéro).

```
import java.util.HashMap;

HashMap<String, Integer> ages = new HashMap<>();
ages.put("Jonathan", 30);   // clé "Jonathan" → valeur 30
ages.put("Léa", 25);

System.out.println(ages.get("Léa"));         // 25
System.out.println(ages.containsKey("Léa")); // true

// parcourir les clés
for (String nom : ages.keySet()) {
    System.out.println(nom + " a " + ages.get(nom) + " ans");
}
```

Méthode	Rôle
`put(cle, valeur)`	ajoute ou met à jour une paire
`get(cle)`	renvoie la valeur associée à la clé
`containsKey(cle)`	la clé existe-t-elle ?
`remove(cle)`	supprime la paire
`keySet()`	l'ensemble des clés (pour parcourir)

> 💡 **Conseil — que retenir en priorité ?**
>
> Pour débuter, concentrez-vous sur **ArrayList** : c'est la collection que vous utiliserez 90 % du temps (nos mini-projets et le projet final s'en servent). `HashSet` et `HashMap` viendront naturellement quand vous en aurez besoin.

Exercices — Chapitre 15

Exercice 15.1 NIVEAU 1

Consignes : créez une `ArrayList<String>` de 3 prénoms, ajoutez-en un 4e, retirez-en un, puis affichez la liste et sa taille.

Exercice 15.2 NIVEAU 2

Consignes : créez une `ArrayList<Integer>` de nombres, puis calculez et affichez leur somme.

Exercice 15.3 NIVEAU 3 — DÉFI

Consignes : avec une `HashMap<String, Integer>`, stockez le stock de 3 produits (nom → quantité). Affichez le produit dont le stock est le plus faible.

Corrections — Chapitre 15

15.1 —

```
ArrayList<String> noms = new ArrayList<>();
noms.add("Léa"); noms.add("Tom"); noms.add("Sara");
noms.add("Max");
noms.remove("Tom");
System.out.println(noms);           // [Léa, Sara, Max]
System.out.println(noms.size());    // 3
```

15.2 —

```
ArrayList<Integer> nombres = new ArrayList<>();
nombres.add(10); nombres.add(20); nombres.add(5);
int somme = 0;
for (int n : nombres) somme += n;
System.out.println("Somme : " + somme); // 35
```

15.3 —

```
HashMap<String, Integer> stock = new HashMap<>();
stock.put("Pommes", 12);
stock.put("Poires", 3);
stock.put("Bananes", 7);

String minProduit = null;
int minQte = Integer.MAX_VALUE;
for (String produit : stock.keySet()) {
    if (stock.get(produit) < minQte) {
        minQte = stock.get(produit);
        minProduit = produit;
    }
}
System.out.println("Stock le plus faible : " + minProduit + " (" + minQte + ")"
// Stock le plus faible : Poires (3)
```

`Integer.MAX_VALUE` est le plus grand entier possible : un point de départ sûr pour chercher un minimum.

Quiz — Chapitre 15

1. (QCM) Pour ajouter à une ArrayList : a) `put()` b) `add()` c) `push()`

2. (QCM) La taille d'une ArrayList : a) `.length` b) `.size()` c) `.count()`

3. (Vrai/Faux) Un `HashSet` accepte les doublons.

4. (QCM) Une `HashMap` associe… a) un index à une valeur b) une clé à une valeur c) rien

5. (Avec vos mots) Un avantage d'ArrayList par rapport à un tableau ?

💡 **Réponses**

1-b · 2-b · 3-Faux · 4-b · 5 : sa taille est variable, on peut ajouter/retirer des éléments à volonté.

📌 📊 **Évaluation intermédiaire n°2 — après la POO**

Vous avez terminé la programmation orientée objet ! Une **évaluation « après la POO »** vous attend au chapitre 23. C'est le bon moment pour la faire, puis d'avancer les mini-projets 4 et 5 (chapitre 21).

Chapitre 16 — La gestion des erreurs

🎓 **Objectifs d'apprentissage**

- distinguer les types d'erreurs (syntaxe, compilation, exécution, logique) ;
- comprendre ce qu'est une exception ;
- gérer une exception avec `try`, `catch`, `finally` ;
- lire un message d'erreur Java sans paniquer.

16.1 Les quatre familles d'erreurs

Type	Quand ?	Exemple
Erreur de syntaxe	À l'écriture	Point-virgule ou accolade oublié
Erreur de compilation	Quand `javac` traduit	Utiliser une variable non déclarée
Erreur d'exécution	Pendant que le programme tourne	Division par zéro, index hors limites
Erreur logique	Le programme tourne mais donne un mauvais résultat	Une moyenne calculée avec une mauvaise formule

💡 **Conseil**

Les erreurs de syntaxe et de compilation sont signalées *avant* le lancement (soulignées en rouge par IntelliJ) : les plus faciles à corriger. Les erreurs d'exécution et logiques sont plus sournoises — c'est là que le débogage (chapitre 20) intervient.

16.2 Qu'est-ce qu'une exception ?

📖 **Vocabulaire — Exception**

Une **exception** est un incident qui survient pendant l'exécution et interrompt le programme s'il n'est pas géré : division par zéro, fichier introuvable, texte impossible à convertir en nombre… **Analogie :** c'est le disjoncteur qui saute. Sans protection, tout s'éteint ; avec un `try/catch`, on rattrape le problème et on continue.

```
int resultat = 10 / 0; // ❌ ArithmeticException: / by zero → le programme s'arrête
```

16.3 `try`, `catch`, `finally`

On place le code « risqué » dans un `try`. Si une exception survient, l'exécution saute dans le `catch`, qui gère le problème proprement.

```
try {
    int resultat = 10 / 0;      // code risqué
    System.out.println(resultat);
} catch (ArithmeticException e) {
    System.out.println("Erreur : division par zéro impossible.");
} finally {
    System.out.println("Fin du calcul.");  // s'exécute TOUJOURS
}
System.out.println("Le programme continue !");
```

```
CONSOLE
Erreur : division par zéro impossible.
Fin du calcul.
Le programme continue !
```

`try { ... }`	le code susceptible de provoquer une exception.
`catch (... e)`	ce qu'on fait *si* l'exception survient. `e` contient les détails.
`finally { ... }`	s'exécute dans tous les cas (erreur ou non). Utile pour « nettoyer » (fermer un fichier).

16.4 Exemple utile : saisie protégée

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Entre ton âge : ");
        try {
            int age = sc.nextInt();
            System.out.println("Dans 10 ans tu auras " + (age + 10) + " ans.");
        } catch (InputMismatchException e) {
            System.out.println("Ce n'est pas un nombre valide !");
        }
        sc.close();
    }
}
```

Si l'utilisateur tape « abc » au lieu d'un nombre, au lieu de planter, le programme affiche un message clair.

16.5 Provoquer une exception avec `throw`

On peut aussi *déclencher* une exception volontairement pour signaler une situation anormale :

```java
static void definirAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("L'âge ne peut pas être négatif");
    }
    System.out.println("Âge défini : " + age);
}
```

16.6 Lire un message d'erreur sans paniquer

Reprenons un message typique :

```
CONSOLE
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 5 out of bounds for length 3
        at fr.jonathan.debut.Main.main(Main.java:8)
```

La méthode en 3 réflexes :

1. **Le type d'erreur** (1re ligne) : `ArrayIndexOutOfBoundsException` → un accès hors limites dans un tableau.
2. **Le détail** : `Index 5 out of bounds for length 3` → on a demandé l'index 5 d'un tableau de 3 cases.
3. **L'endroit** : `Main.java:8` → ligne 8. C'est *là* qu'il faut regarder.

> 📌 **À retenir**
>
> Un message d'erreur n'est pas une punition, c'est une **carte au trésor** : il vous dit *quoi*, *pourquoi* et *où*. Lisez-le calmement, ligne par ligne.

Erreurs d'exécution courantes à connaître

Exception	Cause typique
`ArithmeticException`	Division entière par zéro
`ArrayIndexOutOfBoundsException`	Index de tableau invalide
`NullPointerException`	Utiliser un objet qui vaut `null` (inexistant)
`NumberFormatException`	Convertir un texte non numérique en nombre
`InputMismatchException`	Saisie d'un mauvais type avec Scanner

Exercices — Chapitre 16

Exercice 16.1 NIVEAU 1

Consignes : écrivez un `try/catch` qui tente `int x = 7 / 0;` et affiche « Division impossible » en cas d'erreur.

Exercice 16.2 NIVEAU 2

Consignes : demandez deux entiers et affichez leur division ; gérez la division par zéro *et* une saisie non numérique.

Exercice 16.3 NIVEAU 3 — DÉFI

Consignes : combinez une boucle et un `try/catch` pour *redemander* un entier tant que l'utilisateur ne tape pas un nombre valide.

Corrections — Chapitre 16

16.1 —

```
try {
    int x = 7 / 0;
} catch (ArithmeticException e) {
    System.out.println("Division impossible");
}
```

16.2 —

```
try {
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(a / b);
} catch (ArithmeticException e) {
    System.out.println("Division par zéro.");
} catch (InputMismatchException e) {
    System.out.println("Saisie invalide.");
}
```

On peut enchaîner plusieurs `catch` pour traiter différentes erreurs séparément.

16.3 —

```
Scanner sc = new Scanner(System.in);
int nombre = 0;
boolean valide = false;
while (!valide) {
    System.out.print("Entre un entier : ");
    try {
        nombre = sc.nextInt();
        valide = true;              // on sort seulement si la lecture a réussi
    } catch (InputMismatchException e) {
        System.out.println("Ce n'est pas un entier, réessaie.");
        sc.nextLine();                // on vide la saisie fautive
    }
}
System.out.println("Merci : " + nombre);
```

La combinaison boucle + try/catch est le motif classique d'une saisie robuste.

Quiz — Chapitre 16

1. (QCM) Le code risqué se met dans… a) `catch` b) `try` c) `finally`

2. (Vrai/Faux) `finally` s'exécute même s'il n'y a pas d'erreur.

3. (QCM) `10 / 0` (entiers) déclenche… a) `NullPointerException` b) `ArithmeticException` c) rien

4. (Avec vos mots) Que contient la ligne `Main.java:8` d'un message d'erreur ?

5. (QCM) `throw` sert à… a) attraper une exception b) en déclencher une c) l'ignorer

💡 **Réponses**

1-b · 2-Vrai · 3-b · 4 : le fichier et la ligne où l'erreur s'est produite · 5-b.

Chapitre 17 — Lire et écrire dans des fichiers

🎓 **Objectifs d'apprentissage**

- écrire du texte dans un fichier ;
- lire le contenu d'un fichier texte ligne par ligne ;
- gérer les erreurs liées aux fichiers.

17.1 Pourquoi des fichiers ?

Jusqu'ici, dès que le programme s'arrête, toutes les données disparaissent (elles n'existaient qu'en mémoire vive). Pour **conserver** des informations d'une exécution à l'autre — une liste de tâches, des contacts, un score — on les enregistre dans un **fichier** sur le disque.

📖 **Vocabulaire — Persistance**

La **persistance** désigne le fait de conserver des données de façon durable (sur disque), au-delà de la durée de vie du programme. Un fichier texte est la forme la plus simple de persistance.

17.2 Écrire dans un fichier

On utilise `FileWriter`. Comme un fichier peut poser problème (disque plein, dossier interdit…), ces opérations exigent une gestion d'erreur (`try/catch` ou `throws`).

```java
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("taches.txt")) {
            writer.write("Faire les courses\n");   // \n = passage à la ligne
            writer.write("Réviser Java\n");
            writer.write("Appeler Léa\n");
            System.out.println("Fichier écrit avec succès !");
        } catch (IOException e) {
            System.out.println("Erreur d'écriture : " + e.getMessage());
        }
    }
}
```

📖 **Vocabulaire — le *try-with-resources***

`try (FileWriter writer = ...) { ... }` est un `try` spécial : il **ferme automatiquement** le fichier à la fin, même en cas d'erreur. C'est la façon moderne et sûre de manipuler des fichiers (on n'oublie jamais de fermer).

⚠️ **Attention — écraser ou ajouter ?**

`new FileWriter("taches.txt")` **écrase** le fichier existant. Pour *ajouter* à la fin sans effacer, écrivez `new FileWriter("taches.txt", true)` (le `true` = mode « append »).

Le fichier `taches.txt` est créé dans le dossier du projet. Contenu :

```
TACHES.TXT
Faire les courses
Réviser Java
Appeler Léa
```

17.3 Lire un fichier

On lit ligne par ligne avec un `Scanner` branché sur un `File` (au lieu du clavier) :

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        try {
            File fichier = new File("taches.txt");
            Scanner lecteur = new Scanner(fichier);

            while (lecteur.hasNextLine()) {   // tant qu'il reste une ligne
                String ligne = lecteur.nextLine();
                System.out.println("→ " + ligne);
            }
            lecteur.close();
        } catch (FileNotFoundException e) {
            System.out.println("Fichier introuvable !");
        }
    }
}
```

```
CONSOLE
→ Faire les courses
→ Réviser Java
→ Appeler Léa
```

📖 **Vocabulaire — `hasNextLine()`**

`hasNextLine()` renvoie `true` tant qu'il reste au moins une ligne à lire. C'est la condition idéale pour une boucle `while` de lecture.

17.4 Programme complet : un carnet de tâches persistant

On combine tout : ajouter une tâche (en mode append) puis relire tout le fichier.

```java
import java.io.*;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // 1) Ajouter une tâche à la fin du fichier
        System.out.print("Nouvelle tâche : ");
        String tache = sc.nextLine();
        try (FileWriter w = new FileWriter("taches.txt", true)) {
            w.write(tache + "\n");
        } catch (IOException e) {
            System.out.println("Erreur : " + e.getMessage());
        }

        // 2) Relire et afficher toutes les tâches
        System.out.println("--- Vos tâches ---");
        try (Scanner lecteur = new Scanner(new File("taches.txt"))) {
            int n = 1;
            while (lecteur.hasNextLine()) {
                System.out.println(n + ". " + lecteur.nextLine());
                n++;
            }
        } catch (FileNotFoundException e) {
            System.out.println("Aucune tâche pour l'instant.");
        }
        sc.close();
    }
}
```

💡 **Conseil**

`import java.io.*;` importe d'un coup toutes les classes du paquet `java.io` (l'étoile `*` signifie « tout »). Pratique quand on utilise plusieurs classes du même paquet.

Exercices — Chapitre 17

Exercice 17.1 NIVEAU 1

Consignes : écrivez trois prénoms dans un fichier `noms.txt`, un par ligne.

Exercice 17.2 NIVEAU 2

Consignes : lisez `noms.txt` et affichez le nombre total de lignes.

Exercice 17.3 NIVEAU 3 — DÉFI

Consignes : demandez un nom en boucle et enregistrez chaque saisie dans `noms.txt` (mode append) jusqu'à ce que l'utilisateur tape « stop ». Puis affichez tout le fichier.

Corrections — Chapitre 17

17.1 —

```
try (FileWriter w = new FileWriter("noms.txt")) {
    w.write("Léa\nTom\nSara\n");
} catch (IOException e) { System.out.println(e.getMessage()); }
```

17.2 —

```
int total = 0;
try (Scanner lec = new Scanner(new File("noms.txt"))) {
    while (lec.hasNextLine()) { lec.nextLine(); total++; }
} catch (FileNotFoundException e) { System.out.println("Introuvable"); }
System.out.println("Lignes : " + total);
```

17.3 —

```
Scanner sc = new Scanner(System.in);
String saisie;
do {
    System.out.print("Nom (ou 'stop') : ");
    saisie = sc.nextLine();
    if (!saisie.equalsIgnoreCase("stop")) {
        try (FileWriter w = new FileWriter("noms.txt", true)) {
            w.write(saisie + "\n");
        } catch (IOException e) { System.out.println(e.getMessage()); }
    }
} while (!saisie.equalsIgnoreCase("stop"));

try (Scanner lec = new Scanner(new File("noms.txt"))) {
    while (lec.hasNextLine()) System.out.println(lec.nextLine());
} catch (FileNotFoundException e) { System.out.println("Vide"); }
sc.close();
```

Quiz — Chapitre 17

1. (QCM) Quelle classe écrit dans un fichier ? a) `Scanner` b) `FileWriter` c) `System`

2. (Vrai/Faux) `new FileWriter("f.txt", true)` ajoute à la fin sans écraser.

3. (QCM) Pour lire tant qu'il reste des lignes : a) `hasNextLine()` b) `nextInt()` c) `isEmpty()`

4. (Avec vos mots) Quel est l'intérêt du *try-with-resources* ?

💡 **Réponses**

1-b · 2-Vrai · 3-a · 4 : il ferme automatiquement le fichier à la fin, même en cas d'erreur.

Chapitre 18 — Les bonnes pratiques

🎓 **Objectifs d'apprentissage**

- adopter les habitudes qui distinguent un code propre d'un code brouillon ;
- écrire du code lisible, réutilisable et facile à corriger.

Un bon code n'est pas seulement un code qui marche : c'est un code qu'un *humain* (vous, dans six mois) peut lire et modifier facilement. Voici les habitudes essentielles.

18.1 Les dix commandements du code propre

✅ **Bonne pratique — noms clairs**

Préférez `nombreDeTentatives` à `n`, `prixTotal` à `x`. Un bon nom rend le commentaire inutile.

✅ **Bonne pratique — indentation régulière**

Décalez le code à l'intérieur de chaque `{ }`. Dans IntelliJ, **Ctrl+Alt+L** (Cmd+Alt+L sur Mac) réindente tout automatiquement.

✅ **Bonne pratique — méthodes courtes**

Une méthode devrait faire *une* chose. Si elle dépasse ~20 lignes ou fait plusieurs tâches, découpez-la en plusieurs méthodes (chapitre 10).

✅ **Bonne pratique — ne pas se répéter (principe DRY)**

Don't Repeat Yourself. Si vous copiez-collez du code, transformez-le en méthode. Une correction se fera alors à un seul endroit.

✅ **Bonne pratique — commenter à bon escient**

Commentez le *pourquoi*, pas le *quoi*. Un code clair a besoin de peu de commentaires (chapitre 4).

✅ **Bonne pratique — séparer les responsabilités**

Une classe = un rôle. Une classe `Tache` représente une tâche ; une classe `GestionnaireTaches` les gère. Ne mélangez pas tout dans `Main`.

✅ **Bonne pratique — tester régulièrement**

Lancez votre programme souvent, après chaque petit ajout. Il est bien plus facile de trouver une erreur dans 5 nouvelles lignes que dans 200.

✅ **Bonne pratique — lire les messages d'erreur**

Ne fermez jamais un message rouge sans le lire. Type d'erreur + numéro de ligne = 90 % de la solution (chapitre 16).

✅ **Bonne pratique — sauvegarder et versionner**

Enregistrez souvent, et utilisez Git dès que possible (chapitre 19) pour garder un historique de votre travail.

⚠️ **Attention — ne jamais copier sans comprendre**

Recopier du code trouvé en ligne sans le comprendre crée des bugs impossibles à corriger. Prenez le temps de comprendre *chaque ligne* que vous utilisez. C'est la différence entre « ça marche par hasard » et « je sais pourquoi ça marche ».

18.2 Avant / après : un même code

Brouillon :

```
int a=10;int b=20;int c=a+b;if(c>25){System.out.println("grand");}
```

Propre :

```
int largeur = 10;
int hauteur = 20;
int perimetreMoitie = largeur + hauteur;

if (perimetreMoitie > 25) {
    System.out.println("grand");
}
```

Même résultat, mais le second se lit sans effort.

> 🎯 **Défi**
>
> Reprenez l'un de vos programmes précédents et « nettoyez-le » : renommez les variables vagues, réindentez, extrayez une méthode si un bloc se répète.

Chapitre 19 — Utiliser Git et GitHub

🎓 **Objectifs d'apprentissage**

- comprendre à quoi servent Git et GitHub ;
- connaître les commandes de base pour sauvegarder un projet ;
- écrire un fichier `.gitignore`.

19.1 Git et GitHub, c'est quoi ?

📖 **Vocabulaire — Git**

Git est un outil qui enregistre l'*historique* de votre code : chaque version sauvegardée peut être retrouvée. **Analogie :** c'est une machine à remonter le temps pour votre projet. Vous avez cassé quelque chose ? Revenez à la version d'hier.

📖 **Vocabulaire — GitHub**

GitHub est un site web qui héberge vos projets Git *en ligne*. Il sert de sauvegarde à distance et permet de partager/collaborer. Git = l'outil (sur votre machine) ; GitHub = le cloud (sur Internet).

19.2 Le vocabulaire de base

Terme	Signification
Dépôt (repository / repo)	Le dossier de projet suivi par Git, avec tout son historique
Commit	Une « photo » sauvegardée de votre code à un instant donné, avec un message
Push	Envoyer vos commits locaux vers GitHub
Pull	Récupérer les changements depuis GitHub vers votre machine

19.3 Les premières commandes

Après avoir installé Git (**git-scm.com**) et créé un compte GitHub, dans le terminal, au dossier de votre projet :

```
# Une seule fois : dire à Git qui vous êtes
git config --global user.name "Jonathan"
git config --global user.email "jonathan@exemple.com"

# Démarrer le suivi Git dans le dossier
git init

# Enregistrer une première version (commit)
git add .                          # prépare tous les fichiers
git commit -m "Premier commit : mon projet Java"

# Relier à GitHub puis envoyer (URL fournie par GitHub)
git remote add origin https://github.com/jonathan/mon-projet.git
git push -u origin main
```

> 💡 **Conseil — Git dans IntelliJ**
>
> Vous n'êtes pas obligé de tout taper : IntelliJ intègre Git dans son menu **VCS / Git** (boutons pour commit, push, pull). Mais comprendre les commandes aide énormément.

19.4 Le fichier `.gitignore`

> 📖 **Vocabulaire — `.gitignore`**
>
> Le fichier `.gitignore` liste les fichiers/dossiers que Git doit *ignorer* (ne pas sauvegarder) : fichiers compilés, réglages personnels de l'IDE… inutiles à partager.

Exemple de `.gitignore` pour un projet Java + IntelliJ :

```
# Fichiers compilés
*.class
/out/
/target/

# Réglages IntelliJ
/.idea/
*.iml
```

✅ **Bonne pratique — commits fréquents et parlants**

Faites un commit après chaque petite étape terminée, avec un message clair (« Ajout du menu de la calculatrice »), pas « modif » ou « test ». Votre futur vous vous remerciera.

Quiz — Chapitre 19

1. (Avec vos mots) Différence entre Git et GitHub ?

2. (QCM) Un *commit* est… a) une sauvegarde datée du code b) un bug c) un fichier

3. (QCM) Envoyer ses commits vers GitHub : a) `pull` b) `push` c) `init`

4. (Vrai/Faux) `.gitignore` liste les fichiers à NE PAS suivre.

💡 **Réponses**

1 : Git est l'outil local d'historique, GitHub l'hébergement en ligne · 2-a · 3-b · 4-Vrai.

Chapitre 20 — Déboguer un programme

🎓 **Objectifs d'apprentissage**

- comprendre ce qu'est un bug et comment le reproduire ;
- déboguer avec des affichages temporaires ;
- utiliser le débogueur d'IntelliJ : point d'arrêt, pas-à-pas, inspection.

20.1 Qu'est-ce qu'un bug ?

📖 **Vocabulaire — Bug et débogage**

Un **bug** est une erreur dans le programme qui produit un comportement inattendu. **Déboguer**, c'est le traquer et le corriger. (Le mot vient d'un vrai insecte — *bug* — coincé dans un ordinateur en 1947 !)

20.2 La méthode : reproduire d'abord

Avant de corriger, il faut **reproduire** le bug de façon fiable : quelles données, quelles étapes provoquent le problème ? Un bug qu'on sait déclencher à volonté est un bug à moitié résolu.

1. Notez ce que vous *attendiez* et ce que vous avez *obtenu*.
2. Identifiez la plus petite action qui déclenche le problème.
3. Isolez la zone de code concernée.

20.3 Technique 1 : les affichages temporaires

La méthode la plus simple : parsemer le code de `System.out.println` pour observer les valeurs.

```java
int total = 0;
for (int i = 1; i <= 5; i++) {
    total += i;
    System.out.println("DEBUG i=" + i + " total=" + total); // espion temporaire
}
```

On voit ainsi évoluer les variables tour après tour et on repère où ça dérape.

> 💡 **Conseil**
>
> Préfixez vos messages de débogage par « DEBUG » pour les repérer et les supprimer facilement une fois le bug corrigé.

20.4 Technique 2 : le débogueur d'IntelliJ

Plus puissant : le **débogueur** permet de *geler* le programme et d'examiner tout en direct.

> 📖 **Vocabulaire — Point d'arrêt (breakpoint)**
>
> Un **point d'arrêt** est un marqueur posé sur une ligne : le programme s'y *met en pause*, ce qui vous laisse inspecter l'état à cet instant précis.

Marche à suivre dans IntelliJ

1. **Poser un point d'arrêt :** cliquez dans la marge, à gauche d'un numéro de ligne. Un point rouge apparaît.
2. **Lancer en mode debug :** cliquez sur l'icône « insecte » 🐞 (Debug) au lieu de la flèche verte.
3. **Le programme se met en pause** sur la ligne marquée. En bas, l'onglet « Debug » affiche la valeur de chaque variable.
4. **Avancer pas à pas :**
   - *Step Over* (F8) : exécute la ligne courante et passe à la suivante.
   - *Step Into* (F7) : entre dans la méthode appelée pour la suivre de l'intérieur.
   - *Resume* (F9) : reprend l'exécution jusqu'au prochain point d'arrêt.
5. **Inspecter :** survolez une variable ou lisez le panneau « Variables » pour voir sa valeur en temps réel.

📌 **À retenir — la démarche du débogage**

1. Reproduire le bug de façon fiable.
2. Poser un point d'arrêt avant la zone suspecte.
3. Avancer pas à pas en surveillant les variables.
4. Repérer la ligne où une valeur devient incorrecte.
5. Corriger, puis relancer pour vérifier.

Exercice — Chapitre 20

Exercice 20.1 NIVEAU 2 — CHASSE AU BUG

Consignes : ce code doit calculer la somme de 1 à 5 (attendu : 15), mais il affiche autre chose. Trouvez et corrigez le bug (aidez-vous d'affichages ou d'un point d'arrêt).

```
int total = 0;
for (int i = 1; i < 5; i++) {
    total += i;
}
System.out.println(total);
```

Correction 20.1 — La condition `i < 5` s'arrête à i = 4 : on additionne 1+2+3+4 = 10, pas 15. Le nombre 5 est oublié. Il faut `i <= 5` :

```
for (int i = 1; i <= 5; i++) {
    total += i;
}
// 1+2+3+4+5 = 15 ✅
```

C'est une **erreur logique** classique, dite « erreur de bord » (*off-by-one*) : le programme tourne sans planter mais donne un résultat faux. Un point d'arrêt montrant que la boucle s'arrête à i=4 révèle immédiatement le problème.

Quiz — Chapitre 20

1. (Avec vos mots) Qu'est-ce qu'un point d'arrêt ?

2. (QCM) Quelle touche exécute la ligne courante sans entrer dans les méthodes ?
a) F7 b) F8 c) F9

3. (Vrai/Faux) Reproduire un bug est une étape inutile.

4. (QCM) `for (int i=1; i<5; i++)` exécute la boucle combien de fois ? a) 4 b) 5 c) 6

💡 **Réponses**

1 : un marqueur qui met le programme en pause sur une ligne pour l'inspecter · 2-b (Step Over) · 3-Faux (c'est essentiel) · 4-a (i = 1,2,3,4).

Chapitre 21 — Mini-projets guidés

🎓 **Objectifs d'apprentissage**

Assembler tout ce que vous avez appris dans cinq projets complets et réutilisables. Chaque projet est fourni avec son code entier, expliqué, et des idées d'améliorations.

💡 **Conseil — la bonne méthode**

Pour chaque projet : **essayez d'abord de le construire vous-même** à partir de l'énoncé, puis comparez avec la solution. C'est en butant sur les difficultés que l'on progresse le plus.

Mini-projet 1 — La calculatrice

Notions : Scanner · switch · méthodes · gestion d'erreurs

But : un menu propose les 4 opérations ; l'utilisateur choisit, saisit deux nombres, obtient le résultat. La division par zéro est gérée.

```java
import java.util.Scanner;

public class Calculatrice {

    static double calculer(double a, double b, int op) {
        switch (op) {
            case 1: return a + b;
            case 2: return a - b;
            case 3: return a * b;
            case 4:
                if (b == 0) {
                    throw new ArithmeticException("Division par zéro");
                }
                return a / b;
            default: throw new IllegalArgumentException("Opération inconnue");
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("=== CALCULATRICE ===");
        System.out.println("1) Addition");
        System.out.println("2) Soustraction");
        System.out.println("3) Multiplication");
        System.out.println("4) Division");
        System.out.print("Votre choix : ");
        int choix = sc.nextInt();

        System.out.print("Premier nombre : ");
        double a = sc.nextDouble();
        System.out.print("Deuxième nombre : ");
        double b = sc.nextDouble();

        try {
            double resultat = calculer(a, b, choix);
            System.out.println("Résultat : " + resultat);
        } catch (ArithmeticException e) {
            System.out.println("Erreur : " + e.getMessage());
        } catch (IllegalArgumentException e) {
            System.out.println("Erreur : " + e.getMessage());
        }
        sc.close();
    }
}
```

EXEMPLE

```
Votre choix : 4
Premier nombre : 10
Deuxième nombre : 0
Erreur : Division par zéro
```

Explication : on isole le calcul dans une méthode `calculer` qui *lance* une exception si besoin. Le `main` gère l'affichage et rattrape les erreurs. Cette séparation (calcul / interface) est une bonne pratique.

🎯 **Défi — améliorations**

- Boucler pour enchaîner plusieurs calculs (menu « 5) Quitter »).
- Ajouter le modulo et la puissance.
- Protéger la saisie contre un texte non numérique (`try/catch InputMismatchException`).

Mini-projet 2 — Le jeu du nombre mystère

Notions : Random · while · conditions · compteur · rejouer

But : l'ordinateur tire un nombre entre 1 et 100 ; le joueur devine, guidé par « plus grand » / « plus petit », jusqu'à trouver. On compte les tentatives et on propose de rejouer.

```java
import java.util.Random;
import java.util.Scanner;

public class NombreMystere {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random random = new Random();
        boolean rejouer = true;

        while (rejouer) {
            int secret = random.nextInt(100) + 1; // entre 1 et 100
            int tentative = 0;
            int proposition;

            System.out.println("J'ai choisi un nombre entre 1 et 100. À toi !");

            do {
                System.out.print("Ta proposition : ");
                proposition = sc.nextInt();
                tentative++;

                if (proposition < secret) {
                    System.out.println("C'est plus grand ⬆");
                } else if (proposition > secret) {
                    System.out.println("C'est plus petit ⬇");
                } else {
                    System.out.println("🎉 Bravo ! Trouvé en " + tentative + " tenta
                }
            } while (proposition != secret);

            System.out.print("Rejouer ? (oui/non) : ");
            String reponse = sc.next();
            rejouer = reponse.equalsIgnoreCase("oui");
        }
        System.out.println("Merci d'avoir joué !");
        sc.close();
    }
}
```

EXEMPLE

```
Ta proposition : 50
C'est plus petit ⬇
Ta proposition : 25
C'est plus grand ⬆
Ta proposition : 37
🎉 Bravo ! Trouvé en 3 tentatives.
```

Explication : `random.nextInt(100)` renvoie 0–99 ; on ajoute 1 pour obtenir 1–100. Le `do/while` redemande jusqu'à trouver. Une boucle `while (rejouer)` englobe le tout pour rejouer.

> 🎯 **Défi — améliorations**
>
> - Limiter le nombre d'essais (ex. 7 max).
> - Afficher le meilleur score (moins de tentatives).
> - Laisser le joueur choisir la difficulté (1–100, 1–1000…).

Mini-projet 3 — Le convertisseur

Notions : menu · méthodes · double · switch

But : un menu propose plusieurs conversions : Celsius→Fahrenheit, km→miles, euros→francs suisses (CHF), minutes→heures et minutes.

> ⚠️ **Attention — taux de change pédagogique**
>
> Le taux euro → franc suisse utilisé ici (`0.95`) est un **exemple fixé dans le programme à but pédagogique**. Les taux réels varient chaque jour ; ne l'utilisez pas pour une vraie conversion.

```java
import java.util.Scanner;

public class Convertisseur {

    static double celsiusVersFahrenheit(double c) { return c * 9 / 5 + 32; }
    static double kmVersMiles(double km)        { return km * 0.621371; }
    static double eurosVersChf(double euros)    { return euros * 0.95; } // taux fic

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("=== CONVERTISSEUR ===");
        System.out.println("1) Celsius -> Fahrenheit");
        System.out.println("2) Kilomètres -> Miles");
        System.out.println("3) Euros -> Francs suisses (taux fictif)");
        System.out.println("4) Minutes -> Heures et minutes");
        System.out.print("Choix : ");
        int choix = sc.nextInt();

        switch (choix) {
            case 1 -> {
                System.out.print("Degrés Celsius : ");
                double c = sc.nextDouble();
                System.out.println(c + " °C = " + celsiusVersFahrenheit(c) + " °F");
            }
            case 2 -> {
                System.out.print("Kilomètres : ");
                double km = sc.nextDouble();
                System.out.println(km + " km = " + kmVersMiles(km) + " miles");
            }
            case 3 -> {
                System.out.print("Euros : ");
                double e = sc.nextDouble();
                System.out.println(e + " € = " + eurosVersChf(e) + " CHF");
            }
            case 4 -> {
                System.out.print("Minutes : ");
                int min = sc.nextInt();
                int heures = min / 60;   // division entière
                int reste = min % 60;    // modulo
                System.out.println(min + " min = " + heures + " h " + reste + " min"
            }
            default -> System.out.println("Choix invalide.");
        }
        sc.close();
    }
}
```

```
EXEMPLE
Choix : 4
Minutes : 135
135 min = 2 h 15 min
```

Explication : chaque conversion est une méthode dédiée (lisible et testable). Le cas « minutes » illustre joliment la division entière (`/`) et le modulo (`%`) travaillant ensemble.

🎯 **Défi — améliorations**

- Ajouter les conversions inverses (Fahrenheit→Celsius, etc.).
- Permettre à l'utilisateur de saisir lui-même le taux de change.
- Boucler le menu jusqu'à ce qu'on choisisse « Quitter ».

Mini-projet 4 — Le gestionnaire de tâches

Notions : classe · objets · ArrayList · boucle de menu · encapsulation

But : une petite application de « to-do list ». On peut ajouter des tâches, les afficher, en supprimer une, et marquer une tâche comme terminée. Chaque tâche est un objet de la classe `Tache`.

La classe `Tache` (fichier `Tache.java`)

```java
public class Tache {
    private String description;
    private boolean terminee;

    public Tache(String description) {
        this.description = description;
        this.terminee = false;      // une nouvelle tâche n'est pas terminée
    }

    public void marquerTerminee() { this.terminee = true; }
    public boolean estTerminee()  { return terminee; }
    public String getDescription() { return description; }

    @Override
    public String toString() {
        String etat = terminee ? "[X]" : "[ ]"; // case cochée ou non
        return etat + " " + description;
    }
}
```

📖 **Vocabulaire — `toString()`**

La méthode `toString()` définit comment un objet s'affiche sous forme de texte. En la redéfinissant, `System.out.println(maTache)` affichera « [X] Réviser Java » au lieu d'une adresse mémoire illisible. L'expression `condition ? valeurSiVrai : valeurSiFaux` est l'*opérateur ternaire*, un `if/else` condensé.

Le programme principal (fichier `GestionnaireTaches.java`)

```java
import java.util.ArrayList;
import java.util.Scanner;

public class GestionnaireTaches {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ArrayList<Tache> taches = new ArrayList<>();
        int choix;

        do {
            System.out.println("\n=== MES TÂCHES ===");
            System.out.println("1) Ajouter une tâche");
            System.out.println("2) Afficher les tâches");
            System.out.println("3) Marquer une tâche terminée");
            System.out.println("4) Supprimer une tâche");
            System.out.println("0) Quitter");
            System.out.print("Choix : ");
            choix = sc.nextInt();
            sc.nextLine(); // on avale le retour à la ligne

            switch (choix) {
                case 1 -> {
                    System.out.print("Description : ");
                    String desc = sc.nextLine();
                    taches.add(new Tache(desc));
                    System.out.println("Tâche ajoutée !");
                }
                case 2 -> afficher(taches);
                case 3 -> {
                    afficher(taches);
                    System.out.print("Numéro à terminer : ");
                    int i = sc.nextInt() - 1; // l'utilisateur compte à partir de 1
                    if (i >= 0 && i < taches.size()) {
                        taches.get(i).marquerTerminee();
                        System.out.println("Bravo !");
                    } else System.out.println("Numéro invalide.");
                }
                case 4 -> {
                    afficher(taches);
                    System.out.print("Numéro à supprimer : ");
                    int i = sc.nextInt() - 1;
                    if (i >= 0 && i < taches.size()) {
                        taches.remove(i);
                        System.out.println("Supprimée.");
                    } else System.out.println("Numéro invalide.");
                }
                case 0 -> System.out.println("Au revoir !");
                default -> System.out.println("Choix inconnu.");
            }
```

```
        } while (choix != 0);
        sc.close();
    }

    // Méthode utilitaire pour afficher la liste numérotée
    static void afficher(ArrayList<Tache> taches) {
        if (taches.isEmpty()) {
            System.out.println("(aucune tâche)");
            return;
        }
        for (int i = 0; i < taches.size(); i++) {
            System.out.println((i + 1) + ". " + taches.get(i));
        }
    }
}
```

EXEMPLE

```
=== MES TÂCHES ===
Choix : 2
1. [ ] Faire les courses
2. [X] Réviser Java
```

Explication : la liste `ArrayList<Tache>` grandit à volonté. On affiche les tâches numérotées à partir de 1 (plus naturel pour l'utilisateur), mais on accède à l'ArrayList à partir de 0 : d'où le `- 1`. La méthode `afficher` évite de répéter le code d'affichage.

🎯 **Défi — améliorations**

- Sauvegarder les tâches dans un fichier (chapitre 17) pour les retrouver au prochain lancement.
- Ajouter une priorité (haute/moyenne/basse) à chaque tâche.
- Afficher séparément les tâches en cours et terminées.

Mini-projet 5 — Le gestionnaire de contacts

Notions : classe · ArrayList · recherche · modification · fichiers

But : gérer un carnet d'adresses : ajouter, afficher, rechercher, modifier, supprimer des contacts, et **enregistrer dans un fichier** pour les conserver.

La classe `Contact`

```java
public class Contact {
    private String nom;
    private String telephone;

    public Contact(String nom, String telephone) {
        this.nom = nom;
        this.telephone = telephone;
    }

    public String getNom() { return nom; }
    public String getTelephone() { return telephone; }
    public void setTelephone(String telephone) { this.telephone = telephone; }

    // Ligne prête à écrire dans le fichier : "nom;téléphone"
    public String versLigne() { return nom + ";" + telephone; }

    @Override
    public String toString() { return nom + " — " + telephone; }
}
```

Le programme principal (extraits clés)

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class GestionnaireContacts {
    static final String FICHIER = "contacts.txt";

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ArrayList<Contact> contacts = charger(); // on recharge au démarrage
        int choix;

        do {
            System.out.println("\n=== CONTACTS ===");
            System.out.println("1) Ajouter  2) Afficher  3) Rechercher  4) Modifier
            System.out.print("Choix : ");
            choix = sc.nextInt(); sc.nextLine();

            switch (choix) {
                case 1 -> {
                    System.out.print("Nom : ");       String n = sc.nextLine();
                    System.out.print("Téléphone : "); String t = sc.nextLine();
                    contacts.add(new Contact(n, t));
                    sauvegarder(contacts);
                    System.out.println("Ajouté et sauvegardé.");
                }
                case 2 -> {
                    if (contacts.isEmpty()) System.out.println("(vide)");
                    for (Contact c : contacts) System.out.println("- " + c);
                }
                case 3 -> {
                    System.out.print("Nom à chercher : ");
                    String q = sc.nextLine();
                    Contact trouve = rechercher(contacts, q);
                    System.out.println(trouve != null ? trouve : "Introuvable.");
                }
                case 4 -> {
                    System.out.print("Nom à modifier : ");
                    Contact c = rechercher(contacts, sc.nextLine());
                    if (c != null) {
                        System.out.print("Nouveau téléphone : ");
                        c.setTelephone(sc.nextLine());
                        sauvegarder(contacts);
                        System.out.println("Modifié.");
                    } else System.out.println("Introuvable.");
                }
                case 5 -> {
                    System.out.print("Nom à supprimer : ");
                    Contact c = rechercher(contacts, sc.nextLine());
```

```
                    if (c != null) {
                        contacts.remove(c);
                        sauvegarder(contacts);
                        System.out.println("Supprimé.");
                    } else System.out.println("Introuvable.");
                }
            }
        } while (choix != 0);
        sc.close();
    }

    // Recherche insensible à la casse
    static Contact rechercher(ArrayList<Contact> contacts, String nom) {
        for (Contact c : contacts) {
            if (c.getNom().equalsIgnoreCase(nom)) return c;
        }
        return null; // rien trouvé
    }

    // Écrit tous les contacts dans le fichier
    static void sauvegarder(ArrayList<Contact> contacts) {
        try (FileWriter w = new FileWriter(FICHIER)) {
            for (Contact c : contacts) w.write(c.versLigne() + "\n");
        } catch (IOException e) {
            System.out.println("Erreur de sauvegarde : " + e.getMessage());
        }
    }

    // Recharge les contacts depuis le fichier (s'il existe)
    static ArrayList<Contact> charger() {
        ArrayList<Contact> liste = new ArrayList<>();
        File f = new File(FICHIER);
        if (!f.exists()) return liste; // première utilisation : rien à charger
        try (Scanner lec = new Scanner(f)) {
            while (lec.hasNextLine()) {
                String[] parties = lec.nextLine().split(";"); // coupe "nom;tel"
                if (parties.length == 2) {
                    liste.add(new Contact(parties[0], parties[1]));
                }
            }
        } catch (FileNotFoundException e) {
            System.out.println("Aucun fichier de contacts.");
        }
        return liste;
    }
}
```

📖 **Vocabulaire — `split()`**

`"Léa;0600".split(";")` découpe la chaîne à chaque `;` et renvoie un tableau : `["Léa", "0600"]`. Idéal pour relire des données enregistrées « champ par champ ».

Explication : ce projet réunit presque tout le livre — une classe avec encapsulation, une `ArrayList` d'objets, la recherche (qui renvoie `null` si rien n'est trouvé), et la persistance dans un fichier avec sauvegarde/rechargement automatiques. Chaque action qui modifie les données appelle `sauvegarder()` pour que rien ne soit perdu.

🎯 **Défi — améliorations**

- Ajouter un champ e-mail (attention au séparateur `;`).
- Trier les contacts par ordre alphabétique avant l'affichage.
- Empêcher deux contacts d'avoir le même nom.

Chapitre 22 — Projet final : gestionnaire de budget personnel

🎓 **Objectif du projet**

Construire, étape par étape, une véritable application console de gestion de budget qui mobilise **tout** ce que vous avez appris : variables, conditions, boucles, méthodes, classes, objets, collections, exceptions et fichiers.

Fonctionnalités visées : saisir un revenu, ajouter/modifier/supprimer des dépenses, attribuer une catégorie, afficher les dépenses (toutes ou par catégorie), calculer le total et le solde, sauvegarder et recharger les données.

💡 **Conseil — construisez par étapes**

Ne cherchez pas à tout écrire d'un coup. Suivez les 6 étapes : à chaque étape, le programme *fonctionne* déjà et fait un peu plus. Testez après chacune. C'est ainsi que travaillent les vrais développeurs.

Vue d'ensemble

Architecture : une classe pour *une* dépense, une liste pour les regrouper, une classe pour piloter, un fichier pour conserver.

Étape 1 — Le squelette et le revenu

Objectif : afficher un menu qui tourne en boucle et permettre de saisir le revenu.

Notions utilisées : `Scanner`, `do/while`, `switch`, variables.

```java
import java.util.Scanner;

public class Budget {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double revenu = 0;
        int choix;

        do {
            System.out.println("\n=== MON BUDGET ===");
            System.out.println("1) Définir le revenu");
            System.out.println("0) Quitter");
            System.out.print("Choix : ");
            choix = sc.nextInt(); sc.nextLine();

            if (choix == 1) {
                System.out.print("Revenu du mois : ");
                revenu = sc.nextDouble(); sc.nextLine();
                System.out.println("Revenu enregistré : " + revenu);
            }
        } while (choix != 0);
        sc.close();
    }
}
```

Test : lancez, choisissez 1, entrez 3000, vérifiez l'affichage, puis 0 pour quitter.

Étape 2 — La classe `Depense` et l'ajout

Objectif : représenter une dépense comme un objet et pouvoir en ajouter dans une `ArrayList`.

Notions utilisées : classe, constructeur, encapsulation, `ArrayList`, `toString()`.

```java
// Fichier Depense.java
public class Depense {
    private String description;
    private String categorie;
    private double montant;

    public Depense(String description, String categorie, double montant) {
        this.description = description;
        this.categorie = categorie;
        this.montant = montant;
    }

    public String getDescription() { return description; }
    public String getCategorie()   { return categorie; }
    public double getMontant()     { return montant; }
    public void setMontant(double m) { this.montant = m; }

    public String versLigne() { return description + ";" + categorie + ";" + montant

    @Override
    public String toString() {
        return description + " (" + categorie + ") : " + montant + " €";
    }
}
```

Dans `Budget`, on ajoute la liste et l'option « Ajouter une dépense » :

```java
ArrayList<Depense> depenses = new ArrayList<>();
// ... dans le switch :
case 2 -> {
    System.out.print("Description : "); String d = sc.nextLine();
    System.out.print("Catégorie : ");   String cat = sc.nextLine();
    System.out.print("Montant : ");     double m = sc.nextDouble(); sc.nextLine();
    depenses.add(new Depense(d, cat, m));
    System.out.println("Dépense ajoutée.");
}
```

Test : ajoutez « Courses / Alimentation / 85.50 » et vérifiez qu'aucune erreur ne survient.

Étape 3 — Afficher, total et solde

Objectif : lister toutes les dépenses, calculer leur total et le solde restant.

Notions utilisées : `for-each`, accumulateur, méthodes.

```
// Méthodes utilitaires dans Budget
static void afficherDepenses(ArrayList<Depense> depenses) {
    if (depenses.isEmpty()) { System.out.println("(aucune dépense)"); return; }
    for (int i = 0; i < depenses.size(); i++) {
        System.out.println((i + 1) + ". " + depenses.get(i));
    }
}

static double total(ArrayList<Depense> depenses) {
    double somme = 0;
    for (Depense d : depenses) somme += d.getMontant();
    return somme;
}
```

```
// dans le switch :
case 3 -> afficherDepenses(depenses);
case 4 -> {
    double t = total(depenses);
    System.out.println("Total des dépenses : " + t + " €");
    System.out.println("Solde disponible : " + (revenu - t) + " €");
}
```

EXEMPLE

```
Total des dépenses : 285.5 €
Solde disponible : 2714.5 €
```

Test : ajoutez deux ou trois dépenses, affichez-les, puis vérifiez le total et le solde (revenu − total).

Étape 4 — Dépenses par catégorie

Objectif : afficher le total dépensé pour une catégorie donnée.

Notions utilisées : parcours conditionnel, comparaison de chaînes.

```java
static double totalCategorie(ArrayList<Depense> depenses, String categorie) {
    double somme = 0;
    for (Depense d : depenses) {
        if (d.getCategorie().equalsIgnoreCase(categorie)) {
            somme += d.getMontant();
        }
    }
    return somme;
}
```

```java
case 5 -> {
    System.out.print("Catégorie : ");
    String cat = sc.nextLine();
    System.out.println("Total " + cat + " : " + totalCategorie(depenses, cat) + " €"
}
```

Test : ajoutez plusieurs dépenses « Alimentation », demandez le total de cette catégorie, vérifiez la somme.

Étape 5 — Modifier et supprimer une dépense

Objectif : corriger le montant d'une dépense ou la retirer.

Notions utilisées : index d'ArrayList, setter, validation.

```
case 6 -> { // modifier
    afficherDepenses(depenses);
    System.out.print("Numéro à modifier : ");
    int i = sc.nextInt() - 1; sc.nextLine();
    if (i >= 0 && i < depenses.size()) {
        System.out.print("Nouveau montant : ");
        depenses.get(i).setMontant(sc.nextDouble()); sc.nextLine();
        System.out.println("Modifié.");
    } else System.out.println("Numéro invalide.");
}
case 7 -> { // supprimer
    afficherDepenses(depenses);
    System.out.print("Numéro à supprimer : ");
    int i = sc.nextInt() - 1; sc.nextLine();
    if (i >= 0 && i < depenses.size()) {
        depenses.remove(i);
        System.out.println("Supprimée.");
    } else System.out.println("Numéro invalide.");
}
```

Test : modifiez le montant d'une dépense, vérifiez le nouveau total ; supprimez-en une, vérifiez qu'elle disparaît.

Étape 6 — Sauvegarder et recharger (fichiers)

Objectif : conserver les données entre deux lancements.

Notions utilisées : `FileWriter`, `Scanner` sur fichier, `split()`, exceptions.

```
static final String FICHIER = "budget.txt";

static void sauvegarder(double revenu, ArrayList<Depense> depenses) {
    try (FileWriter w = new FileWriter(FICHIER)) {
        w.write("REVENU;" + revenu + "\n");          // 1re ligne = revenu
        for (Depense d : depenses) {
            w.write(d.versLigne() + "\n");            // une dépense par ligne
        }
    } catch (IOException e) {
        System.out.println("Erreur de sauvegarde : " + e.getMessage());
    }
}
```

Le rechargement lit le fichier au démarrage. Comme une méthode ne peut renvoyer qu'une valeur mais qu'on veut récupérer le revenu *et* la liste, on remplit la liste passée en paramètre (les objets sont partagés par référence) et on renvoie le revenu :

```
static double charger(ArrayList<Depense> depenses) {
    double revenu = 0;
    File f = new File(FICHIER);
    if (!f.exists()) return 0;              // premier lancement
    try (Scanner lec = new Scanner(f)) {
        while (lec.hasNextLine()) {
            String[] p = lec.nextLine().split(";");
            if (p[0].equals("REVENU")) {
                revenu = Double.parseDouble(p[1]);
            } else if (p.length == 3) {
                depenses.add(new Depense(p[0], p[1], Double.parseDouble(p[2])));
            }
        }
    } catch (Exception e) {
        System.out.println("Fichier illisible, on repart de zéro.");
    }
    return revenu;
}
```

> 📖 **Vocabulaire — `Double.parseDouble()`**
>
> Les fichiers ne contiennent que du texte. `Double.parseDouble("85.5")` convertit le texte `"85.5"` en nombre `double` 85.5. De même, `Integer.parseInt("7")` convertit en `int`.

Test : ajoutez des dépenses, quittez, relancez : les données doivent réapparaître.

Le programme complet

Voici `Budget.java` assemblé (la classe `Depense` ci-dessus reste dans son propre fichier). On sauvegarde après chaque modification et on recharge au démarrage.

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class Budget {
    static final String FICHIER = "budget.txt";

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ArrayList<Depense> depenses = new ArrayList<>();
        double revenu = charger(depenses);   // rechargement au lancement
        int choix;

        do {
            System.out.println("\n=== MON BUDGET ===");
            System.out.println("1) Revenu   2) Ajouter dépense   3) Voir dépenses");
            System.out.println("4) Total & solde   5) Total par catégorie");
            System.out.println("6) Modifier   7) Supprimer   0) Quitter");
            System.out.print("Choix : ");
            choix = sc.nextInt(); sc.nextLine();

            switch (choix) {
                case 1 -> {
                    System.out.print("Revenu du mois : ");
                    revenu = sc.nextDouble(); sc.nextLine();
                    sauvegarder(revenu, depenses);
                }
                case 2 -> {
                    System.out.print("Description : "); String d = sc.nextLine();
                    System.out.print("Catégorie : ");   String cat = sc.nextLine();
                    System.out.print("Montant : ");
                    try {
                        double m = sc.nextDouble(); sc.nextLine();
                        depenses.add(new Depense(d, cat, m));
                        sauvegarder(revenu, depenses);
                        System.out.println("Ajoutée.");
                    } catch (Exception e) {
                        System.out.println("Montant invalide."); sc.nextLine();
                    }
                }
                case 3 -> afficherDepenses(depenses);
                case 4 -> {
                    double t = total(depenses);
                    System.out.println("Total : " + t + " €  |  Solde : " + (revenu
                }
                case 5 -> {
                    System.out.print("Catégorie : "); String cat = sc.nextLine();
                    System.out.println("Total " + cat + " : " + totalCategorie(depen
                }
```

```
                case 6 -> {
                    afficherDepenses(depenses);
                    System.out.print("N° à modifier : ");
                    int i = sc.nextInt() - 1; sc.nextLine();
                    if (i >= 0 && i < depenses.size()) {
                        System.out.print("Nouveau montant : ");
                        depenses.get(i).setMontant(sc.nextDouble()); sc.nextLine();
                        sauvegarder(revenu, depenses);
                    } else System.out.println("Invalide.");
                }
                case 7 -> {
                    afficherDepenses(depenses);
                    System.out.print("N° à supprimer : ");
                    int i = sc.nextInt() - 1; sc.nextLine();
                    if (i >= 0 && i < depenses.size()) {
                        depenses.remove(i);
                        sauvegarder(revenu, depenses);
                    } else System.out.println("Invalide.");
                }
                case 0 -> System.out.println("À bientôt !");
                default -> System.out.println("Choix inconnu.");
            }
        } while (choix != 0);
        sc.close();
    }

    static void afficherDepenses(ArrayList<Depense> depenses) {
        if (depenses.isEmpty()) { System.out.println("(aucune dépense)"); return; }
        for (int i = 0; i < depenses.size(); i++)
            System.out.println((i + 1) + ". " + depenses.get(i));
    }

    static double total(ArrayList<Depense> depenses) {
        double s = 0;
        for (Depense d : depenses) s += d.getMontant();
        return s;
    }

    static double totalCategorie(ArrayList<Depense> depenses, String cat) {
        double s = 0;
        for (Depense d : depenses)
            if (d.getCategorie().equalsIgnoreCase(cat)) s += d.getMontant();
        return s;
    }

    static void sauvegarder(double revenu, ArrayList<Depense> depenses) {
        try (FileWriter w = new FileWriter(FICHIER)) {
            w.write("REVENU;" + revenu + "\n");
            for (Depense d : depenses) w.write(d.versLigne() + "\n");
        } catch (IOException e) {
```

```
            System.out.println("Erreur de sauvegarde : " + e.getMessage());
        }
    }

    static double charger(ArrayList<Depense> depenses) {
        double revenu = 0;
        File f = new File(FICHIER);
        if (!f.exists()) return 0;
        try (Scanner lec = new Scanner(f)) {
            while (lec.hasNextLine()) {
                String[] p = lec.nextLine().split(";");
                if (p[0].equals("REVENU")) revenu = Double.parseDouble(p[1]);
                else if (p.length == 3)
                    depenses.add(new Depense(p[0], p[1], Double.parseDouble(p[2])));
            }
        } catch (Exception e) {
            System.out.println("Fichier illisible, on repart de zéro.");
        }
        return revenu;
    }
}
```

📌 À retenir — ce que ce projet démontre

Vous venez d'assembler une application complète : des **objets** (Depense) rangés dans une **collection** (ArrayList), pilotés par des **méthodes**, à travers un **menu en boucle**, avec **calculs**, **gestion d'erreurs** et **persistance sur fichier**. C'est exactement la structure d'un vrai logiciel — en plus petit.

🎯 Défi — faites évoluer le projet

- Afficher le total de *chaque* catégorie automatiquement (avec une `HashMap<String, Double>`).
- Alerter quand le solde devient négatif (« Attention, budget dépassé ! »).
- Ajouter une date à chaque dépense.
- Afficher un petit « graphique » en console (des `▮` proportionnels au montant par catégorie).
- Gérer plusieurs mois de budget.

Chapitre 23 — Évaluations et auto-évaluation

> 🎓 **À quoi servent ces évaluations ?**
>
> Trois évaluations jalonnent votre parcours pour mesurer vos acquis. Les réponses sont regroupées à la fin de ce chapitre : cherchez d'abord seul(e), c'est là que l'apprentissage se solidifie.

Évaluation 1 — Après les bases (chapitres 1 à 12)

1. (QCM) Qui exécute le bytecode ? a) le compilateur b) la JVM c) l'IDE

2. (Vrai/Faux) `int` peut stocker 3.14.

3. (Prédire) Qu'affiche `System.out.println(7 / 2);` ?

4. (Complétez) Pour comparer deux chaînes, on utilise `a._____(b)`.

5. (Trouvez l'erreur) `if (age = 18) { ... }`

6. (Prédire) Combien de tours fait `for (int i = 0; i < 5; i++)` ?

7. (QCM) Quelle boucle s'exécute au moins une fois ? a) for b) while c) do while

8. (Code) Écrivez une méthode `carre(int n)` qui renvoie n×n.

9. (Prédire) Pour `int[] t = {4, 8, 15};`, que vaut `t.length` ? Et `t[1]` ?

10. (Avec vos mots) Expliquez la différence entre `=` et `==`.

Évaluation 2 — Après la POO (chapitres 13 à 15)

1. (Avec vos mots) Qu'est-ce qu'une classe ? Qu'est-ce qu'un objet ?

2. (QCM) Le constructeur sert à… a) supprimer un objet b) initialiser un objet c) afficher

3. (Complétez) On rend un attribut invisible de l'extérieur avec `_____`.

4. (QCM) `extends` exprime… a) l'héritage b) une boucle c) une exception

5. (Vrai/Faux) `@Override` redéfinit une méthode héritée.

6. (Code) Créez une classe `Chat extends Animal` qui redéfinit `faireDuBruit()`.

7. (QCM) Pour une liste à taille variable, on utilise… a) un tableau b) une `ArrayList` c) un `int`

8. (Prédire) Après `liste.add("A"); liste.add("B"); liste.remove("A");`, que vaut `liste.size()` ?

9. (Avec vos mots) Qu'est-ce que le polymorphisme ?

10. (QCM) `this` désigne… a) la classe parente b) l'objet courant c) une variable globale

Évaluation finale — L'ensemble du cours

1. (QCM) Le fichier d'une classe publique `Compte` se nomme… a) `compte.java` b) `Compte.java` c) `Compte.class`

2. (Prédire) Qu'affiche : `String s = "Java"; System.out.println(s.toUpperCase());`

3. (Trouvez l'erreur) `int[] t = {1, 2, 3}; System.out.println(t[3]);`

4. (Code) Écrivez un `try/catch` qui protège une division de la division par zéro.

5. (Prédire) Que vaut `17 % 5` ?

6. (Code) Écrivez une boucle qui affiche les nombres pairs de 2 à 10.

7. (Avec vos mots) Pourquoi séparer une application en plusieurs classes/méthodes ?

8. (QCM) Pour conserver des données après l'arrêt du programme, on utilise… a) une variable b) un fichier c) un commentaire

9. (Code) Déclarez une `ArrayList<Integer>` et ajoutez-y les nombres 10 et 20.

10. (Défi de synthèse) Décrivez, en quelques lignes, comment vous structureriez un petit programme « carnet de notes » (attributs, classe, collection, fonctionnalités).

Grille d'auto-évaluation

Cochez honnêtement chaque compétence. Objectif : atteindre « Acquis » sur les fondamentaux avant de vous lancer seul dans des projets.

Je sais…	Pas encore	En cours	Acquis
Expliquer JDK, JVM, compilateur, bytecode	☐	☐	☐
Créer, compiler et lancer un programme	☐	☐	☐
Déclarer des variables et choisir le bon type	☐	☐	☐
Utiliser les opérateurs (dont `%` et logiques)	☐	☐	☐
Lire une saisie avec Scanner (et gérer le piège nextInt/nextLine)	☐	☐	☐
Écrire des conditions `if/else/switch`	☐	☐	☐
Écrire des boucles `for/while/do-while`	☐	☐	☐
Créer et appeler des méthodes avec paramètres et retour	☐	☐	☐
Manipuler des tableaux et des `ArrayList`	☐	☐	☐
Manipuler des chaînes (dont `.equals()`)	☐	☐	☐
Créer une classe, un objet, un constructeur	☐	☐	☐
Appliquer l'encapsulation (private, getters/setters)	☐	☐	☐
Utiliser l'héritage et comprendre le polymorphisme	☐	☐	☐
Gérer des exceptions avec `try/catch`	☐	☐	☐
Lire et écrire dans un fichier	☐	☐	☐
Lire un message d'erreur et déboguer	☐	☐	☐
Réaliser un mini-projet complet seul(e)	☐	☐	☐

💡 **Comment interpréter ?**

- **Majorité « Acquis » :** bravo, vous êtes prêt(e) pour des projets personnels et pour approfondir (voir la feuille de route, chapitre 26).
- **Plusieurs « En cours » :** normal ! Refaites les exercices des chapitres concernés.
- **Des « Pas encore » :** reprenez le chapitre en tapant chaque exemple — sans se juger, on apprend en pratiquant.

Corrigés des évaluations

💡 **Évaluation 1 — Après les bases**

1-b · 2-Faux (int = entiers seulement) · 3 : `3` (division entière) · 4 : `equals` · 5 : `=` est une affectation ; il faut `==` · 6 : 5 tours · 7-c · 8 : `static int carre(int n){ return n*n; }` · 9 : `length` = 3, `t[1]` = 8 · 10 : `=` range une valeur (affectation) ; `==` teste l'égalité (renvoie true/false).

💡 **Évaluation 2 — Après la POO**

1 : la classe est un modèle/plan, l'objet une instance concrète créée à partir de ce modèle · 2-b · 3 : `private` · 4-a · 5-Vrai · 6 : `class Chat extends Animal { @Override void faireDuBruit(){ System.out.println("Miaou"); } }` · 7-b · 8 : `1` · 9 : traiter des objets de types différents via un type commun, chacun exécutant sa propre version d'une méthode · 10-b.

💡 Évaluation finale

1-b · 2 : `JAVA` · 3 : index 3 hors limites (les index valides sont 0,1,2) → `ArrayIndexOutOfBoundsException` · 4 : `try { int r = a/b; } catch (ArithmeticException e) { System.out.println("Division par zéro"); }` · 5 : `2` · 6 : `for (int i = 2; i <= 10; i += 2) System.out.println(i);` · 7 : pour rendre le code lisible, réutilisable et facile à corriger (chaque partie a un rôle) · 8-b · 9 : `ArrayList<Integer> l = new ArrayList<>(); l.add(10); l.add(20);` · 10 : une classe `Note` (matière, valeur), une `ArrayList<Note>`, et des méthodes ajouter/afficher/calculer la moyenne — architecture identique à nos mini-projets.

Chapitre 24 — Plan d'apprentissage sur 8 semaines

> 🎓 **Comment utiliser ce plan ?**
>
> Un rythme réaliste de **5 à 7 heures par semaine** permet de terminer le livre en 8 semaines sans se décourager. Adaptez à votre disponibilité : mieux vaut 1 h régulière chaque jour qu'un marathon le dimanche.

> 💡 **Conseil — la règle des 3 gestes hebdomadaires**
>
> Chaque semaine : **(1)** lisez et tapez les exemples, **(2)** faites tous les exercices, **(3)** avancez un projet. Lire sans coder ne suffit pas.

Semaine 1 — Découverte et installation

Chapitres	Chapitres 1 à 3
Exercices	3.1
Projet	—
Temps conseillé	5 h
Objectifs	Comprendre JDK/JVM/compilateur ; installer l'environnement ; écrire et lancer « Hello World ».

Semaine 2 — Variables, opérateurs, saisie

Chapitres	Chapitres 4 à 7
Exercices	5.1 à 5.3, 6.1 à 6.3, 7.1 à 7.3
Projet	—
Temps conseillé	6 h
Objectifs	Manipuler variables et types ; maîtriser les opérateurs ; lire une saisie utilisateur.

Semaine 3 — Conditions et boucles

Chapitres	Chapitres 8 et 9
Exercices	8.1 à 8.4, 9.1 à 9.5
Projet	**Mini-projet 2 (Nombre mystère)**
Temps conseillé	6 h
Objectifs	Écrire des programmes qui décident et répètent ; réaliser un premier jeu interactif.

Semaine 4 — Méthodes, tableaux, chaînes

Chapitres	Chapitres 10 à 12
Exercices	10.1 à 10.4, 11.1 à 11.4, 12.1 à 12.3
Projet	**Mini-projets 1 & 3 (Calculatrice, Convertisseur)** + Évaluation 1
Temps conseillé	7 h
Objectifs	Découper un programme en méthodes ; manipuler tableaux et chaînes ; valider les bases.

Semaine 5 — Programmation orientée objet

Chapitres	Chapitres 13 et 14
Exercices	13.1 à 13.3, 14.1 et 14.2
Projet	—
Temps conseillé	6 h
Objectifs	Créer classes et objets ; appliquer encapsulation, héritage, polymorphisme.

Semaine 6 — Collections et gestion des erreurs

Chapitres	Chapitres 15 et 16
Exercices	15.1 à 15.3, 16.1 à 16.3
Projet	**Mini-projet 4 (Gestionnaire de tâches)** + Évaluation 2
Temps conseillé	7 h
Objectifs	Utiliser `ArrayList` ; gérer les exceptions ; assembler une petite application à objets.

Semaine 7 — Fichiers, bonnes pratiques, Git, débogage

Chapitres	Chapitres 17 à 20
Exercices	17.1 à 17.3, 20.1
Projet	**Mini-projet 5 (Gestionnaire de contacts)**
Temps conseillé	6 h
Objectifs	Persister des données ; adopter les bonnes pratiques ; débuter avec Git ; savoir déboguer.

Semaine 8 — Projet final

Chapitres	Chapitre 22 (+ révisions ciblées)
Exercices	Défis d'amélioration des mini-projets
Projet	**Projet final (Budget personnel), étapes 1 à 6** + Évaluation finale
Temps conseillé	7 h
Objectifs	Réaliser une application complète de bout en bout ; remplir la grille d'auto-évaluation.

📌 **À retenir**

Après ces 8 semaines, vous ne serez pas « expert » — personne ne l'est en 2 mois — mais vous serez **autonome** : capable de lire du code Java, de comprendre une erreur, et de construire vos propres petits programmes. C'est exactement l'objectif.

💡 **Conseil — si vous avez pris du retard**

Aucun problème : ce plan est indicatif. L'important n'est pas la vitesse mais la régularité. Reprenez où vous en êtes, sans culpabiliser. Chaque ligne tapée compte.

Chapitre 25 — Glossaire

Les termes essentiels du livre, expliqués simplement, par ordre alphabétique.

Terme	Définition simple
Algorithme	Suite d'étapes logiques pour résoudre un problème, indépendamment du langage. Ex. : la « recette » pour trouver le plus grand nombre d'une liste.
Argument	Valeur réelle passée à une méthode lors de son appel. Dans `additionner(3, 5)`, 3 et 5 sont les arguments.
Attribut	Donnée appartenant à un objet (ex. le `prenom` d'une `Personne`). Aussi appelé « champ ».
Bug	Erreur dans un programme provoquant un comportement inattendu. Le corriger s'appelle « déboguer ».
Bytecode	Code intermédiaire (`.class`) produit par le compilateur et exécuté par la JVM.
Classe	Modèle (plan) décrivant des attributs et des méthodes. Sert à créer des objets.
Collection	Objet contenant un groupe d'éléments avec des outils pour les gérer (ex. `ArrayList`).
Compilation	Traduction du code source (`.java`) en bytecode (`.class`) par le compilateur `javac`.
Concaténation	Coller des chaînes bout à bout avec l'opérateur `+`.
Constante	Variable dont la valeur ne change jamais, déclarée avec `final`.
Constructeur	Méthode spéciale (même nom que la classe) qui initialise un objet à sa création.
Encapsulation	Cacher les attributs (`private`) et n'y accéder que par des méthodes contrôlées (getters/setters).
Exception	Incident survenant à l'exécution (division par zéro…), qui interrompt le programme s'il n'est pas géré (`try/catch`).
Héritage	Mécanisme par lequel une classe enfant récupère les attributs et méthodes d'une classe parente (`extends`).
IDE	Logiciel tout-en-un pour programmer (éditeur, exécution, débogueur). Ex. : IntelliJ IDEA.
Index	Numéro de position d'un élément dans un tableau ou une liste. On compte à partir de 0.
Instance	Autre mot pour « objet » : une réalisation concrète d'une classe.
Itération	Un « tour » de boucle.
Java	Langage de programmation portable et robuste, exécuté par la JVM. Version utilisée ici : Java 21 LTS.

JDK	*Java Development Kit* : la boîte à outils complète du développeur (compilateur + JVM + bibliothèques).
JVM	*Java Virtual Machine* : le programme qui exécute le bytecode ; assure la portabilité de Java.
Méthode	Bloc d'instructions nommé, que l'on peut appeler ; peut recevoir des paramètres et renvoyer une valeur.
Objet	Réalisation concrète d'une classe, créée avec `new`. Possède ses propres valeurs d'attributs.
Opérateur	Symbole effectuant une opération : `+ - * /`, comparaisons, `&& || !`.
Package	Sous-dossier organisant les classes par thème (ex. `fr.jonathan.debut`).
Paramètre	Variable déclarée dans la définition d'une méthode, qui recevra un argument.
Persistance	Conservation durable de données (sur disque), au-delà de l'exécution du programme.
Polymorphisme	Capacité de traiter des objets de types différents via un type commun, chacun exécutant sa propre version d'une méthode.
POO	*Programmation orientée objet* : façon d'organiser le code autour de classes et d'objets.
Portée (scope)	Zone du code où une variable existe et est utilisable.
Référence	« Adresse » qui indique où trouver un objet en mémoire. `==` compare des références ; `.equals()` compare des contenus.
Syntaxe	Règles d'écriture d'un langage (accolades, points-virgules…).
Type	Nature d'une valeur (`int`, `double`, `boolean`, `String`…). Détermine ce qu'on peut y stocker.
Variable	Espace mémoire nommé contenant une valeur réutilisable.

Chapitre 26 — Conclusion et feuille de route

Félicitations !

Vous êtes parti(e) de zéro — sans savoir ce qu'était une variable — et vous voilà capable de créer des applications console complètes en Java. C'est un vrai accomplissement. Prenez un instant pour le mesurer.

Vous savez désormais :

- comment un programme Java passe du code source à l'exécution ;
- manipuler variables, opérateurs, conditions et boucles ;
- structurer votre code avec des méthodes et des classes ;
- utiliser tableaux, chaînes et collections ;
- gérer les erreurs, lire/écrire des fichiers et déboguer ;
- mener un projet complet du début à la fin.

> 📌 **À retenir — le plus important**
>
> La compétence numéro un que vous avez acquise n'est pas « connaître Java par cœur » : c'est **savoir apprendre en pratiquant** — taper du code, lire les erreurs, chercher, corriger, recommencer. C'est cette compétence qui fait les développeurs.

Feuille de route pour continuer

Voici un parcours réaliste pour progresser après ce livre.

Étape 1 — Consolider (1 à 2 mois)

- **Recodez** les mini-projets sans regarder les solutions.
- **Inventez** vos propres petits programmes : quiz, gestionnaire de mots de passe simple, tirage au sort, mini-jeu.
- Relevez tous les « Défis » du livre.

Étape 2 — Approfondir le langage

- **Interfaces** et classes **abstraites** (la suite naturelle de l'héritage).
- **Généricité** avancée, **énumérations** (`enum`), le **records** (classes de données concises, modernes).
- **Expressions lambda** et l'**API Stream** (traiter des collections de façon élégante).
- Les autres collections : `LinkedList` , `TreeMap` , files et piles.

Étape 3 — Sortir de la console

- **JavaFX** : créer des applications avec fenêtres, boutons et champs.
- **Bases de données** avec SQL et JDBC (remplacer les fichiers texte par une vraie base).
- **Spring Boot** : le framework de référence pour créer des applications web et des API en Java.

Étape 4 — Devenir un vrai développeur

- Approfondir **Git/GitHub** et publier vos projets (votre « portfolio »).
- Apprendre les **tests automatisés** (JUnit).
- Découvrir **Maven** ou **Gradle** (gérer les dépendances et les projets).
- Lire du code des autres, contribuer à un projet open source, rejoindre une communauté.

> 💡 **Conseil — ressources pour continuer**
>
> - **Documentation officielle Oracle** et **tutoriels Dev.java** (le site officiel de la communauté Java).
> - Pratiquez sur des plateformes d'exercices (katas de programmation).
> - Un projet personnel qui vous tient à cœur est le meilleur moteur d'apprentissage.

> 📌 **Le mot de la fin**
>
> Programmer, c'est une aventure sans fin — et c'est ce qui la rend passionnante. Il y aura toujours quelque chose de nouveau à apprendre, un problème à résoudre, un projet à bâtir. Vous avez maintenant les fondations. Le reste, c'est de la pratique, de la curiosité et de la persévérance.
>
> **Bon code, et bienvenue dans le monde des développeurs ! 🚀**

Apprendre le Java par Jonathan — Formation complète et pratique pour débutants
Édition 2026 · Basé sur Java 21 LTS