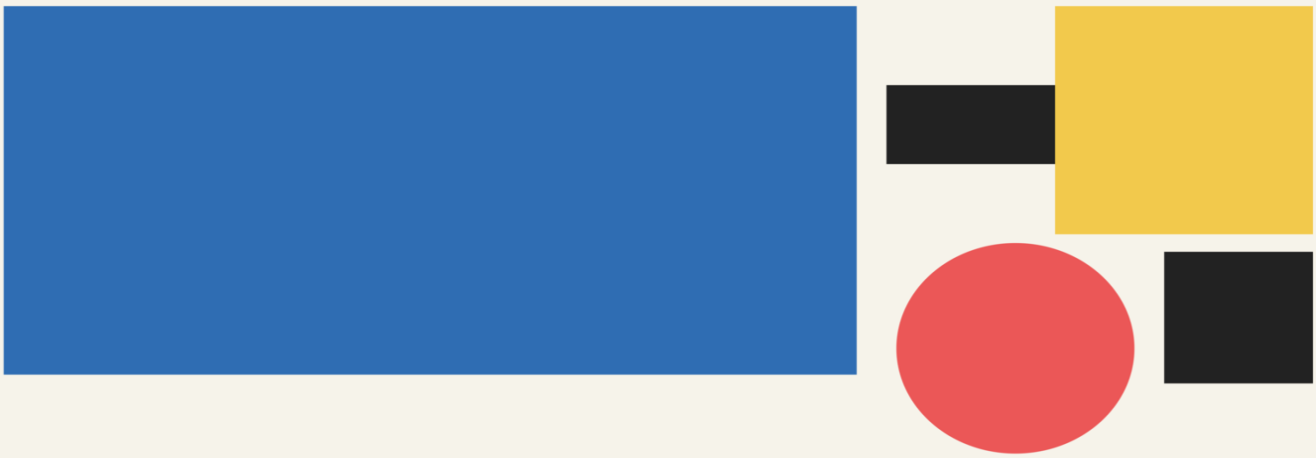
Ars Digitale
Engineering Notes Series

JAVA 21 ENGINEERING GUIDE



Langage Java, architecture
et bonnes pratiques

Alessandro Fabri

Édition 2026
info@ars-digitale.com

Guide d’ingénierie Java 21

Alessandro Fabri

Ars Digitale

Guide d’ingénierie Java 21

Un guide d’étude structuré pour le développement Java moderne et la certification

Alessandro Fabri

2026 Edition

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

Copyright

Version: 1.0
Langue : Français

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

2026 Edition

Préface

Ce guide est conçu comme une feuille de route structurée et pratique pour l'étude de Java 21, avec un accent particulier sur la clarté, la rigueur et la compréhension durable.

Le contenu est organisé en modules, allant des fondements du langage jusqu'aux API, à la concurrence, aux entrées/sorties et au Java Platform Module System.

Chaque chapitre est pensé à la fois comme une étape d'un parcours progressif et comme une référence technique autonome.

À propos de ce livre

Ce livre a été conçu comme un compagnon d'apprentissage technique pour Java 21.

Il vise à combiner :

- une progression structurée
- des explications concises
- une précision technique
- une clarté adaptée à la certification
- une utilité durable comme ouvrage de référence

L'édition EPUB est optimisée pour la lecture numérique et la navigation par chapitres.

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

🌎 **Langue:** English | Italiano | Français

Index du cours (Java 21)

Module 00 — Prerequisites & Setup

Module 01 — Java Language Basics

Module 02 — Control Flow

Module 03 — Core Standard APIs

Module 04 — Object-Oriented Fundamentals

Module 05 — Functional Programming

Module 06 — Collections Framework

- 22. Introduction au Framework des Collections
- 23. Opérations Partagées des Collections & Égalité
- 24. Comparable, Comparator & Tri en Java
- 25. L'API List
- 26. Set API
- 27. API Queue & Deque
- 28. Map API
- 29. Collections Séquencées & Map Séquencées

Module 07 — Concurrency and Threads

- 30. Thread Java – Fondamentaux et Modèle d'Exécution
- 31. Java Concurrency APIs

Module 08 — Java I/O and NIO

- 32. Fondamentaux des fichiers et des chemins
- 33. APIs des fichiers et des chemins
- 34. Streams I/O Java
- 35. API Java d'E/S (Legacy et NIO)
- 36. Interagir avec l'Utilisateur (Flux E/S Standard)

Module 09 — Java Platform Module System (JPMS)

- 37. Java Platform Module System (JPMS)
- 38. Compiler, Empaqueter et Exécuter des Modules
- 39. Services en JPMS (Le Modèle ServiceLoader)

Prerequisites & Setup

Prerequisite material for the course

This is all the prerequisite material you will need for the course

DOCUMENTATION

- **Java 21 APIs** - Java 21 API Specification
- **Eclipse shortcuts** - Eclipse IDE shortcuts

EDITOR

- **Eclipse IDE** - Download Eclipse here

PANDOC

- **Pandoc** - Download Pandoc here

◀ Index du cours (Java 21) | ▲ Index | Eclipse main shortcuts ▶

Eclipse main shortcuts

WIN	APPLE	DESCRIPTION
<kbd>Ctrl</kbd> + 3	<kbd>⌘</kbd> + 3	Go to quick access search for available views, actions, wizards, menus and more
<kbd>Alt</kbd> + <kbd>⇧</kbd> + Q Q	<kbd>⌥</kbd> + <kbd>⌘</kbd> + Q Q	Show all available views and select one or more to open
F2	F2	Show Javadoc for the selected element
F3 or <kbd>Ctrl</kbd> + Left click	F3 or <kbd>⌘</kbd> + Left click	In a code editor, go to the declaration of the selected symbol
F4	F4	Show selected symbol in the “Type Hierarchy” view
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + T	<kbd>⌘</kbd> + <kbd>⇧</kbd> + T	Open dialog to search for a type (class, interface, enum)
<kbd>Ctrl</kbd> + Alt + H	^ + <kbd>⌥</kbd> + H	Open selected callable symbol in the “Call Hierarchy” view
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + G	<kbd>⇧</kbd> + <kbd>⌘</kbd> + G	Search workspace for all references to the symbol
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + R	<kbd>⌘</kbd> + <kbd>⇧</kbd> + R	Open dialog to search resources (e.g. text files) by filename
<kbd>Ctrl</kbd> + F	<kbd>⌘</kbd> + F	Find/replace in the current file
<kbd>Ctrl</kbd> + H	<kbd>⌘</kbd> + H	Find/replace in current file, project, or workspace
<kbd>Ctrl</kbd> + L	<kbd>⌘</kbd> + L	Go to a line number
<kbd>Ctrl</kbd> + .	<kbd>⌘</kbd> + .	Jump to next occurrence, warning or error
<kbd>Ctrl</kbd> + ,	<kbd>⇧</kbd> + <kbd>⌘</kbd> + .	Jump to previous occurrence, warning or error
<kbd>Ctrl</kbd> + D	<kbd>⌘</kbd> + D	Delete line at cursor position
<kbd>Alt</kbd> + ↑ or <kbd>Alt</kbd> + ↓	<kbd>⌥</kbd> + ↑ or <kbd>⌥</kbd> + ↓	Move current line one line above or one line below
<kbd>Ctrl</kbd> + Space	<kbd>⌘</kbd> + Space	Open content assist dialog (based on current context)
Type "main", "if", "for", "while", "do", "syso" + <kbd>Ctrl</kbd> + Space	(same as before) + <kbd>⌘</kbd> + Space	Autocomplete element
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + F		Format code
<kbd>Alt</kbd> + <kbd>⇧</kbd> + Z	<kbd>⌥</kbd> + <kbd>⌘</kbd> + Z	Toggle Try Catch and other predefined blocks of code
<kbd>Alt</kbd> + <kbd>⇧</kbd> + A	<kbd>⌥</kbd> + <kbd>⌘</kbd> + A	Toggle block / column selection in the current text editor

WIN	APPLE	DESCRIPTION
<kbd>Alt</kbd> + <kbd>⇧</kbd> + R	<kbd>⌥</kbd> + <kbd>⌘</kbd> + R	Rename (variable, field, method, class...)
<kbd>Alt</kbd> + <kbd>⇧</kbd> + S	<kbd>⌥</kbd> + <kbd>⌘</kbd> + S	Show advanced editing operations for current selection
<kbd>Alt</kbd> + <kbd>⇧</kbd> + T	<kbd>⌥</kbd> + <kbd>⌘</kbd> + T	Show available refactoring operations for current selection
<kbd>Ctrl</kbd> + 1	<kbd>⌘</kbd> + 1	Show all possible fixes for a problem (on a text element with a problem marker, or in the problem view)
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + C	<kbd>⌘</kbd> + /	Add/Remove line comment
<kbd>Ctrl</kbd> + <kbd>⇧</kbd> + /	^ + <kbd>⌘</kbd> + /	Add/Remove block line comment

---

◀ Prerequisite material for the course | ▲ Index | 1. Blocs Syntaxiques Fondamentaux ▶

Module 01

Java Language Basics

1. Blocs Syntaxiques Fondamentaux

Table des matières

Tout système logiciel ou programme informatique est composé d'un ensemble de **données** et d'un ensemble **d'opérations** appliquées à ces données afin de produire un résultat.

Plus formellement :

> Un programme informatique consiste en un ensemble de structures de données représentant l'état du système, accompagné d'algorithmes qui spécifient les opérations à effectuer sur cet état pour produire des sorties.

Ce document décrit une **hiérarchie d'abstractions** : les *blocs élémentaires* qui, combinés en structures de plus en plus complexes, forment un logiciel.

La séquence est présentée dans un **ordre croissant de complexité**, avec des définitions générales (informatique) et des références à Java.

1.1 Valeur

- **Définition :** Entité abstraite représentant une information (nombre, caractère, booléen, chaîne, etc.).
- **Théorie :** Une valeur appartient à un domaine (ensemble) mathématique, comme $\mathbb{N}$ pour les nombres naturels ou Σ^* pour les chaînes de caractères.
- **Exemple (abstrait) :** le nombre quarante-deux, la valeur de vérité *true*, le caractère « a ».

Exemple Java (valeurs) :

```java
// Voici des valeurs :
42        // une valeur int
true      // une valeur boolean
'a'       // une valeur char
"Hello"   // une valeur String
```

1.2 Littéral

- **Définition :** Un **littéral** est la notation concrète dans le code source qui désigne directement une valeur fixe.
- **En Java :** `42`, `'a'`, `true`, `"Hello"`.
- **Théorie :** Un littéral est de la **syntaxe**, tandis que la valeur correspond à sa **sémantique**.
- **Remarque :** Les littéraux sont la manière la plus courante d'introduire des valeurs dans les programmes.

Exemple Java (littéraux) :

```java
int answer = 42;          // 42 est un littéral int
char letter = 'a';        // 'a' est un littéral char
boolean flag = true;      // true est un littéral boolean
String msg = "Hello";     // "Hello" est un littéral String
```

1.3 Identifiant

- **Définition :** Nom symbolique qui associe une valeur (ou une structure) à une étiquette lisible.
- **En Java :**
  - **Identifiants définis par l'utilisateur :** choisis par le programmeur pour nommer variables, méthodes, classes, etc.
    Exemples : `x`, `counter`, `MyClass`, `calculateSum`.
  - **Mots-clés (réservés) :** noms prédéfinis réservés par le langage Java et qui ne peuvent pas être redéfinis.
    Exemples : `class`, `public`, `static`, `if`, `return`.

> **Note**
>
> Les identifiants doivent respecter les règles de nommage de Java : voir *Java Naming Rules*.

- **Théorie :** Fonction de liaison (binding) : relie un nom à une valeur ou une ressource.

Exemple Java (identifiants) :

```java
int counter = 0;          // counter est un identifiant (nom de variable)
String userName = "Bob";  // userName est un identifiant
class MyService { }       // MyService est un identifiant de classe
```

1.4 Variable

- **Définition :** « Case mémoire » étiquetée par un identifiant, qui peut contenir et changer de valeur.
- **En Java :** `int counter = 0; counter = counter + 1;`.
- **Théorie :** État mutable susceptible d'évoluer pendant l'exécution.

Exemple Java (variable qui évolue dans le temps) :

```java
int counter = 0;       // variable initialisée
counter = counter + 1; // variable mise à jour
counter++;             // autre mise à jour (post-incrément)
```

1.5 Type

- **Définition :** Un type est un ensemble de valeurs et un ensemble d'opérations autorisées sur ces valeurs.
- **En Java :**

- **Types primitifs (simples) :** représentent directement des valeurs de base. Exemples : `int`, `double`, `boolean`, `char`, `byte`, `short`, `long`, `float`.
- **Types référence :** représentent des références (pointeurs) vers des objets en mémoire. Exemples : `String`, tableaux (par ex. `int[]`), classes, interfaces et types définis par l'utilisateur.

> **Note**
>
> Voir *Java Data Types*.

- **Théorie :** Un système de types est l'ensemble des règles qui associent des ensembles de valeurs et des opérations admissibles.

Exemple Java (types) :

```
int age = 30;           // type int
double price = 9.99;    // type double
boolean active = true;  // type boolean
String name = "Alice";  // type référence (classe String)
```

1.6 Opérateur

- **Définition : Symbole ou mot-clé** qui effectue un calcul ou une action sur un ou plusieurs opérandes.
- **Rôle :** Les opérateurs combinent valeurs, variables et expressions pour produire de nouvelles valeurs ou modifier l'état du programme.
- **En Java :**

> **Note**
>
> Voir *Java Operators*.

- **Théorie :** Les opérateurs définissent les calculs autorisés sur les types ; avec les valeurs et variables, ils forment les **expressions**.

Exemple Java (opérateurs en contexte) :

```
int a = 5 + 3;          // + arithmétique
boolean ok = a > 3;     // > comparaison
ok = ok && true;        // && logique
a += 2;                 // += affectation composée
int sign = (a >= 0) ? 1 : -1; // ?: ternaire
```

1.7 Expression

- **Définition :** Combinaison de valeurs, littéraux, variables, opérateurs et fonctions produisant une nouvelle valeur.
- **En Java :** `x + 3`, `Math.sqrt(25)`, `"Hello" + " world"`.
- **Théorie :** Arbre syntaxique qui, une fois évalué, donne un résultat.

Exemple Java (expressions) :

```
int x = 10;
int y = x + 3;              // x + 3 est une expression
double r = Math.sqrt(25);   // Math.sqrt(25) est une expression
String msg = "Hello" + " "; // "Hello" + " " est une expression
msg = msg + "world";        // msg + "world" est une autre expression
```

1.8 Instruction

- **Définition :** Unité d'exécution qui modifie l'état ou contrôle le flot d'exécution.
- **En Java :** `x = x + 1;`, `if (x > 0) { ... }`.
- **Théorie :** Séquence d'actions qui ne renvoie pas de valeur en tant que résultat de l'instruction elle-même, mais modifie la configuration de la machine abstraite.

Exemple Java (instructions) :

```java
int x = 0;                // déclaration (instruction de définition)
x = x + 1;                // instruction d'affectation

if (x > 0) {              // instruction if
    System.out.println("Positive");
}
```

1.9 Bloc de code

- **Définition :** Ensemble d'instructions délimitées formant une unité exécutable.
- **En Java :** `{ int y = 5; x = x + y; }`.
- **Théorie :** Composition séquentielle d'instructions, avec des règles de *portée* (visibilité).

Exemple Java (bloc de code et portée) :

```java
int x = 10;

{
    int y = 5;        // y n'est visible qu'à l'intérieur de ce bloc
    x = x + y;        // OK : x est visible ici
}

// y n'est plus visible ici
// x est toujours visible ici
```

1.10 Fonction / Méthode

- **Définition :** Séquence d'instructions encapsulée, identifiée par un nom, pouvant recevoir des entrées (paramètres) et renvoyer une sortie (valeur).
- **En Java :**

```java
int square(int n) {
    return n * n;
}
```

- **Théorie :** Application (mapping) entre domaines d'entrée et de sortie, avec un corps opérationnel.

Exemple d'utilisation Java :

```java
int result = square(5); // result = 25
```

1.11 Classe / Objet

- **Définition :**
    - **Classe :** description abstraite d'un ensemble d'objets (état + comportement).
    - **Objet :** instance concrète de la classe.
- **En Java :**

```java
class Point {
    int x, y;

    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
}

Point p = new Point();    // p est un objet (instance de Point)
p.move(1, 2);             // appel de méthode sur l'objet
```

- **Théorie :** Abstraction d'un *ADT* (Abstract Data Type, type de donnée abstrait).

1.12 Module / Package

- **Définition :** Regroupement logique de classes, fonctions et ressources partageant un objectif commun.
- **En Java :** `package java.util;` → regroupe des utilitaires.
- **Théorie :** Mécanisme d'organisation et de réutilisation, réduisant la complexité.

Exemple Java (package) :

```java
package com.example.app;

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

1.13 Programme

- **Définition :** Ensemble cohérent de modules, classes et fonctions qui, lorsqu'il est exécuté sur une machine, réalise un comportement global.
- **En Java :** La méthode `main` et tout ce qu'elle invoque.
- **Théorie :** Spécification de transformations d'entrées en sorties sur une machine abstraite.

Exemple Java (programme minimal) :

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java 21!");
    }
}
```

1.14 Système

- **Définition :** Ensemble de programmes coopérant qui interagissent avec des ressources externes (utilisateur, réseau, périphériques).
- **Exemple :** Une plateforme Java d'entreprise avec base de données, services REST, interface utilisateur.
- **Théorie :** Architecture complexe de composants logiciels et matériels.

Exemple (conceptuel) :

- Un backend Java (service Spring Boot)
- Une base de données (PostgreSQL)
- Une application web front-end
- Des services externes (API REST, files de messages)

Ensemble, ils forment un **système**.

1.15 📌 Résumé sous forme d'échelle croissante

`Valeur → Littéral → Identifiant → Variable → Type → Opérateur → Expression → Instruction → Bloc de code → Fonction/Méthode → Classe/Objet → Module/Package → Programme → Système`

Cette échelle montre comment de petites unités conceptuelles sont combinées en structures de plus en plus grandes et complexes.

1.16 📊 Diagramme hiérarchique (ASCII)

Description : Ce diagramme ASCII montre la relation hiérarchique entre les blocs de construction, du plus complexe (Système) au plus simple (Valeur et sa forme concrète, le Littéral).

```
Système
└── Programme
    └── Module / Package
        └── Classe / Objet
            └── Fonction / Méthode
                └── Bloc de code
                    └── Instruction
                        └── Expression
                            └── Opérateur
                                └── Type
                                    └── Variable
                                        └── Identifiant
                                            └── Littéral
                                                └── Valeur
```

1.17 📊 Diagramme hiérarchique (Mermaid)

Description : Le diagramme Mermaid rend la même hiérarchie sous forme d'arbre descendant. Il met en évidence qu'un Littéral est la forme syntaxique d'une Valeur.

```
graph TD
    A[Système]
    A --> B[Programme]
    B --> C[Module / Package]
    C --> D[Classe / Objet]
    D --> E[Fonction / Méthode]
    E --> F[Bloc de code]
    F --> G[Instruction]
    G --> H[Expression]
    H --> H2[Opérateur]
    H2 --> I[Type]
    I --> J[Variable]
    J --> K[Identifiant]
    K --> L[Littéral]
    L --> M[Valeur]
```

◀ Eclipse main shortcuts | ▲ Index | 2. Blocs de base du langage Java ▶

2. Blocs de base du langage Java

Table des matières

Ce chapitre présente les éléments structurels essentiels d'un programme Java : `classes`, `méthodes`, `commentaires`, `modificateurs d'accès`, `packages`, la méthode `main` et les outils de base en ligne de commande (`javac` et `java`).

Ce sont les éléments minimaux nécessaires pour écrire, compiler, organiser et exécuter du code Java à l'aide du JDK (Java Development Kit) — sans utiliser aucun IDE (Integrated Development Environment).

2.1 Définition de classe

Une `class` Java est le bloc fondamental d'un programme Java.

Une `classe` représente un **type de donnée défini par l'utilisateur**, constitué d'un ensemble de données internes (`fields`) et des opérations pouvant agir sur celles-ci (`methods`).

Une `class` est un **plan** (blueprint), tandis que les `objects` sont des **instances concrètes** créées à l'exécution.

Une classe Java est composée de deux éléments principaux, appelés ses **membres**:

- **Fields** (ou variables): représentent les données qui définissent l'état de ce nouveau type.
- **Methods** (ou fonctions): représentent les opérations qui peuvent être effectuées sur ces données.

Certains membres peuvent être déclarés avec le mot-clé **static**.

Un membre static appartient à la classe elle-même, et non aux objets créés à partir d'elle.

Cela signifie que :

- il existe une seule copie partagée entre toutes les instances
- il peut être utilisé sans créer un objet de la classe
- il est chargé en mémoire lorsque la classe est chargée par la JVM

Les membres non static (appelés **d'instance**) appartiennent en revanche aux objets individuels et chaque instance possède sa propre copie.

Normalement, chaque classe est définie dans son propre fichier "**.java**" ; par exemple, une classe nommée `Person` sera définie dans le fichier correspondant `Person.java`.

Toute classe définie indépendamment dans son propre fichier source est appelée **top-level class**.

Une telle classe ne peut être déclarée que `public` ou avec le modificateur d'accès par défaut (`package-private`, c'est-à-dire sans modificateur explicite).

Un seul fichier peut cependant contenir plusieurs définitions de classe.
Dans ce cas, une seule classe peut être déclarée `public`, et le nom du fichier doit correspondre à cette classe.

Les **nested classes**, c'est-à-dire les classes déclarées à l'intérieur d'une autre classe, peuvent utiliser n'importe quel modificateur d'accès : `public`, `protected`, `private`, `default` (package-private).

- Exemple :

```
public class Person {

    // This is a comment: explains the code but is ignored by the compiler. See section below.

    // Field → defines data/state
    String personName;

    // Method → defines behavior (this one take a parameter, newName, in input but does not re
    void setPersonName(String newName) {
        personName = newName;
    }

    // Method → defines behavior  (this one does not take parameters in input but does return
    String getPersonName(){
        return personName;
    }
}
```

> **Note**
>
> Dans sa forme la plus simple, on pourrait théoriquement avoir une classe sans méthode et sans field. Une telle classe se compilerait, mais aurait très peu de sens pratique.

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	détermine quelles autres classes peuvent utiliser ou voir cet élément	Mandatory (lorsqu'il est absent, l'accès est par défaut package-private)
class	Keyword	Déclare un type de classe.	Mandatory
Person	Class name (identifier)	Le nom de la classe.	Mandatory
personName	Field name (identifier)	Stocke le nom de la personne.	Optional
String	Type / Keyword	Type du field `personName` et du paramètre `newName`.	Mandatory
setPersonName, getPersonName	Method names (identifier)	nomment un comportement de la classe.	Optional
newName	Parameter name (identifier)	paramètre passé à la méthode `setPersonName`.	Mandatory (si la méthode a besoin d'un paramètre)
return	Keyword	Quitte une méthode et renvoie une valeur.	Mandatory (dans les méthodes avec type de retour non void)
void	Return Type / Keyword	Indique que la méthode ne renvoie aucune valeur.	Mandatory (si la méthode ne renvoie pas de valeur)

> **Note**
>
> Mandatory = requis par la syntaxe Java, Optional = non requis par la syntaxe ; dépend du design.

2.2 Commentaires

Les commentaires ne sont pas du code exécutable : ils **expliquent** le code mais sont ignorés par le compilateur.

En Java, il existe 3 types de commentaires : - Single-line (`//`) - Multi-line (`/* ... */`) - Javadoc (`/** ... */`)

Un **single-line comment** commence par 2 barres obliques : tout le texte qui suit sur la même ligne est ignoré par le compilateur.

- Exemple :

```
// This is a single-line comment. It starts with 2 slashes and ends at the end of the line.
```

Un **multiline comment** inclut tout ce qui se trouve entre les symboles `/*` et `*/`.

- Exemple :

```
/*
 * This is a multi-line comment.
 * It can span multiple lines.
 * All the text between its opening and closing symbols is ignored by the compiler.
 *
 */
```

Un **commentaire Javadoc** est similaire à un **multiline comment**, sauf qu'il commence par `/**` : tout le texte entre les symboles d'ouverture et de fermeture est traité par l'outil Javadoc pour générer la documentation d'API.

```
/**
 * This is a Javadoc comment
 *
 * This class represents a Person.
 * It stores a name and provides methods
 * to set and retrieve it.
 *
 * <p>Javadoc comments can include HTML tags,
 * and special annotations like @param and @return.</p>
 */
```

> **Warning**
>
> En Java, chaque block comment doit être correctement fermé avec `*/`.

- Exemple :

```
/* valid block comment */
```

est correct, mais

```
/* */ */
```

provoquera une erreur de compilation, car les deux premiers symboles font partie du commentaire, mais le dernier non. Le symbole supplémentaire `*/` n'est pas une syntaxe valide et le compilateur se plaindra.

2.3 Modificateurs d'accès

En Java, un **modificateur d'accès** (access modifier) est un mot-clé qui spécifie la visibilité (ou accessibilité) d'une **classe**, d'une **méthode** ou d'un **champ**. Il détermine quelles autres classes peuvent utiliser ou voir cet élément.

> **Note**
>
> **Table des modificateurs d'accès disponibles en Java**

Token / Identifier	Category	Meaning	Optional?
public	Keyword / access modifier	Visible depuis n'importe quelle classe, dans n'importe quel package	Oui
no modifier (default)	Keyword / access modifier	Visible uniquement à l'intérieur du même package	Oui
protected	Keyword / access modifier	Visible dans le même package et par les sous-classes (même dans d'autres packages)	Oui
private	Keyword / access modifier	Visible uniquement à l'intérieur de la même classe	Oui

> **Tip**
>
> **private > default > protected > public** Pense que la “visibilité s’étend vers l’extérieur”.

2.4 Packages

Les **packages Java** sont des regroupements logiques de classes, d’interfaces et de sous-packages.
Ils aident à organiser de grands codebases, à éviter les conflits de noms et à contrôler l’accès entre différentes parties d’une application.

2.4.1 Organisation et objectif

- La dénomination des packages suit les mêmes règles que les noms de variables. Voir : *Java Naming Rules*.
- Les packages sont comme des **dossiers** pour votre code source Java.
- Ils permettent de regrouper des classes liées entre elles (par exemple toutes les utilitaires dans `java.util`, toutes les classes réseau dans `java.net`).
- En utilisant les packages, vous pouvez éviter les **conflits de noms** : par exemple, vous pouvez avoir deux classes nommées `Date`, l’une `java.util.Date` et l’autre `java.sql.Date`.

2.4.2 Correspondance avec le système de fichiers et déclaration d’un package

- Les packages correspondent directement à la **hiérarchie de répertoires** sur le système de fichiers.
- Vous déclarez le package en haut du fichier source (**avant tout import**).
- Si vous ne déclarez pas de package, la classe appartient au package par défaut.
    - Ceci n’est pas recommandé pour des projets réels, car cela complique l’organisation et les imports.
- Exemple :

```
package com.example.myapp.utils;

public class MyApp{

}
```

> **Important**
>
> Cette déclaration signifie que la classe doit être située dans le répertoire : **com/example/myapp/utils/MyApp.java**

2.4.3 Appartenir au même package

Deux classes appartiennent au même package si et seulement si :

- Elles sont déclarées avec la même instruction `package` en haut de leur fichier source.
- Elles se trouvent dans le même répertoire de la hiérarchie des sources.
- Exemple :

Une classe dans le package `A.B.C` appartient uniquement à `A.B.C`, pas à `A.B`.
Les classes dans `A.B` ne peuvent pas accéder directement aux membres **package-private** des classes de `A.B.C`, car il s’agit de packages différents.

Les classes dans le même package :

- Peuvent accéder aux membres package-private les unes des autres (c’est-à-dire les membres sans modificateur d’accès explicite).
- Partagent le même espace de noms, donc il n’est pas nécessaire de les importer pour les utiliser.

Exemple : deux fichiers dans le même package

```java
// File: src/com/example/tools/Tool.java
package com.example.tools;

public class Tool {
    static void hello() { System.out.println("Hi!"); }
}
```

```java
// File: src/com/example/tools/Runner.java
package com.example.tools;

public class Runner {
    public static void main(String[] args) {
        Tool.hello();    // OK : même package, aucun import nécessaire
    }
}
```

2.4.4 Importer depuis un package

Pour utiliser des classes provenant d'un autre package, vous devez les importer :

- Exemple :

```java
import java.util.List;       // importe une classe spécifique
import java.util.*;          // importe toutes les classes dans java.util

import java.nio.file.*.*     // ERROR! only one wildcard is allowed and it must be at the end!
```

> **Note**
>
> Le caractère wildcard `*` importe tous les types du package, mais pas ses sous-packages.

Vous pouvez toujours utiliser le nom complètement qualifié (fully qualified name) au lieu d'importer toutes les classes du package :

```java
java.util.List myList = new java.util.ArrayList<>();
```

> **Note**
>
> Si vous importez explicitement un nom de classe, il est prioritaire sur tout import avec wildcard ; si vous voulez utiliser deux classes ayant le même nom (par exemple `Date` de `java.util` et de `java.sql`), il est préférable d'utiliser un import avec nom entièrement qualifié.

2.4.5 Imports statiques

En plus d'importer des classes depuis un package, Java permet un autre type d'import : l'**import statique**. Un *static import* permet d'importer les **membres statiques** d'une classe — tels que des méthodes statiques et des variables statiques — afin de les utiliser **sans faire référence au nom de la classe**.

Vous pouvez importer des membres statiques **spécifiques** ou utiliser un **wildcard** pour importer tous les membres statiques d'une classe.

Exemple — import statique spécifique

```java
import static java.util.Arrays.asList;   // Imports Arrays.asList()

public class Example {

    List<String> arr = asList("first", "second");
    // We can call asList() directly, without using Arrays.asList()
}
```

Exemple — import statique d'une constante

```java
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class Circle {
    double radius = 3;

    double area = PI * radius * radius;
    double diagonal = sqrt(2);
}
```

Exemple — import statique avec wildcard

```java
import static java.lang.Math.*;

public class Calculator {
    double x = sqrt(49);   // 7.0
    double y = max(10, 20);
    double z = random();   // calls Math.random()
}
```

Les imports statiques avec wildcard se comportent exactement comme les imports normaux avec wildcard : ils mettent **tous les membres statiques** de la classe dans la portée courante.

> **Warning**
>
> Vous pouvez **toujours** appeler un membre statique avec le nom de la classe : `Math.sqrt(16)` fonctionne toujours — même si le membre a été importé statiquement.

2.4.5.1 Règles de précédence

Si la classe courante déclare déjà une méthode ou une variable portant le même nom qu'un membre importé statiquement :

- Le **membre local est prioritaire**.
- Le membre statique importé est **masqué** (shadowing).

Exemple :

```java
import static java.lang.Math.max;

public class Test {

    static int max(int a, int b) {   // version locale
        return a > b ? a : b;
    }

    void run() {
        System.out.println(max(2, 5));
        // Appelle la version LOCALE de max(), pas Math.max()
    }
}
```

> **Warning**
>
> - Un import statique doit toujours respecter exactement la syntaxe : `import static`.
> - Le compilateur interdit d'importer **deux membres statiques portant le même simple name** si cela crée une ambiguïté — même s'ils proviennent de classes ou de packages différents.

Exemple — **Non autorisé** :

```java
import static java.util.Collections.emptyList;
import static java.util.List.of;

// ❌ ERROR: both methods have the same name `of()`
import static java.util.Set.of;
```

Le compilateur ne sait pas quel `of()` vous souhaitez appeler → échec de compilation.

> **Tip**
>
> - Si deux imports statiques introduisent le même nom, **toute tentative d'utiliser ce nom provoque une erreur de compilation**.
> - Les imports statiques **n'importent pas les classes**, seulement les membres statiques.
> - Vous pouvez toujours appeler le membre statique avec le nom de la classe, même s'il est importé statiquement.

Exemple :

```java
import static java.lang.Math.sqrt;

double a = sqrt(16);        // import statique
double b = Math.sqrt(25);   // fully qualified — toujours autorisé
```

2.4.6 Packages standard vs packages définis par l'utilisateur

- **Packages standard** : fournis avec le JDK (par ex. `java.lang`, `java.util`, `java.io`).
- **Packages définis par l'utilisateur** : créés par les développeurs pour organiser le code de l'application.

2.5 La méthode `main`

En Java, la méthode `main` sert de **point d'entrée** à une application autonome.
Sa déclaration correcte est cruciale pour que la JVM puisse la reconnaître.

2.5.1 Signature de la méthode main

Observons la signature de la méthode `main` dans deux des classes les plus simples possibles :

- Exemple : sans modificateurs optionnels

```java
public class MainFirstExample {

    public static void main(String[] args){

        System.out.print("Hello World!!");

    }

}
```

- Exemple : avec les deux modificateurs optionnels `final`

```java
public class MainSecondExample {

    public final static void main(final String options[]){

        System.out.print("Hello World!!");

    }

}
```

> **Note**
>
> **Table des modificateurs pour la méthode main**

Token / Identifier	Category	Meaning	Optional?
public	Keyword / Access Modifier	Rend la méthode accessible depuis n'importe où. Nécessaire pour que la JVM puisse l'appeler depuis l'extérieur de la classe.	Mandatory
static	Keyword	Signifie que la méthode appartient à la classe elle-même et peut être appelée sans créer d'objet. Nécessaire car la JVM n'a aucune instance au démarrage du programme.	Mandatory
final (before return type)	Modifier	Empêche la méthode d'être redéfinie (overridden). Peut apparaître légalement avant le type de retour, mais n'a aucun effet particulier sur `main` et n'est pas requis.	Optional
main	Method name (predefined)	Nom exact que la JVM recherche comme point d'entrée du programme. Doit être écrit exactement `main` (en minuscules).	Mandatory
void	Return Type / Keyword	Déclare que la méthode ne renvoie aucune valeur à la JVM.	Mandatory
String[] args	Parameter list	Tableau de `String` qui contient les arguments de ligne de commande passés au programme. Peut aussi s'écrire `String args[]` ou `String... args`. Le nom du paramètre (`args`) est arbitraire.	Mandatory (le type du paramètre est requis, le nom peut varier)
final (in parameter)	Modifier	Marque le paramètre comme non réaffectable à l'intérieur du corps de la méthode (vous ne pouvez pas réassigner `args` vers un autre tableau).	Optional

Important

Les modificateurs `public`, `static` (obligatoires) et `final` (s'il est présent) peuvent être permutés ; `public` et `static` ne peuvent pas être omis.

Java considère `String[] args` et `String... args` comme équivalents.
Les deux variantes compilent et fonctionnent correctement comme points d'entrée.

2.6 Compiler et exécuter le code

Cette section présente des commandes `javac` et `java` **correctes et fonctionnelles** pour des situations courantes en Java 21 : fichiers uniques, plusieurs fichiers, packages, répertoires de sortie séparés, utilisation du classpath/module-path.

Suivez exactement l'organisation des répertoires.

> check your tools

```
javac -version   # should print: javac 21.x
java  -version   # should print: java version "21.0.7" ... (the output could be different depe
```

Warning

Lors de l'exécution d'une classe à l'intérieur d'un package, **java exige le nom complètement qualifié**, JAMAIS le chemin :

`java com.example.app.Main` ✓
`java src/com/example/app/Main` ❌

2.6.1 Compiler un fichier, package par défaut (répertoire unique)

Fichiers

```
.
└── Hello.java
```

Hello.java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, Java 21!");
    }
}
```

Compiler (dans le même répertoire)

```
javac Hello.java
```

Cette commande va créer, dans le même répertoire, un fichier portant le même nom que le fichier ".java" mais avec l'extension ".class" ; c'est le fichier de bytecode qui sera interprété et exécuté par la JVM.

Une fois que vous avez le fichier `.class`, dans ce cas `Hello.class`, vous pouvez lancer le programme avec :

Exécution

```
java Hello
```

> **Important**
>
> Vous n'avez pas à préciser l'extension ".class" lors de l'exécution du programme.

2.6.2 Plusieurs fichiers, package par défaut (répertoire unique)

Fichiers

```
.
├── A.java
└── B.java
```

Tout compiler

```
javac *.java
```

Ou, si les classes appartiennent à un package spécifique :

```
javac packagex/*.java
```

Ou en les spécifiant explicitement :

```
javac A.java B.java
```

et

```
javac packagex/A.java packagey/B.java
```

Exécuter le point d'entrée : la classe qui possède une méthode `main`

```
java A    # si A possède main(...)
# ou
java B
```

> **Important**
>
> Le chemin vers vos classes correspond, en Java, au **classpath**. Vous pouvez spécifier le **classpath** avec l'une des options suivantes :
>
> - `-cp <classpath>`
> - `-classpath <classpath>`
> - `--class-path <classpath>`

2.6.3 Code dans des packages (organisation standard src → out)

Fichiers

```
.
├── src/
│   └── com/
│       └── example/
│           └── app/
│               └── Main.java
└── out/
```

> **Note**
>
> Les dossiers `src` et `out` ne font pas partie de nos packages : ce sont simplement les répertoires qui contiennent tous les fichiers source et les fichiers `.class` compilés.

Main.java

```java
package com.example.app;

public class Main {
    public static void main(String[] args) {
        System.out.println("Packages done right.");
    }
}
```

Compiler dans le même répertoire

```
# Crée le fichier .class juste à côté du fichier source
javac src/com/example/app/Main.java
```

2.6.4 Compiler vers un autre répertoire (`-d`)

L'option `-d out` place les fichiers `.class` compilés dans le répertoire `out/`, en créant des sous-dossiers qui reflètent les noms de packages :

```
javac -d out -sourcepath src src/com/example/app/Main.java
```

Exécution (utiliser le classpath pointant sur out/)

```
# Unix/macOS
java -cp out com.example.app.Main

# Windows
java -cp out com.example.app.Main
```

2.6.5 Plusieurs fichiers sur plusieurs packages (compiler tout l'arbre des sources)

Fichiers

```
.
├── src/
│   └── com/
│       └── example/
│           ├── util/
│           │   └── Utils.java
│           └── app/
│               └── Main.java
└── out/
```

Compiler tout l'arbre des sources dans `out/`

```
# Option A : indiquer à javac les packages de plus haut niveau
javac -d out   src/com/example/util/Utils.java   src/com/example/app/Main.java

# Option B : utiliser -sourcepath pour laisser javac trouver les dépendances
javac -d out -sourcepath src   src/com/example/app/Main.java
```

> **Important**
>
> `-sourcepath <sourcepath>` indique à `javac` où chercher d'autres fichiers `.java` dont dépendent les sources.

2.6.6 Exécution d'un fichier source unique (lancement rapide sans `javac`)

Java 21 (depuis Java 11) permet d'exécuter de petits programmes directement à partir du code source :

```
# Uniquement package par défaut
java Hello.java
```

Plusieurs fichiers source sont autorisés s'ils se trouvent dans le **package par défaut** et dans le **même répertoire** :

```
java Main.java Helper.java
```

> Si vous utilisez des **packages**, il est préférable de compiler dans `out/` et d'exécuter avec `-cp`.

2.6.7 Passer des paramètres à un programme Java

Vous pouvez transmettre des données à votre programme Java via les paramètres du point d'entrée `main`.

Comme nous l'avons vu, la méthode `main` peut recevoir un tableau de chaînes sous la forme : **`String[] args`**. Voir la section sur main.

Main.java qui affiche deux paramètres reçus en entrée par la méthode `main` :

```java
package com.example.app;

public class Main {
    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

Pour passer des paramètres, il suffit de taper (par exemple) :

```
java Main.java Hello World  # spaces are used to separate the two arguments
```

Si vous voulez passer un argument contenant des espaces, utilisez des guillemets :

```
java Main.java Hello "World Mario" # spaces are used to separate the two arguments
```

> Si vous déclarez utiliser (ici, afficher) les deux premiers éléments du tableau de paramètres, mais que vous passez en réalité moins d'arguments, la JVM vous signalera un problème via une `java.lang.ArrayIndexOutOfBoundsException`.

> Si, au contraire, vous passez plus d'arguments que ce que la méthode utilise, elle n'affichera que les deux premiers (dans ce cas).

> `args.length` vous indique combien d'arguments ont été fournis.

---

3. Règles de nommage Java

Table des matières

Java définit des règles précises pour les **identifiants**, c'est-à-dire les noms donnés aux variables, méthodes, classes, interfaces et packages.

Tant que vous respectez les règles de nommage décrites ci-dessous, vous êtes libre de choisir des noms significatifs pour les éléments de votre programme.

3.1 Règles pour les identifiants

3.1.1 Mots réservés

Les `identifiers` **ne peuvent pas** être identiques aux **mots-clés** Java ni aux **littéraux réservés**.

Les `keywords` sont des mots spéciaux prédéfinis dans le langage Java que vous n'êtes pas autorisé à utiliser comme identifiants (voir tableau ci-dessous).

Les `literals` comme `true`, `false` et `null` sont également réservés et ne peuvent pas être utilisés comme identifiants.

- Exemple :

```
int class = 5;        // invalid: 'class' is a keyword
boolean true = false; // invalid: 'true' is a literal
int year = 2024;      // valid
```

3.1.1.1 Mots-clés Java réservés

a -> c	c -> f	f -> n	n -> s	s -> w
abstract	continue	for	new	switch
assert	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const*	float	native	super	while

> **Note**
>
> `goto` et `const` sont réservés mais non utilisés.

3.1.1.2 Littéraux réservés

- `true`
- `false`
- `null`

3.1.2 Sensibilité à la casse

Les identifiants en Java sont **sensibles à la casse** (case sensitive).
Cela signifie que `myVar`, `MyVar` et `MYVAR` sont trois identifiants différents.

- Exemple :

```
int myVar = 1;
int MyVar = 2;
int MYVAR = 3;
int CLASS = 6; // legal but, please, don't do it!!
```

> **Tip**
>
> Java traite les identifiants littéralement : `Count`, `count` et `COUNt` sont indépendants et peuvent coexister.
>
> À cause de la sensibilité à la casse, il est possible d'utiliser des variantes de mots-clés qui diffèrent uniquement par la casse.
> Même si c'est légal, ce type de nommage est fortement déconseillé, car il nuit à la lisibilité et est considéré comme une très mauvaise pratique.

3.1.3 Début des identifiants

Les identifiants en Java doivent commencer par une lettre, un symbole monétaire (`$`, `€`, `£`, `₹`...) ou le symbole `_`.

Exemple :

```
int myVarA;
int $myVarB;
int _myVarC;
String €uro = "currency"; // legal (rarely seen in practice)
```

> **Note**
>
> Les symboles de devise sont autorisés, mais ils ne sont pas recommandés dans du code réel.

3.1.4 Chiffres dans les identifiants

Les identifiants Java peuvent contenir des chiffres, mais **ne peuvent pas commencer** par un chiffre.

Exemple :

```
int my33VarA;
int $myVar44;
int 3myVarC; // invalid: identifier cannot start with a digit
int var2024 = 10; // valid
```

3.1.5 Jeton `underscore` seul

- Un underscore (`_`) seul n'est pas autorisé comme identifiant.
- Depuis Java 9, `_` est un jeton réservé pour un usage futur du langage.
- Exemple :

```
int _;  // invalid since Java 9
```

> **Warning**
>
> `_` est autorisé à l'intérieur des littéraux numériques (voir section suivante), mais pas comme identifiant isolé.

3.1.6 Littéraux numériques et caractère underscore

Vous pouvez utiliser un ou plusieurs caractères `_` (underscore) dans les littéraux numériques afin de les rendre plus lisibles.

Vous pouvez placer des underscores presque partout, **sauf** au début, à la fin ou juste autour du point décimal (immédiatement avant ou après).

- Exemple :

```
int firstNum = 1_000_000;
int secondNum = 1 _____________ 2;

double firstDouble = _1000.00   // DOES NOT COMPILE
double secondDouble = 1000_.00  // DOES NOT COMPILE
double thirdDouble = 1000._00   // DOES NOT COMPILE
double fourthDouble = 1000.00_  // DOES NOT COMPILE

double pi = 3.14_159_265; // valid
long mask = 0b1111_0000;  // valid in binary literals
```

> **Tip**
>
> Les underscores améliorent la lisibilité : `1_000_000` est plus facile à lire que `1000000`.

---

4. Types de données Java et cast

Table des matières

Comme nous l'avons vu dans les Blocs Syntaxiques, Java propose deux catégories de types de données :

- **Types primitifs**
- **Types référence**

👉 Pour une vue complète des types primitifs avec leur taille, plage de valeurs, valeurs par défaut et exemples, voir le Tableau des types primitifs.

4.1 Types primitifs

Les `primitives` représentent des **valeurs brutes uniques** stockées directement en mémoire.
Chaque type primitif possède une taille fixe qui détermine le nombre d'octets qu'il occupe.

Conceptuellement, un primitif est simplement une **cellule mémoire** contenant une valeur :

```
+-------+
|  42   |   ← valeur de type short (2 octets en mémoire)
+-------+
```

4.2 Types référence

Un type `référence` ne contient pas l'`objet` lui-même, mais une **référence (pointeur)** vers celui-ci.
La référence a une taille fixe (dépendante de la JVM, souvent 4 ou 8 octets) qui pointe vers un emplacement mémoire où l'objet réel est stocké.

- Exemple : une variable de type `String` pointe vers un objet chaîne dans le tas (heap), qui est lui-même composé en interne d'un tableau de primitives `char`.

Diagramme :

```
Référence (4 ou 8 octets)
+---------+
| address | ──────────────►  Objet dans le tas (Heap)
+---------+                 +-------------------+
                            |   "Hello"         |
                            | ['H','e','l','l','o']  ← tableau de char
                            +-------------------+
```

4.3 Tableau des types primitifs

Mot-clé	Type	Taille	Valeur min	Valeur max	Valeur par défaut	Exemple
`byte`	int 8 bits	1 octet	−128	127	0	`byte b = 100;`
`short`	int 16 bits	2 octets	−32 768	32 767	0	`short s = 2000;`
`int`	int 32 bits	4 octets	−2 147 483 648 (−2^31)	2 147 483 647 (2^31−1)	0	`int i = 123456;`
`long`	int 64 bits	8 octets	−2^63	2^63−1	0L	`long l = 123456789L;`
`float`	flottant 32 bits	4 octets	voir note	voir note	0.0f	`float f = 3.14f;`
`double`	flottant 64 bits	8 octets	voir note	voir note	0.0	`double d = 2.718;`
`char`	UTF-16	2 octets	`'\u0000'` (0)	`'\uffff'` (65 535)	`'\u0000'`	`char c = 'A';`
`boolean`	true/false	dépend de la JVM (souvent 1 octet)	false	true	false	`boolean b = true;`

4.4 Notes

`float` et `double` n'ont pas de bornes entières fixes comme les types entiers.
Ils suivent la norme IEEE 754 :

- **Plus petites valeurs positives non nulles** :
  - `Float.MIN_VALUE ≈ 1.4E-45`
  - `Double.MIN_VALUE ≈ 4.9E-324`
- **Plus grandes valeurs finies** :
  - `Float.MAX_VALUE ≈ 3.4028235E+38`
  - `Double.MAX_VALUE ≈ 1.7976931348623157E+308`

Ils supportent également des valeurs spéciales : `+Infinity`, `-Infinity`, et `NaN` (Not a Number).

- **FP** = floating point (virgule flottante).
- La taille de `boolean` dépend de la JVM mais le type se comporte logiquement comme `true` / `false`.
- Les valeurs par défaut s'appliquent aux **champs** (variables de classe).
  Les **variables locales** doivent être explicitement initialisées avant utilisation.

4.5 Récapitulatif

- **Primitif** = valeur réelle, stockée directement en mémoire.
- **Référence** = pointeur vers un objet ; l'objet lui-même peut contenir des primitives et d'autres références.
- Pour les détails des primitives, voir le Tableau des types primitifs.

4.6 Arithmétique et promotion numérique primitive

Lorsqu'on applique des opérateurs arithmétiques ou de comparaison à des **types primitifs**, Java convertit (ou *promeut*) automatiquement les valeurs vers des types compatibles selon des **règles de promotion numérique** bien définies.

Ces règles garantissent des calculs cohérents et évitent la perte de données lors du mélange de types numériques différents.

4.6.1 🔹 Règles de promotion numérique en Java

4.6.1.1 Règle 1 – Types numériques mixtes → le plus petit type est promu vers le plus grand

Si deux opérandes appartiennent à des **types numériques différents**, Java promeut automatiquement le type le **plus petit** vers le type le **plus grand** avant d'effectuer l'opération.

Exemple	Explication
`int x = 10; double y = 5.5; double result = x + y;`	La valeur `int x` est promue en `double`, donc le résultat est un `double` (`15.5`).

Ordre de promotion valide (du plus petit au plus grand) :
`byte → short → int → long → float → double`

4.6.1.2 Règle 2 – Entier + flottant → l'entier est promu vers le flottant

Si un opérande est de type **entier** (`byte`, `short`, `char`, `int`, `long`) et l'autre de type **flottant** (`float`, `double`),
la **valeur entière est promue** vers le type flottant avant l'opération.

Exemple	Explication
`float f = 2.5F; int n = 3; float result = f * n;`	`n` (int) est promu en `float`. Le résultat est un `float` (`7.5`).
`double d = 10.0; long l = 3; double result = d / l;`	`l` (long) est promu en `double`. Le résultat est un `double` (`3.333...`).

4.6.1.3 Règle 3 – `byte`, `short` et `char` sont promus en `int` lors des opérations arithmétiques

Lorsqu'on effectue une opération arithmétique sur des **variables** (et non des constantes littérales) de type `byte`, `short` ou `char`,
Java les promeut automatiquement en `int`, même si **les deux opérandes sont plus petits que** `int`.

Exemple	Explication
`byte a = 10, b = 20; byte c = a + b;`	❌ Erreur de compilation : le résultat de `a + b` est un `int`, pas un `byte`. Il faut caster → `byte c = (byte)(a + b);`
`short s1 = 1000, s2 = 2000; short sum = (short)(s1 + s2);`	Les opérandes sont promus en `int`, un cast explicite est nécessaire pour affecter à `short`.
`char c1 = 'A', c2 = 2; int result = c1 + c2;`	`'A'` (65) et `2` sont promus en `int`, résultat = `67`.

> **Note**
>
> Cette règle s'applique uniquement lorsqu'on utilise des **variables**. Lorsque l'on utilise des **littéraux constants**, le compilateur peut parfois évaluer l'expression à la compilation et l'affecter sans problème.

```
byte a = 10 + 20;    // ✅ OK : expression constante qui tient dans un byte
byte b = 10;
byte c = 20;
byte d = b + c;      // ❌ Erreur : b + c est évalué à l'exécution → int
```

4.6.1.4 Règle 4 – Le type du résultat correspond au type promu des opérandes

Une fois les promotions appliquées, et lorsque les deux opérandes sont du même type,
le **résultat** de l'expression a ce **même type promu**.

Exemple	Explication
`int i = 5; double d = 6.0; var result = i * d;`	`i` est promu en `double`, le résultat est un `double`.
`float f = 3.5F; long l = 4L; var result = f + l;`	`l` est promu en `float`, le résultat est un `float`.
`int x = 10, y = 4; var div = x / y;`	Les deux sont `int`, le résultat est un `int` (`2`), la partie fractionnaire est tronquée.

> **Warning**
>
> La division entière produit toujours un **résultat entier**. Pour obtenir un résultat décimal, **au moins un opérande doit être flottant** :

```
double result = 10.0 / 4; // ✅ 2.5
int result2 = 10 / 4;     // ❌ 2 (fraction ignorée)
```

4.6.2 ✅ Récapitulatif du comportement de promotion numérique

Situation	Résultat de promotion	Exemple
Mélange de petits et grands types numériques	Le plus petit est promu vers le plus grand	int + double → double
Entier + flottant	L'entier est promu vers le flottant	long + float → float
Arithmétique sur byte, short, char	Promu en int	byte + byte → int
Résultat après promotion	Le type du résultat correspond au type promu	float + long → float

4.6.2.1 🧠 Points clés

- Toujours tenir compte de la **promotion de type** lorsqu'on mélange des types dans une expression arithmétique.
- Pour les petits types (`byte` , `short` , `char`), la promotion en `int` est automatique dès qu'il y a une opération avec des **variables**.
- Utilisez le **cast explicite** seulement lorsque vous êtes sûr que le résultat tient dans le type cible.
- Rappelez-vous : **la division entière tronque**, la **division en flottant conserve les décimales**.
- Comprendre ces règles est essentiel pour éviter les **pertes de précision inattendues** ou les **erreurs de compilation** à l'examen de certification Java.

4.7 Cast en Java

En Java, le `cast` est le processus par lequel on convertit explicitement une valeur d'un type vers un autre. Cela s'applique à la fois aux `types primitifs` (nombres) et aux `types référence` (objets dans une hiérarchie de classes).

4.7.1 Cast primitif

Le cast primitif modifie le type d'une valeur numérique.

Il existe deux catégories de cast :

Type	Description	Exemple	Explicite ?	Risque
Widening	plus petit type → plus grand type	int → double	Non	Aucune perte
Narrowing	plus grand type → plus petit type	double → int	Oui	Perte possible

4.7.1.1 Cast large implicite

Conversion automatique d'un type "plus petit" vers un type compatible "plus grand".
Gérée par le compilateur, **ne nécessite pas de syntaxe explicite**.

```
int i = 100;
double d = i;  // cast implicite : int → double
System.out.println(d); // 100.0
```

✅ **Sûr** – pas de dépassement (même s'il faut garder en tête la précision des flottants).

4.7.1.2 Cast étroit explicite

Conversion manuelle d'un type « plus grand » vers un type « plus petit ».
Nécessite une **cast expression** car cela peut provoquer une perte de données.

```java
double d = 9.78;
int i = (int) d;  // explicit cast: double → int
System.out.println(i); // 9 (fraction discarded)
```

> **Warning**
>
> ⚠ Utiliser uniquement lorsque vous êtes certain que la valeur tient dans le type cible.

4.7.1.3 Cast étroit Implicite à la Compilation

Dans certains cas spécifiques, le compilateur autorise une conversion de narrowing **sans cast explicite**.

Si une variable est déclarée `final` et initialisée avec une constant expression dont la valeur tient dans le type cible, le compilateur peut effectuer la conversion en toute sécurité au moment de la compilation.

```java
final int k = 10;
byte b = k;  // allowed: value 10 fits into byte range

final int x = 200;
byte c = x;  // does NOT compile: 200 is outside byte range
```

Cela fonctionne parce que le compilateur connaît la valeur exacte d'une variable `final` et peut vérifier qu'elle se situe dans l'intervalle du type plus petit.

Ce type de narrowing est autorisé entre : - `byte` - `short` - `char` - `int`

Cependant, cela ne s'applique pas à : - `long` - `float` - `double`

Par exemple :

```java
final float f = 10.0f;
int n = f;   // does not compile
```

Même si la valeur semble compatible, les types à virgule flottante ne sont pas éligibles pour cette forme de narrowing implicite.

4.7.2 Perte de données, dépassement et sous-dépassement

Lorsqu'une valeur dépasse la capacité d'un type, on peut obtenir :

- **Dépassement (overflow)** : résultat supérieur à la valeur maximale représentable
- **Sous-dépassement (underflow)** : résultat inférieur à la valeur minimale représentable
- **Troncature** : les données qui ne tiennent pas sont perdues (par exemple, les décimales)
- Exemple – overflow/underflow avec int

```java
int max = Integer.MAX_VALUE;
int overflow = max + 1;     // retour ("wrap-around") vers une valeur négative

int min = Integer.MIN_VALUE;
int underflow = min - 1;    // retour ("wrap-around") vers une valeur positive
```

- Exemple – troncature

```java
double d = 9.99;
int i = (int) d; // 9 (fraction supprimée)
```

> **Note**
>
> Les types flottants (`float` , `double`) **ne font pas de wrap-around** : - overflow → `Infinity` / `-Infinity`
> - underflow (valeurs très petites) → 0.0 ou valeurs dénormalisées.

4.7.3 Cast de valeurs versus variables

Java traite :

- Les **littéraux entiers** comme des `int` par défaut
- Les **littéraux flottants** comme des `double` par défaut

Le compilateur **n'exige pas de cast** lorsqu'un littéral tient dans la plage du type cible :

```
byte first = 10;         // OK : 10 tient dans un byte
short second = 9 * 10;   // OK : expression constante évaluée à la compilation
```

Mais :

```
long a = 5729685479;     // ❌ erreur : littéral int hors plage
long b = 5729685479L;    // ✅ littéral long (suffixe L)

float c = 3.14;          // ❌ double → float : nécessite F ou cast
float d = 3.14F;         // ✅ littéral float

int e = 0x7FFF_FFFF;     // ✅ max int en hexadécimal
int f = 0x8000_0000;     // ❌ hors plage int (nécessite L)
```

Cependant, lorsque les règles de promotion numérique s'appliquent :

> Avec des variables de type `byte`, `short` et `char` dans une expression arithmétique, les opérandes sont promus en `int` et le résultat est un `int`.

```
byte first = 10;
short second = 9 + first;        // ❌ 9 (littéral int) + first (byte → int) = int
// second = (short) (9 + first);  // ✅ cast de l'expression entière
```

```
short b = 10;
short a = 5 + b;                 // ❌ 5 (int) + b (short → int) = int
short a2 = (short) (5 + b);      // ✅ cast de l'expression entière
```

> **Warning**
>
> Le cast est un **opérateur unaire** :
>
> `short a = (short) 5 + b;`
> Le cast s'applique uniquement à `5` → le résultat de l'expression reste un int → l'affectation échoue toujours.

4.7.4 Cast de référence d'objets

Le cast s'applique aussi aux **références d'objets** dans une hiérarchie de classes.
Il ne change pas l'objet en mémoire — seulement le **type de référence** utilisé pour y accéder.

Règles importantes :

- Le **type réel de l'objet** détermine quels champs/méthodes existent réellement.
- Le **type de la référence** détermine ce que vous pouvez appeler/accéder à cet endroit du code.

4.7.4.1 Upcasting (cast large de référence)

Conversion d'une **sous-classe** vers une **super-classe**.

Implicite et **toujours sûre** : chaque `Dog` est aussi un `Animal`.

```
class Animal { }
class Dog extends Animal { }

Dog dog = new Dog();
Animal a = dog;     // upcast implicite : Dog → Animal
```

4.7.4.2 Downcasting (cast étroit de référence)

Conversion d'une **super-classe** vers une **sous-classe**.

- **Explicite**
- Peut échouer à l'exécution avec `ClassCastException` si l'objet n'est pas réellement de ce type

```
Animal a = new Dog();
Dog d = (Dog) a;   // OK : a référence réellement un Dog

Animal x = new Animal();
Dog d2 = (Dog) x;  // ⚠ Erreur à l'exécution : ClassCastException
```

Pour plus de sécurité, utilisez `instanceof` :

```
if (x instanceof Dog) {
    Dog safeDog = (Dog) x;   // cast sûr
}
```

4.7.5 Résumé des points clés

Type de cast	S'applique à	Direction	Syntaxe	Sûr ?	Effettué par
Widening Primitive	Primitifs	petit → grand	Implice	Oui	Compilateur
Narrowing Primitive	Primitifs	grand → petit	Explicite	Non	Programmeur
Upcasting	Objets	sous-classe → super-classe	Implice	Oui	Compilateur
Downcasting	Objets	super-classe → sous-classe	Explicite	Vérification à l'exécution	Programmeur

---

4.7.6 Exemples

```
// Cast primitif
short s = 50;
int i = s;           // widening
byte b = (byte) i;   // narrowing (perte possible)

// Cast d'objet
Object obj = "Hello";
String str = (String) obj; // OK : obj référence une String

Object n = Integer.valueOf(10);
// String fail = (String) n;  // ClassCastException à l'exécution
```

4.8 Résumé

- Le **cast primitif** change le type numérique.
- Le **cast de référence** change la "vue" d'un objet dans la hiérarchie.
- **Upcasting** → sûr et implicite.
- **Downcasting** → explicite, à utiliser avec prudence (souvent après `instanceof`).

◀ 3. Règles de nommage Java | ▲ Index | 5. Opérateurs Java ▶

5. Opérateurs Java

Table des matières

5.1 Définition

En Java, les **opérateurs** sont des symboles spéciaux qui effectuent des opérations sur des variables et des valeurs.
Ce sont les briques de base des expressions et ils permettent aux développeurs de manipuler des données, de comparer des valeurs, d'effectuer des opérations arithmétiques et de contrôler le flux logique.

Une **expression** est une combinaison d'opérateurs et d'opérandes qui produit un résultat.

Par exemple :

```
int result = (a + b) * c;
```

Ici, `+` et `*` sont des opérateurs, et `a`, `b` et `c` sont des opérandes.

5.2 Types d'opérateurs

Java définit trois types d'opérateurs, regroupés selon le nombre d'opérandes qu'ils utilisent :

Type	Description	Exemples
Unary	Opère sur un seul opérande	`+x`, `-x`, `++x`, `--x`, `!flag`, `~num`
Binary	Opère sur deux opérandes	`a + b`, `a - b`, `x * y`, `x / y`, `x % y`
Ternary	Opère sur trois opérandes (un seul en Java)	`condition ? valueIfTrue : valueIfFalse`

5.3 Catégories d'opérateurs

Les opérateurs peuvent également être regroupés, selon leur objectif, en catégories :

Catégorie	Description	Exemples
Assignment	Assigne des valeurs aux variables	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
Relational	Compare des valeurs	`==`, `!=`, `<`, `>`, `<=`, `>=`
Logical	Combine ou inverse des expressions booléennes	`|`, `&`, `^`
Conditional	Combine ou inverse des expressions booléennes	`||`, `&&`
Bitwise	Manipule des bits	`&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
Instanceof	Teste le type d'un objet	`obj instanceof ClassName`
Lambda	Utilisé dans les expressions lambda	`(x, y) -> x + y`

5.4 Priorité des opérateurs et ordre d'évaluation

La **priorité des opérateurs** détermine comment les opérateurs sont regroupés dans une expression — c'est-à-dire quelles opérations sont effectuées en premier.
L'**associativité** (ou **ordre d'évaluation**) détermine si l'expression est évaluée de **gauche à droite** ou de **droite à gauche** lorsque des opérateurs ont la même priorité.

Exemple :

```
int result = 10 + 5 * 2;  // La multiplication est effectuée avant l'addition → result = 20
```

Les parenthèses `()` peuvent être utilisées pour **surclasser la priorité** :

```
int result = (10 + 5) * 2;  // Les parenthèses sont évaluées en premier → result = 30
```

Note

- La **priorité** des opérateurs concerne le *regroupement*, pas l'ordre concret d'évaluation.
- Utilise les parenthèses pour la priorité et la clarté dans les expressions complexes.

5.5 Tableau récapitulatif des opérateurs Java

Priorité (Haute → Basse)	Type	Opérateurs	Exemple	Ordre d'évaluation	Applicable à
1	Postfix Unary	`expr++`, `expr--`	`x++`	Gauche → Droite	Types numériques
2	Prefix Unary	`++expr`, `--expr`	`--x`	Gauche → Droite	Numériques
3	Other Unary	`(type)`, `+`, `-`, `~`, `!`	`-x`, `!flag`	Droite → Gauche	Numériques, boolean
4	Cast Unary	`(Type) reference`	`(short) 22`	Droite → Gauche	reference, primitifs
5	Multiplication/division/modulus	`*`, `/`, `%`	`a * b`	Gauche → Droite	Types numériques
6	Additive	`+`, `-`	`a + b`	Gauche → Droite	Numériques, String (concaténation)
7	Shift	`<<`, `>>`, `>>>`	`a << 2`	Gauche → Droite	Types intégraux
8	Relational	`<`, `>`, `<=`, `>=`, `instanceof`	`a < b`, `obj instanceof Person`	Gauche → Droite	Numériques, reference
9	Equality	`==`, `!=`	`a == b`	Gauche → Droite	Tous les types (sauf boolean pour `<`, `>`)
10	Logical AND	`&`	`a & b`	Gauche → Droite	boolean
11	Logical exclusive OR	`^`	`a ^ b`	Gauche → Droite	boolean
12	Logical inclusive OR	`|`	`a | b`	Gauche → Droite	boolean
13	Conditional AND	`&&`	`a && b`	Gauche → Droite	boolean
14	Conditional OR	`||`	`a || b`	Gauche → Droite	boolean
15	Ternary (Conditional)	`? :`	`a > b ? x : y`	Droite → Gauche	Tous les types
16	Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 5`	Droite → Gauche	Tous les types assignables
17	Arrow operator	`->`	`(x, y) -> x + y`	Droite → Gauche	Expressions lambda, switch rules

5.5.1 Notes complémentaires

- La **concaténation de chaînes (`+`)** a une priorité plus faible que le `+` arithmétique sur les nombres.
- Utilise les parenthèses `()` pour la priorité et la lisibilité — elles ne changent pas la sémantique mais rendent l'intention explicite.

5.6 Opérateurs unaires

Les opérateurs unaires opèrent sur **un seul opérande** pour produire une nouvelle valeur.
Ils sont utilisés pour des opérations comme l'incrémentation/décrémentation, la négation d'une valeur, l'inversion d'un booléen ou le complément bit à bit.

5.6.1 Catégories d'opérateurs unaires

Opérateur	Nom	Description	Exemple	Résultat
`+`	Unary plus	Indique une valeur positive (souvent redondant).	`+x`	Identique à `x`
`-`	Unary minus	Indique qu'un nombre littéral est négatif ou nie une expression.	`-5`	`-5`
`++`	Increment	Augmente une variable de 1. Peut être préfixe ou postfixe.	`++x`, `x++`	`x+1`
`--`	Decrement	Diminue une variable de 1. Peut être préfixe ou postfixe.	`--x`, `x--`	`x-1`
`!`	Logical complement	Inverse une valeur booléenne.	`!true`	`false`
`~`	Bitwise complement	Inverse chaque bit d'un entier. Quick rule: ~n = -(n + 1)	`~5`	`-6`
`(type)`	Cast	Convertit la valeur vers un autre type.	`(int) 3.9`	`3`

5.6.2 Exemples

```
int x = 5;
System.out.println(++x);  // 6  (préfixe : incrémente x à 6, puis renvoie 6)
System.out.println(x++);  // 6  (postfixe : renvoie 6, puis incrémente x à 7)
System.out.println(x);    // 7

boolean flag = false;
System.out.println(!flag);  // true

int a = 5;                  // binaire : 0000 0101
System.out.println(~a);     // -6 → binaire : 1111 1010 (complément à deux)
```

> **Note**
>
> - Préfixe (`++x` / `--x`) : met à jour la valeur d'abord, puis renvoie la nouvelle valeur.
> - Postfixe (`x++` / `x--`) : renvoie d'abord la valeur courante, puis la met à jour.
> - L'opérateur `!` s'applique aux valeurs boolean ; `~` s'applique aux types numériques intégraux.

5.7 Opérateurs binaires

Les opérateurs binaires nécessitent **deux opérandes**.
Ils effectuent des opérations arithmétiques, relationnelles, logiques, bit à bit et d'affectation.

5.7.1 Catégories d'opérateurs binaires

Catégorie	Opérateurs	Exemple	Description
Arithmetic	`+`, `-`, `*`, `/`, `%`	`a + b`	Opérations mathématiques de base.
Relational	`<`, `>`, `<=`, `>=`, `==`, `!=`	`a < b`	Compare des valeurs.
Logical (boolean)	`&`, `|`, `^`	`a & b`	Voir note ci-dessous.
Conditional	`&&`, `||`	`a && b`	Voir note ci-dessous.
Bitwise (integral)	`&`, `|`, `^`, `<<`, `>>`, `>>>`	`a << 2`	Opère sur les bits.
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	`x += 3`	Modifie puis affecte.
String Concatenation	`+`	`"Hello " + name`	Concatène des chaînes.

Important

- Les opérateurs **logiques** (`&`, `|`, `^`) *évaluent toujours les deux côtés*.
- Les opérateurs **conditionnels** (`&&`, `||`) sont **short-circuit** :
  - `a && b` → `b` est évalué uniquement si `a` est true
  - `a || b` → `b` est évalué uniquement si `a` est false

Important

Cheat Sheet Pattern Bitwise et Boolean

```
a ^ a = 0

a ^ 0 = a

a ^ -1 = ~a

a ^ ~a = -1

a & a = a

a | 0 = a
```

Exemple arithmétique :

```java
int a = 10, b = 4;
System.out.println(a + b);  // 14
System.out.println(a - b);  // 6
System.out.println(a * b);  // 40
System.out.println(a / b);  // 2
System.out.println(a % b);  // 2
```

Exemple relationnel :

```java
int a = 5, b = 8;
System.out.println(a < b);   // true
System.out.println(a >= b);  // false
System.out.println(a == b);  // false
System.out.println(a != b);  // true
```

Exemple logique :

```
boolean x = true, y = false;
System.out.println(x && y);  // false
System.out.println(x || y);  // true
System.out.println(!x);      // false
```

Exemple bit à bit :

```
int a = 5;   // 0101
int b = 3;   // 0011
System.out.println(a & b);  // 1  (0001)
System.out.println(a | b);  // 7  (0111)
System.out.println(a ^ b);  // 6  (0110)
System.out.println(a << 1); // 10 (1010)
System.out.println(a >> 1); // 2  (0010)
```

5.7.2 Opérateurs de division et de modulo (reste)

5.7.2.1 Opérateur de Division

Diviser un `entier` par zéro (par exemple, 10 / 0) provoque le lancement par la JVM d'une `java.lang.ArithmeticException: / by zero`.

Cependant, la division en virgule flottante se comporte différemment.

Lorsqu'une valeur `float` ou `double` est divisée par 0 ou 0.0, aucune exception n'est levée. À la place, le résultat est :

- **Float.POSITIVE_INFINITY** ou **Float.NEGATIVE_INFINITY**
- **Double.POSITIVE_INFINITY** ou **Double.NEGATIVE_INFINITY**

Le signe dépend des opérandes impliqués dans l'opération.

Pour déterminer si une valeur en virgule flottante représente l'infini, les classes `Float` et `Double` fournissent des méthodes utilitaires :

Méthodes statiques :

- **Float.isInfinite(float value)**
- **Double.isInfinite(double value)**

Méthodes d'instance :

- **Float.isInfinite()**
- **Double.isInfinite()**

Ces méthodes retournent `true` si la valeur correspond à l'infini positif ou à l'infini négatif.

5.7.2.2 Opérateur Modulo

L'opérateur modulo donne le reste lorsque deux nombres sont divisés.
Si deux nombres se divisent exactement, le reste est 0 : par exemple **10 % 5** vaut 0.
En revanche, **13 % 4** donne un reste de 1.

On peut utiliser le modulo avec des nombres négatifs selon les règles suivantes :

- si le **diviseur** est négatif (ex. : 7 % -5), le signe est ignoré et le résultat est **2** ;
- si le **dividende** est négatif (ex. : -7 % 5), le signe est conservé et le résultat est **-2** ;

```java
System.out.println(8 % 5);      // GIVES 3
System.out.println(10 % 5);     // GIVES 0
System.out.println(10 % 3);     // GIVES 1
System.out.println(-10 % 3);    // GIVES -1
System.out.println(10 % -3);    // GIVES 1
System.out.println(-10 % -3);   // GIVES -1

System.out.println(8 % 9);      // GIVES 8
System.out.println(3 % 4);      // GIVES 3
System.out.println(2 % 4);      // GIVES 2
System.out.println(-8 % 9);     // GIVES -8
```

5.7.3 La valeur de retour de l'opérateur d'affectation

En Java, l'**opérateur d'affectation (`=`)** ne fait pas que stocker une valeur dans une variable — il **renvoie aussi la valeur affectée** comme résultat de l'expression entière.

Cela signifie que l'affectation peut être **utilisée comme partie d'une autre expression**, par exemple dans une condition `if`, dans la condition d'une boucle, ou même dans une autre affectation.

```java
int x;
int y = (x = 10);   // l'affectation (x = 10) renvoie 10
System.out.println(y);  // 10

// x = 10 affecte 10 à x.
// L'expression (x = 10) s'évalue à 10.
// Cette valeur est ensuite affectée à y.
// Donc x et y finissent avec la même valeur (10).
```

Comme l'affectation renvoie une valeur, elle peut aussi apparaître dans un **if**.
Cependant, cela conduit souvent à des erreurs logiques si c'est fait involontairement.

```java
boolean flag = false;

if (flag = true) {
    System.out.println("This will always execute!");
}

// Ici la condition (flag = true) affecte true à flag, puis s'évalue à true,
// donc le bloc if s'exécute toujours.

// Usage correct (comparaison au lieu d'affectation) :

if (flag == true) {
    System.out.println("Condition checked, not assigned");
}
```

> **Warning**
>
> Si tu vois `if (x = quelque chose)`, stop : c'est une **affectation**, pas une comparaison.

5.7.4 Opérateurs d'affectation composée

Les **opérateurs d'affectation composée** en Java combinent une opération arithmétique ou bit à bit avec une affectation en une seule étape.
Au lieu d'écrire `x = x + 5`, tu peux utiliser la forme abrégée `x += 5`.
Ils effectuent automatiquement un **cast implicite** vers le type de la variable à gauche lorsque c'est nécessaire.

Les opérateurs composés courants incluent :
`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, et `>>>=`.

```java
int x = 10;

// Affectations composées arithmétiques
x += 5;   // équivalent à x = x + 5 → x = 15
x -= 3;   // équivalent à x = x - 3 → x = 12
x *= 2;   // équivalent à x = x * 2 → x = 24
x /= 4;   // équivalent à x = x / 4 → x = 6
x %= 5;   // équivalent à x = x % 5 → x = 1

// Affectations composées bit à bit
int y = 6;   // 0110 (binaire)
y &= 3;      // y = y & 3 → 0110 & 0011 = 0010 → y = 2
y |= 4;      // y = y | 4 → 0010 | 0100 = 0110 → y = 6
y ^= 5;      // y = y ^ 5 → 0110 ^ 0101 = 0011 → y = 3

// Affectations composées avec décalage
int z = 8;   // 0000 1000
z <<= 2;     // z = z << 2 → 0010 0000 → z = 32
z >>= 1;     // z = z >> 1 → 0001 0000 → z = 16
z >>>= 2;    // z = z >>> 2 → 0000 0100 → z = 4

// Exemple de cast de type
byte b = 10;
// b = b + 1;   // ❌ erreur de compilation : le résultat int ne peut pas être affecté à byte
b += 1;         // ✅ fonctionne : cast implicite vers byte
```

> **Note**
>
> Les affectations composées effectuent un **cast implicite** vers le type de la variable à gauche. C'est pourquoi `b += 1` compile alors que `b = b + 1` ne compile pas.

5.7.5 Opérateurs d'égalité (`==` et `!=`)

Les **opérateurs d'égalité** en Java `==` (égal à) et `!=` (différent de) servent à comparer deux opérandes. Cependant, leur comportement diffère **selon qu'ils s'appliquent à des types primitifs ou à des types référence (objets)**.

> **Note**
>
> - `==` compare les **valeurs** pour les primitifs
> - `==` compare les **références** pour les objets
> - `.equals()` compare le **contenu** d'un objet (si implémenté)

5.7.5.1 Égalité avec les types primitifs

Lorsqu'on compare des **valeurs primitives**, `==` et `!=` comparent les **valeurs stockées**.

```java
int a = 5, b = 5;
System.out.println(a == b);  // true  → mêmes valeurs
System.out.println(a != b);  // false → valeurs égales
```

> **Important**
>
> - Si les opérandes sont de types numériques différents, Java les promeut automatiquement vers un type commun avant la comparaison.
> - Cependant, comparer float et double peut produire des résultats inattendus à cause des erreurs de précision (voir ci-dessous).

```java
int x = 10;
double y = 10.0;
System.out.println(x == y);  // true → x promu en double (10.0)

double d = 0.1 + 0.2;
System.out.println(d == 0.3); // false → problème d'arrondi floating-point
```

5.7.5.2 Égalité avec les types référence (objets)

Pour les objets, `==` et `!=` comparent les références, pas le contenu.
Ils renvoient true uniquement si les deux références pointent vers **le même objet** en mémoire.

```java
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2);      // false → objets différents en mémoire
System.out.println(s1 != s2);      // true  → pas la même référence
```

Même si deux objets ont un contenu identique, `==` compare leurs **adresses**, pas leurs valeurs.
Pour comparer le **contenu** des objets, utilise `.equals()`.

```java
System.out.println(s1.equals(s2)); // true → même contenu de chaîne
```

Cas particulier : null et littéraux String

- Toute référence peut être comparée à `null` avec `==` ou `!=`.

```java
String text = null;
System.out.println(text == null);  // true
```

- Les littéraux String sont *internés* par la JVM :
  des littéraux identiques peuvent donc pointer vers la même référence en mémoire :

```java
String a = "Java";
String b = "Java";
System.out.println(a == b);       // true → même littéral interné
```

- Égalité avec types mixtes :
  avec `==` entre catégories différentes (ex. primitif vs objet),
  le compilateur tente l'unboxing si l'un des deux est une **classe wrapper**.

```java
Integer i = 100;
int j = 100;
System.out.println(i == j);   // true → unboxing avant comparaison
```

5.7.6 L'opérateur `instanceof`

`instanceof` est un **opérateur relationnel** qui teste si une référence est une **instance** d'un certain **type référence** à l'exécution.
Il renvoie un `boolean`.

```java
Object o = "Java";
boolean b1 = (o instanceof String);   // true
boolean b2 = (o instanceof Number);   // false
```

Comportement avec **null** :
si l'expression est null, **expr instanceof Type** est toujours **false**.

```java
Object n = null;
System.out.println(n instanceof Object);  // false
```

> **Warning**
>
> `instanceof` renvoie toujours `false` lorsque l'opérande gauche est `null`.

5.7.6.1 Vérification à la compilation vs à l'exécution

- À la compilation, le compilateur rejette les types inconvertibles (qui ne peuvent pas être liés à l'exécution).
- À l'exécution, si la vérification compile-time a passé, la JVM évalue le type réel de l'objet.

```
// ❌ Erreur de compilation : types inconvertibles (String est sans rapport avec Integer)
boolean bad = ("abc" instanceof Integer);

// ✅ Compile, mais le résultat à l'exécution dépend de l'objet réel :

Number num = Integer.valueOf(10);
System.out.println(num instanceof Integer); // true à l'exécution
System.out.println(num instanceof Double);  // false à l'exécution
```

5.7.6.2 Pattern matching pour instanceof

Java supporte les *type patterns* avec `instanceof`, qui testent et lient une variable si le test réussit. Ajouter une variable après le type indique au compilateur d'interpréter cela comme du *Pattern Matching*.

Syntaxe (forme pattern) :

```
Object obj = "Hello";

if (obj instanceof String str) {
    // Ajouter la variable str après le type indique au compilateur de faire du Pattern Matchi

    System.out.println(str.toUpperCase()); // l'identifiant est en scope ici, de type String (
}
```

Propriétés clés :

- Si le test réussit, la variable de pattern (ex. `str`) est définitivement assignée et visible dans la branche true.
- Les variables de pattern sont implicitement final (ne peuvent pas être réassignées).
- Le nom ne doit pas entrer en conflit avec une variable existante dans le même scope.

5.7.6.3 Flow scoping & logique short-circuit

Les variables de pattern deviennent disponibles selon l'analyse de flux :

```
Object obj = "data";

// Test négatif, variable disponible dans la branche else
if (!(obj instanceof String s)) {
    // s n'est pas en scope ici
} else {
    System.out.println(s.length()); // s est en scope ici
}

// Avec &&, la variable de pattern peut être utilisée à droite si la gauche l'a établie
if (obj instanceof String s && s.length() > 3) {
    System.out.println(s.substring(0, 3)); // s en scope
}

// Avec ||, la variable de pattern n'est PAS sûre à droite (le short-circuit peut empêcher de
if (obj instanceof String s || s.length() > 3) {  // ❌ s n'est pas en scope ici
    // ...
}

// Les parenthèses peuvent aider à regrouper la logique
if ((obj instanceof String s) && s.contains("a")) { // ✅ s en scope après le test groupé
    System.out.println(s);
}
```

Le pattern matching avec `null` s'évalue, comme toujours pour `instanceof`, à `false` :

```java
String str = null;

// instanceof classique
if (str instanceof String) {
    System.out.print("NOT EXECUTED"); // instanceof vaut false
}

// Pattern matching
if (str instanceof String s) {
    System.out.print("NOT EXECUTED"); // instanceof vaut toujours false
}
```

Types supportés :

Le type de la variable de pattern doit être un sous-type, un super-type, ou le même type que la variable référence.

```java
Number num = Short.valueOf(10);

if (num instanceof String s) {}  // ❌ Erreur de compilation
if (num instanceof Short s) {}   // ✅ Ok
if (num instanceof Object s) {}  // ✅ Ok
if (num instanceof Number s) {}  // ✅ Ok
```

5.7.6.4 Tableaux et types réifiables

`instanceof` fonctionne avec les tableaux (réifiables) et avec des formes génériques effacées ou avec wildcard.
Les **types réifiables** sont ceux dont la représentation à l'exécution conserve pleinement leur type (par exemple : raw types, tableaux, classes non génériques, wildcard `?`).
À cause de l'effacement de type (*type erasure*), `List<String>` ne peut pas être testée directement à l'exécution.

```java
Object arr = new int[]{1,2,3};
System.out.println(arr instanceof int[]); // true

Object list = java.util.List.of(1,2,3);
// System.out.println(list instanceof List<Integer>); // ❌ Erreur de compilation : type para
System.out.println(list instanceof java.util.List<?>); // ✅ true
```

5.8 Opérateur ternaire

L'**opérateur ternaire** (`? :`) est le seul opérateur en Java qui prend **trois opérandes**.
Il constitue une forme concise de l'instruction `if-else`.

5.8.1 Règles de Typage de l'Opérateur Ternaire

Le type d'une expression conditionnelle (ternaire) est déterminé par les types du deuxième et du troisième opérande.

5.8.1.1 Opérandes Numériques

- Si un opérande est de type `byte` et l'autre de type `short`, le type résultant est `short`.
- Si un opérande est de type `T` (`byte`, `short` ou `char`) et l'autre est une expression constante de type `int` dont la valeur est représentable dans `T`, alors le type résultant est `T`.
- Dans tous les autres cas numériques, la **binary numeric promotion** est appliquée aux deux opérandes.
  Le type de l'expression conditionnelle devient le type promu.

> La binary numeric promotion inclut la **conversion d'unboxing** et la **value set conversion**.

5.8.1.2 Types de Référence

- Si un opérande est `null` et l'autre est un type de référence, le type résultant est ce type de référence.

- Si les deux opérandes sont de types de référence différents, l'un doit être assignable à l'autre (compatibilité d'assignation).
  Le type résultant est le type le plus général, c'est-à-dire celui auquel l'autre peut être assigné.
- Si aucun des deux types n'est compatible par assignation avec l'autre, une **erreur à la compilation** se produit.

En résumé, l'opérateur ternaire détermine son type en appliquant :

- Des règles spécifiques de narrowing pour les petits types entiers
- La binary numeric promotion pour les valeurs numériques
- Les règles de compatibilité d'assignation pour les types de référence

> **Tip**
>
> L'opérateur ternaire **doit** produire une valeur d'un type compatible. Si les deux branches retournent des types non liés, la compilation échoue.
>
> ```
> String s = true ? "ok" : 5; // ❌ erreur de compilation : types incompatibles
> ```

5.8.2 Syntaxe

```
condition ? expressionIfTrue : expressionIfFalse;
```

5.8.3 Exemple

```
int age = 20;
String access = (age >= 18) ? "Autorisé" : "Refusé";
System.out.println(access);  // "Autorisé"
```

5.8.4 Exemple de Ternaire Imbriqué

```
int score = 85;
String grade = (score >= 90) ? "A" :
               (score >= 75) ? "B" :
               (score >= 60) ? "C" : "F";
System.out.println(grade);  // "B"
```

5.8.5 Remarques

> **Warning**
>
> - Les expressions ternaires imbriquées peuvent réduire la lisibilité. Utilisez des parenthèses pour plus de clarté.
> - L'opérateur ternaire retourne une **valeur**, contrairement à `if-else`, qui est une instruction.

◀ 4. Types de données Java et cast | ▲ Index | 6. Instanciation des types ▶

6. Instanciation des types

Table des matières

6.1 Introduction

En Java, un **type** peut être soit un **type primitif** (comme `int`, `double`, `boolean`, etc.), soit un **type référence** (classes, interfaces, tableaux, enums, records, etc.). Voir : Java Data Types and Casting

La façon dont les instances sont créées dépend de la catégorie du type :

- **Types primitifs**
  Les instances des types primitifs sont créées simplement en déclarant une variable.
  La JVM alloue automatiquement la mémoire nécessaire pour contenir la valeur, et aucun mot-clé explicite n'est requis.

```
int age = 30;          // crée un int primitif avec la valeur 30
boolean flag = true;   // crée un boolean primitif avec la valeur true
double pi = 3.14159;   // crée un double primitif avec la valeur 3.14159
```

- **Types référence (objets)**
  Les instances des types classe sont créées à l'aide du mot-clé `new` (sauf quelques cas particuliers comme les littéraux String, les records avec constructeur canonique, ou les méthodes factory). Le mot-clé `new` alloue de la mémoire sur le tas (heap) et invoque un constructeur de la classe.

```
String name = new String("Alice"); // crée explicitement un nouvel objet String
Person p = new Person();           // crée un nouvel objet Person via son constructeur
```

Il est aussi courant de s'appuyer sur des littéraux ou des méthodes factory pour créer des objets.

```
String text = "Hello World";

List<String> list = List.of("A", "B", "C");              // factory method immutabile
Map<String, Integer> map = Map.of("one", 1, "two", 2);   // factory method immutabile
Optional<String> opt = Optional.of("value");             // factory method

LocalDate date = LocalDate.of(2025, 3, 15);
Integer boxed = Integer.valueOf(10);
```

> **Important**
>
> Les littéraux String **ne nécessitent pas** `new` et sont stockés dans le **String pool**. Utiliser `new String("x")` crée toujours un nouvel objet sur le heap.

6.1.1 Gestion des types primitifs

6.1.1.1 Déclarer un primitif

Déclarer un type primitif (comme pour les types référence) signifie réserver un espace mémoire pour une variable d'un type donné, sans nécessairement lui attribuer une valeur.

> **Warning**
>
> Contrairement aux primitifs, dont la taille dépend du type spécifique (par ex. `int` vs `long`), les variables référence occupent toujours la même taille fixe en mémoire — ce qui varie, c'est la taille de l'objet qu'elles pointent.

- Exemples de syntaxe (déclaration uniquement) :

```
int number;

boolean active;

char letter;

int x, y, z;          // Déclarations multiples dans une seule instruction : Java autorise la
```

> **Important**
>
> Les `modificateurs` et le `type` déclarés au début d'une déclaration de variables s'appliquent à toutes les variables déclarées dans la même instruction.
>
> **Exception** : lors de la déclaration de tableaux en utilisant les crochets après le nom de la variable, les crochets font partie du déclarateur de la variable individuelle, et non du type de base.

- Examples

```
static int a, b, c;

// est équivalent à :

static int a;
static int b;
static int c;


int[] a, b;   // les deux sont des tableaux de int
int c[], d;   // seul c est un tableau, d est un int normal
```

6.1.1.2 Affecter un primitif

Affecter un type primitif (comme pour les types référence) signifie stocker une valeur dans une variable déclarée de ce type.
Pour les primitifs, la variable contient la valeur elle-même, tandis que pour les types référence elle contient l'adresse mémoire (une référence) de l'objet pointé.

- Exemples de syntaxe :

```
int number;                    // Déclaration d'un int : une variable appelée "number"

number = 10;                   // Affectation de la valeur 10 à cette variable

char letter = 'A';             // Déclaration et affectation en une seule instruction : décla

int a1, a2;                     // Déclarations multiples

int a = 1, b = 2, c = 3;       // Déclarations multiples & affectations

char b1, b2, b3 = 'C';         // Déclarations mixtes (2 déclarations + 1 affectation)

double d1, double d2;           // ERROR - NOT LEGAL

int v1; v2;                     // ERROR - NOT LEGAL
```

> **Important**
>
> Quand tu écris un nombre directement dans le code (un littéral numérique), Java suppose par défaut qu'il est de type **int**. Si la valeur ne tient pas dans un `int`, le code ne compile pas, à moins de le marquer explicitement avec le suffixe approprié.

- Exemple de syntaxe pour un littéral numérique :

```
long exNumLit = 5729685479; // ❌ Does not compile.
                            // Même si la valeur pourrait tenir dans un long,
                            // un littéral numérique simple est considéré comme un int,
                            // et ce nombre est trop grand pour un int.

// Changing the declaration adding the correct suffix (L or l) will solve:

long exNumLit = 5729685479L;

or

long exNumLit = 5729685479l;
```

Déclarer un type `reference` signifie réserver de l'espace mémoire pour une variable qui contiendra une référence (pointeur) vers un objet du type spécifié.

À ce stade, aucun objet n'est encore créé — la variable a seulement la capacité d'en référencer un.

> **Warning**
>
> Contrairement aux primitifs, dont la taille dépend du type spécifique (par ex. `int` vs `long`), les variables référence occupent toujours la même taille fixe en mémoire (suffisante pour stocker une référence). Ce qui varie, c'est la taille de l'objet pointé, qui est alloué séparément sur le heap.

- Exemples de syntaxe (déclaration uniquement) :

```
String name;
Person person;
List<Integer> numbers;

Person p1, p2, p3;   // Déclarations multiples dans une seule instruction

String a = "abc", b = "def", c = "ghi";     // Déclarations multiples & affectations

String b1, b2, b3 = "abc"                   // Déclarations mixtes (b1, b2) avec une affectati

String d1, String d2;                   // ERROR - NOT LEGAL

String v1; v2;                          // ERROR - NOT LEGAL
```

6.1.2 Gestion des types référence

6.1.2.1 Créer et affecter une référence

Affecter un type `reference` signifie stocker dans la variable l'adresse mémoire d'un objet.

On le fait normalement après la création de l'objet avec le mot-clé **new** et un constructeur, ou en utilisant un littéral ou une méthode factory.

Une référence peut aussi être affectée à un autre objet du même type ou d'un type compatible.

Les types référence peuvent également recevoir **null**, ce qui signifie qu'ils ne référencent aucun objet.

- Exemples de syntaxe :

```
Person person = new Person(); // Exemple avec 'new' et un constructeur 'Person()' :
                              // 'new Person()' crée un nouvel objet Person sur le heap
                              // et renvoie sa référence, stockée dans la variable 'person'.

String greeting = "Hello";   // Exemple avec un littéral (pour String).

List<Integer> numbers = List.of(1, 2, 3);   // Exemple avec une méthode factory.
```

6.1.2.2 Constructeurs

Dans l'exemple, `Person()` est un constructeur — un type spécial de méthode utilisée pour initialiser de nouveaux objets.

Chaque fois que tu appelles `new Person()`, le constructeur s'exécute et configure l'instance nouvellement créée.

Les constructeurs ont trois caractéristiques principales :

- Le nom du constructeur **doit correspondre exactement au nom de la classe** (sensible à la casse).
- Les constructeurs **ne déclarent pas de type de retour** (pas même `void`).
- Si tu ne définis aucun constructeur dans ta classe, le compilateur fournit automatiquement un **constructeur par défaut sans argument** qui ne fait rien.

> **Warning**
>
> Si tu vois une méthode qui a le même nom que la classe **mais qui déclare un type de retour**, ce n'est **pas** un constructeur. C'est simplement une méthode ordinaire (même si commencer les noms de méthodes par une majuscule va à l'encontre des conventions de nommage Java).

Le **but d'un constructeur** est d'initialiser l'état d'un objet nouvellement créé — généralement en affectant des valeurs à ses champs, soit avec des valeurs par défaut, soit à partir de paramètres passés au constructeur.

- Exemple 1 : Constructeur par défaut (sans paramètres)

```
public class Person {
    String name;
    int age;

    // Default constructor
    public Person() {
        name = "Unknown";
        age = 0;
    }
}

Person p1 = new Person(); // name = "Unknown", age = 0
```

- Exemple 2 : Constructeur avec paramètres

```
public class Person {
    String name;
    int age;

    // Constructor with parameters
    public Person(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

Person p2 = new Person("Alice", 30); // name = "Alice", age = 30
```

- Exemple 3 : Plusieurs constructeurs (surcharge de constructeurs)

```java
public class Person {
    String name;
    int age;

    // Default constructor
    public Person() {
        this("Unknown", 0); // calls the other constructor
    }

    // Constructor with parameters
    public Person(String newName, int newAge) {
        name = newName;
        age = newAge;
    }
}

Person p1 = new Person();            // name = "Unknown", age = 0
Person p2 = new Person("Bob", 25);   // name = "Bob", age = 25
```

> **Important**
>
> - Les constructeurs ne sont pas hérités : si une superclasse définit des constructeurs, ils ne sont pas automatiquement disponibles dans la sous-classe — tu dois les déclarer explicitement.
> - Si tu déclares un constructeur quelconque dans une classe, le compilateur ne génère pas le constructeur par défaut sans argument : si tu as encore besoin d'un constructeur sans argument, tu dois le déclarer manuellement.

6.1.2.3 Blocs d'initialisation d'instance

En plus des constructeurs, Java fournit un mécanisme appelé **initializer blocks** pour aider à initialiser les objets.
Ce sont des blocs de code à l'intérieur d'une classe, entourés par `{ }`, qui s'exécutent **à chaque création d'instance**, juste avant l'exécution du corps du constructeur.

Caractéristiques

- Aussi appelés **instance initializer blocks**.
- Exécutés, avec les initialisations de champs, dans l'ordre où ils apparaissent dans la définition de la classe, mais toujours avant les constructeurs.
- Utiles lorsque plusieurs constructeurs doivent partager un code d'initialisation commun.

Exemple : utilisation d'un Instance Initializer Block

```java
public class Person {
    String name;
    int age;

    // Instance initializer block
    {
        System.out.println("Instance initializer block executed");
        age = 18; // default age for every Person
    }

    // Default constructor
    public Person() {
        name = "Unknown";
    }

    // Constructor with parameters
    public Person(String newName) {
        name = newName;
    }
}

Person p1 = new Person();          // prints "Instance initializer block executed"
Person p2 = new Person("Alice");   // prints "Instance initializer block executed"
```

> **Note**
>
> Dans cet exemple, le bloc d'initialisation s'exécute avant le corps de chacun des constructeurs. p1 et p2 commenceront tous les deux avec age = 18, quel que soit le constructeur utilisé.

Plusieurs blocs d'initialisation : si une classe contient plusieurs blocs d'initialisation, ils s'exécutent dans l'ordre où ils apparaissent dans le fichier source :

- Exemple :

```
public class Example {
    {
        System.out.println("First block");
    }

    {
        System.out.println("Second block");
    }
}

Example ex = new Example();
// Output:
// First block
// Second block
```

> **Note**
>
> Les blocs d'initialisation d'instance sont moins courants en pratique, car une logique similaire peut souvent être placée directement dans les constructeurs. Il est important de savoir que : - Ils s'exécutent toujours avant le corps du constructeur. - Ils sont exécutés dans l'ordre de déclaration dans la classe. - Ils peuvent être combinés avec les constructeurs pour éviter la duplication de code.

> **Warning**
>
> **Ordre d'initialisation lors de la création d'un objet** 1. Champs statiques 2. Blocs d'initialisation statiques 3. Champs d'instance 4. Blocs d'initialisation d'instance 5. Corps du constructeur

---

6.2 Initialisation par défaut des variables

6.2.1 Variables d'instance et de classe

- Une **variable d'instance (un champ/field)** est une valeur définie dans une instance d'un objet ;
- Une **variable de classe** (définie avec le mot-clé **static**) est définie au niveau de la classe et est partagée entre tous les objets (instances de la classe)

Les variables d'instance et de classe reçoivent une valeur par défaut, par le compilateur, si elles ne sont pas initialisées.

- Tableau des valeurs par défaut pour les variables d'instance et de classe :

Type	Default Value
Object	null
Numeric	0
boolean	false
char	'000' (NUL)

6.2.2 Variables d'instance final

Contrairement aux variables d'instance et de classe ordinaires, les variables `final` **ne sont pas initialisées par défaut par le compilateur**.
Une variable `final` **doit être affectée explicitement exactement une fois**, sinon le code ne compile pas.

Cela s'applique aux :

- **variables final d'instance**
- **variables de classe static final**

> **Note**
>
> On peut affecter une valeur `null` à une variable final d'instance ou de classe tant que cela est fait explicitement.

Java impose cette règle car une variable `final` représente une valeur qui doit être *connue et fixée* avant usage.

`Final Instance Variables`

Une **variable final d'instance** doit être affectée **exactement une fois**, et l'affectation doit se produire dans *un* des endroits suivants :

1. **Au point de déclaration**
2. **Dans un bloc d'initialisation d'instance**
3. **À l'intérieur de chaque constructeur**

Si la classe a *plusieurs constructeurs*, la variable doit être affectée dans **tous**.

- Exemple :

```
public class Person {
    final int id;   // doit être affectée avant la fin du constructeur
    String name;

    // Constructeur 1
    public Person(int id, String name) {
        this.id = id;        // ok
        this.name = name;
    }

    // Constructeur 2
    public Person() {
        this.id = 0;         // requis aussi ici
        this.name = "Unknown";
    }
}
```

> **Warning**
>
> Compiler sans affecter `id` dans **chaque** constructeur produit une erreur de compilation : variable id might not have been initialized

Variables de classe `static final` (constantes)

Une **variable static final** appartient à la classe plutôt qu'à une instance.
Elle doit aussi être affectée exactement une fois, mais l'affectation peut se faire à l'un des endroits suivants :

1. **Au point de déclaration**
2. **Dans un bloc d'initialisation statique**

- Exemple :

```java
public class AppConfig {

    static final int TIMEOUT = 5000;    // affectée à la déclaration

    static final String VERSION;        // affectée dans le bloc static

    static {
        VERSION = "1.0.0";              // ok
    }
}
```

Tenter d'affecter une `static final` dans un constructeur est illégal.

Règles clés pour les champs `final`

Scenario	Allowed?	Notes
Assign at declaration	✓	Most common pattern
Assign in constructor	✓	All constructors must assign it
Assign in instance initializer	✓	Before constructor body runs
Assign in static initializer (static final only)	✓	For class-level constants
Assign multiple times	❌	Compilation error
Default initialization	❌	Must be explicitly assigned

Exemple d'une situation **illégale** :

```java
public class Example {
    final int x;        // not initialized
}

Example e = new Example(); // ❌ compile-time error
```

Pourquoi les variables `final` ne sont-elles pas initialisées par défaut ?

Parce que :

- Leur valeur doit être **connue et immuable**, et
- Java doit garantir que la valeur est définie **avant utilisation**,
- Une initialisation par défaut créerait une situation où `0`, `null` ou `false` pourraient devenir involontairement la valeur permanente.

Ainsi, Java oblige les développeurs à initialiser explicitement les champs `final`.

> **Tip**
>
> `final` signifie **affecté une seule fois**, pas **objet immuable**.
>
> Une référence final peut toujours pointer vers un objet mutable.

```java
final List<String> list = new ArrayList<>();
list.add("ok");      // autorisé
list = new ArrayList<>(); // ❌ impossible de réaffecter la référence
```

6.2.3 Variables locales

Les **variables locales** sont des variables définies dans un constructeur, une méthode ou un bloc d'initialisation ;

Les variables locales n'ont pas de valeurs par défaut et doivent être initialisées avant d'être utilisées. Si tu essaies d'utiliser une variable locale non initialisée, le compilateur signalera une ERREUR.

- Exemple

```
public int localMethod {

    int firstVar = 25;
    int secondVar;
    secondVar = 35;
    int firstSum = firstVar + secondVar;    // OK variables are both initialized before use

    int thirdVar;
    int secondSum = firstSum + thirdVar;    // ERROR: variable thirdVar has not been initializ
}
```

6.2.3.1 Inférer les types avec var

Dans certaines conditions, tu peux utiliser le mot-clé **var** à la place du type approprié lors de la déclaration de variables **locales** ;

> **Warning**
>
> - **var** N'EST PAS un mot réservé en Java ;
> - **var** ne peut être utilisé que pour les variables locales : il NE PEUT PAS être utilisé pour les **paramètres de constructeur**, les **variables d'instance** ou les **paramètres de méthode** ;
> - Le compilateur infère le type en regardant UNIQUEMENT le code **sur la ligne de déclaration** ; une fois le bon type inféré, tu ne peux pas réaffecter avec un autre type.

- Exemple

```
public int localMethod {

    var inferredInt = 10;   // The compiler infer int by the context;
    inferredInt = 25;       // OK

    inferredInt = "abcd";   // ERROR: the compiler has already inferred the type of the variab

    var notInferred;
    notInferred = 30;       // ERROR: in order to infer the type, the compiler looks ONLY at t

    var first, second = 15; // ERROR: var cannot be used to define two variables on the same s

    var x = null;           // ERROR: var cannot be initialized with null but it can be reassi
}
```

> **Warning**
>
> Les variables locales **n'obtiennent jamais** de valeurs par défaut. Les champs d'instance et statiques **en obtiennent toujours**.

6.3 Types wrapper

En Java, les **types wrapper** sont des représentations objet des huit types primitifs.
Chaque primitif a une classe wrapper correspondante dans le package `java.lang` :

Primitive	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Les objets wrapper sont immuables — une fois créés, leur valeur ne peut pas changer.

6.3.1 Objectif des types wrapper

- Permettre d'utiliser des primitifs dans des contextes qui exigent des objets (par ex. collections, generics).
- Fournir des méthodes utilitaires pour parser, convertir et manipuler des valeurs.
- Supporter des constantes comme `Integer.MAX_VALUE` ou `Double.MIN_VALUE`.

6.3.2 Autoboxing et unboxing

Depuis Java 5, le compilateur convertit automatiquement entre primitifs et wrappers :

- **Autoboxing** : primitif → wrapper
- **Unboxing** : wrapper → primitif

```
Integer i = 10;       // autoboxing: int → Integer
int n = i;            // unboxing: Integer → int

Integer int1 = Integer.valueOf(11);
long long1 = int1;  // Unboxing --> implicit cast OK

Long long2 = 11;    // ❌ Does not compile.
                    // 11 is an int literal → requires autoboxing + widening → illegal

Character char1 = null;
char char2 = char1;  // WARNING: NullPointerException

Integer  arr1 = {11.5, 13.6}  // WARNING: Does not compile!!
Double[] arr2 = {11, 22};     // WARNING: Does not compile!!
```

> **Tip**
>
> Java **ne réalise jamais** autoboxing + widening/narrowing en une seule étape.

> **Warning**
>
> - **AUTOBOXING** et **cast implicite** ne sont pas autorisés dans la même instruction : tu ne peux pas faire les deux en même temps. (voir l'exemple ci-dessus)
> - Cette règle s'applique aussi aux appels de méthode.

6.3.3 Parsing et conversion

Les wrappers fournissent des méthodes statiques pour convertir des chaînes ou d'autres types en primitifs :

```java
int x = Integer.parseInt("123");    // returns primitive int
Integer y = Integer.valueOf("456"); // returns Integer object
double d = Double.parseDouble("3.14");

// On the numeric wrapper class valueOf() throws a NumberFormatException on invalid input.
// Example:

Integer w = Integer.valueOf("two"); // NumberFormatException

// On Boolean, the method returns Boolean.TRUE for any value that matches "true" ignoring case
// Example:

Boolean.valueOf("true");    // true
Boolean.valueOf("TrUe");    // true
Boolean.valueOf("TRUE");    // true
Boolean.valueOf("false");   // false
Boolean.valueOf("FALSE");   // false
Boolean.valueOf("xyz");     // false
Boolean.valueOf(null);      // false

// The numeric integral classes Byte, Short, Integer and Long include an overloaded **valueOf(
// Example with base 16 (hexadecimal) which includes character 0 -> 9 and A -> F (ignore case)

Integer.valueOf("6", 16);   // 6
Integer.valueOf("a", 16);   // 10
Integer.valueOf("A", 16);   // 10
Integer.valueOf("F", 16);   // 15
Integer.valueOf("G", 16);   // NumberFormatException
```

> **Note**
>
> Les méthodes **parseXxx()** renvoient un primitif tandis que **valueOf()** renvoie un objet wrapper.

6.3.4 Méthodes utilitaires

Toutes les classes wrapper numériques étendent la classe `Number` et héritent donc de méthodes utilitaires telles que : `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, `doubleValue()`.

Les classes wrapper `Boolean` et `Character` incluent : `booleanValue()` et `charValue()`.

- Exemple :

```java

// In trying to convert those helper methods can result in a loss of precision.

Double baseDouble = Double.valueOf("300.56");

double wrapDouble = baseDouble.doubleValue();
System.out.println("baseDouble.doubleValue(): " + wrapDouble);  // 300.56

byte wrapByte = baseDouble.byteValue();
System.out.println("baseDouble.byteValue(): " + wrapByte);      // 44  -> There is no 300 in b

int wrapInt = baseDouble.intValue();
System.out.println("baseDouble.intValue(): " + wrapInt);        // 300 -> The value is truncat
```

6.3.5 Valeurs null

Contrairement aux primitifs, les wrappers peuvent contenir **null**.
Tenter d'unboxer `null` provoque une NullPointerException :

```java
Integer val = null;
int z = val; // ❌ NullPointerException at runtime
```

6.4 Égalité en Java

Java fournit deux mécanismes différents pour vérifier l'égalité :

- `==` (opérateur d'égalité)
- `.equals()` (méthode définie dans `Object` et redéfinie dans de nombreuses classes)

Comprendre la différence est essentiel.

6.4.1 Égalité avec les types primitifs

Pour les **valeurs primitives** (`int` , `double` , `char` , `boolean` , etc.),
l'opérateur `==` compare leur **valeur numérique ou booléenne** réelle.

Exemple :

```
int a = 5;
int b = 5;
System.out.println(a == b);     // true

char c1 = 'A';
char c2 = 65;                   // same Unicode code point
System.out.println(c1 == c2);   // true
```

6.4.1.1 Points clés

- `==` effectue une **comparaison de valeur** pour les primitifs.
- Les types primitifs n'ont **pas** de méthode `.equals()` .
- Les types primitifs mixtes suivent les **règles de promotion** numérique
  (par ex. `int == long` → `int` promu en `long`).

6.4.2 Égalité avec les types référence

Avec les objets (types référence), la signification de `==` change.

6.4.2.1 `==` (Comparaison d'identité)

`==` vérifie si **deux références pointent vers le même objet en mémoire**.

```
String s1 = new String("Hi");
String s2 = new String("Hi");

System.out.println(s1 == s2);      // false → objets différents
```

Même si les contenus sont identiques, `==` est false sauf si les deux variables référencent
le même objet exact.

6.4.2.2 `.equals()` (Comparaison logique)

De nombreuses classes redéfinissent `.equals()` pour comparer les **valeurs**, pas les adresses mémoire.

```
System.out.println(s1.equals(s2)); // true → même contenu
```

6.4.2.3 Points clés

- `.equals()` est définie dans `Object` .
- Si une classe ne redéfinit *pas* `.equals()` , elle se comporte comme `==` .
- Des classes comme `String` , `Integer` , `List` , etc. redéfinissent `.equals()`
  pour fournir une comparaison de valeur pertinente.

6.4.3 String Pool et égalité

Les littéraux String sont stockés dans le **String pool**, donc des littéraux identiques
référencent **le même objet**.

```java
String a = "Java";
String b = "Java";
System.out.println(a == b);       // true → même littéral dans le pool
```

Mais l'utilisation de `new` crée un objet différent :

```java
String x = new String("Java");
String y = "Java";

System.out.println(x == y);       // false → x n'est pas dans le pool
System.out.println(x.equals(y));  // true
```

Pièges courants

```java
String x = "Java string literal";
String y = " Java string literal".trim();

System.out.println(x == y);       // false → x et y ne sont pas identiques à la compilation

String a = "Java string literal";
String b = "Java ";
b += "string literal";

System.out.println(a == b);  // false
```

> **Warning**
>
> Toute String créée à **l'exécution** n'entre *pas* automatiquement dans le pool. Utilise `intern()` si tu veux le pooling.

> **Tip**
>
> `"Hello" == "Hel" + "lo"` → true (constante à la compilation)
>
> `"Hello" == getHello()` → false (concaténation à l'exécution)

```java
String x = "Hello";
String y = "Hel" + "lo";   // compile-time → même littéral
String z = "Hel";
z += "lo";                 // runtime → nouvelle String

System.out.println(x == y); // true
System.out.println(x == z); // false
```

6.4.3.1 La méthode intern

Tu peux aussi demander à Java d'utiliser une String du String Pool (si elle existe déjà) via la méthode `intern()` :

```java
String x = "Java";
String y = new String("Java").intern();

System.out.println(x == y);       // true
```

6.4.4 Égalité avec les types wrapper

Les classes wrapper (`Integer` , `Double` , `Boolean` , etc.) se comportent comme des objets normaux.

Par conséquent :

- `==` → compare **les références des objets**
- `.equals()` → compare **les valeurs numériques**

Exemple :

```java
Integer a = 100;
Integer b = 100;
System.out.println(a == b);        // true → cached

Integer c = 1000;
Integer d = 1000;
System.out.println(c == d);        // false → objets différents

System.out.println(c.equals(d));   // true → même valeur numérique
```

Puisque, comme rappelé précédemment, toutes les classes wrapper sont **immutables**, leur valeur interne **ne peut pas être modifiée** une fois créées.

Les opérations qui semblent modifier un wrapper créent en réalité **un nouvel objet**.

Exemple :

```java
Integer i = 5;
i++;
```

Cela est conceptuellement équivalent à :

```java
i = Integer.valueOf(i.intValue() + 1);
```

Donc un **nouvel objet** `Integer` est créé et assigné à `i`.

6.4.4.1 Mise en cache des wrapper

Pour réduire l'utilisation de la mémoire et la création d'objets, Java **réutilise certaines instances de wrapper**.

Les valeurs suivantes sont mises en cache :

- Toutes les valeurs `Boolean` (`true` et `false`)
- Toutes les valeurs `Byte`
- Toutes les valeurs `Character` de `\u0000` à `\u007f` (0–127)
- Toutes les valeurs `Short` de **–128 à 127**
- Toutes les valeurs `Integer` de **–128 à 127**

À cause de ce mécanisme de mise en cache :

```java
Integer a = 100;
Integer b = 100;

System.out.println(a == b);   // true
```

Les deux variables font référence **au même objet mis en cache**.

Cependant, les valeurs en dehors de l'intervalle du cache produisent **des objets distincts** :

```java
Integer c = 1000;
Integer d = 1000;

System.out.println(c == d);   // false
System.out.println(c.equals(d)); // true
```

6.4.4.2 Le mot-clé `new` contourne le cache

Lorsqu'un objet wrapper est créé en utilisant `new`, **une nouvelle instance est toujours créée**, même si une valeur mise en cache existe.

Exemple :

```java
Integer i = 10;          // objet mis en cache
Integer j = 10;          // même objet mis en cache
Integer k = new Integer(10); // nouvel objet (non mis en cache)

System.out.println(i == j); // true
System.out.println(i == k); // false
```

Cependant, les constructeurs des wrapper ont été **dépréciés en Java 9** et marqués pour suppression.
Le code moderne devrait utiliser **l'autoboxing** ou des méthodes factory comme `Integer.valueOf()`.

6.4.4.3 Comparaison des wrapper

Lorsque deux références wrapper sont comparées en utilisant `==`, le résultat dépend du fait qu'elles **référencent le même objet**, et non du fait que leurs valeurs soient égales.

Par conséquent, les tests d'égalité entre wrapper devraient normalement utiliser :

```java
equals()
```

au lieu de `==`.

6.4.4.4 Des types wrapper différents ne peuvent pas être comparés

Les objets wrapper de **types différents** ne peuvent pas être comparés en utilisant `==`.

Exemple :

```java
Byte b = 1;
Integer i = 1;

b == i;   // erreur de compilation
```

Les opérandes doivent être **des types compatibles**, sinon la comparaison n'est pas valide.

> **Warning**
>
> Faire très attention lorsque l'on compare des objets wrapper avec `==`.
> À cause du caching des wrapper, les comparaisons peuvent parfois retourner `true` et parfois `false` selon la valeur.

6.4.5 Égalité et `null`

- `== null` est toujours sûr.
- Appeler `.equals()` sur une référence `null` déclenche une `NullPointerException`.

```java
String s = null;
System.out.println(s == null);   // true
// s.equals("Hi");               // ❌ NullPointerException
```

6.4.6 Tableau récapitulatif

Comparison	Primitives	Objects / Wrappers	Strings
`==`	compares **value**	compares **reference**	identity (affected by String pool)
`.equals()`	N/A	compares **content** if overridden	**content** comparison

---

Control Flow

7. Flux de contrôle

Table des matières

Le **flux de contrôle** en Java fait référence à l'**ordre dans lequel les instructions individuelles, les commandes ou les appels de méthode sont exécutés** pendant l'exécution du programme.

Par défaut, les instructions s'exécutent séquentiellement de haut en bas, mais les instructions de contrôle du flux permettent au programme de **prendre des décisions**, **répéter des actions** ou **dériver les chemins d'exécution** en fonction de conditions.

Java fournit trois grandes catégories de constructions de contrôle du flux :

- **Instructions décisionnelles** — `if`, `if-else`, `switch`
- **Instructions de boucle** — `for`, `while`, `do-while` et le `for` amélioré
- **Instructions de branchement** — `break`, `continue` et `return`

> **Tip**
>
> Comprendre le flux de contrôle est essentiel pour voir comment les données circulent dans votre programme et comment chaque décision logique est évaluée étape par étape.

7.1 L'instruction `if`

L'instruction `if` est une structure conditionnelle de contrôle du flux qui exécute un bloc de code uniquement si une expression booléenne spécifiée est évaluée à `true`. Elle permet au programme de prendre des décisions à l'exécution.

Syntaxe :

```
if (condition) {
    // exécuté uniquement lorsque la condition est true
}
```

Une clause `else` optionnelle gère le chemin alternatif :

```java
if (score >= 60) {
    System.out.println("Passed");
} else {
    System.out.println("Failed");
}
```

Plusieurs conditions peuvent être chaînées à l'aide de `else if` :

```java
if (grade >= 90) {
    System.out.println("A");
} else if (grade >= 80) {
    System.out.println("B");
} else if (grade >= 70) {
    System.out.println("C");
} else {
    System.out.println("D or below");
}
```

> **Note**
>
> La condition de `if` doit être évaluée comme un **boolean** ; les types numériques ou les objets ne peuvent pas être utilisés directement comme conditions.
>
> Les accolades `{}` sont facultatives pour une seule instruction mais sont fortement recommandées afin d'éviter des erreurs logiques subtiles.
>
> Une chaîne `if-else` est évaluée de haut en bas, et seul le premier branchement dont la condition est évaluée à `true` est exécuté.

7.2 L'instruction `switch` & l'expression

La construction `switch` est une structure de contrôle du flux qui sélectionne une branche parmi plusieurs alternatives en fonction de la valeur d'une expression (le **selector**).

Comparé aux longues chaînes de `if-else-if`, un `switch` :

- Est souvent **plus facile à lire** lorsqu'on teste de nombreuses valeurs discrètes (constantes, enums, chaînes).
- Peut être **plus sûr et plus concis** lorsqu'il est utilisé comme **expression switch**

parce que :

- Il produit une valeur.
- Le compilateur peut imposer l'**exhaustivité** et la **cohérence de type**.

Java 21 prend en charge :

- Le `switch` classique en tant qu'**instruction** (contrôle du flux uniquement).
- Le `switch` en tant qu'**expression** (produit un résultat).
- **Le pattern matching** dans `switch`, y compris les type patterns et les guards.

Les deux formes de `switch` partagent les mêmes règles concernant le selector (la **variable cible** du switch) et les valeurs case acceptables.

7.2.1 La `variable cible` du switch peut être

Control Variable type
`byte` / `Byte`
`short` / `Short`
`char` / `Character`
`int` / `Integer`
`String`
Enum types (selectors of an `enum`)
Any reference type (with pattern matching)
`var` (if it resolves to one of the allowed types)

> **Warning**
>
> **Non autorisé** comme type de selector pour switch :
>
> - `boolean`
> - `long`
> - `float`
> - `double`

7.2.2 Valeurs `case` acceptables

Pour un switch non pattern, chaque étiquette `case` doit être une constante à la compilation compatible avec le type du selector.

Autorisé comme étiquettes case :

- **Littéraux** tels que `0`, `'A'`, `"ON"`.
- **Constantes enum**, par ex. `RED` ou `Color.GREEN`.
- **Variables constantes final** (constantes à la compilation).

Une variable constante à la compilation :

- Doit être déclarée avec `final` et initialisée dans la même instruction.
- Son initialiseur doit lui-même être une expression constante (généralement en utilisant des littéraux et d'autres constantes à la compilation).

7.2.3 Compatibilité de type entre selector et case

Le type du selector et chaque étiquette `case` doivent être compatibles :

- Les constantes numériques des case doivent être dans l'intervalle du type du selector.
- Pour un selector `enum`, les étiquettes case doivent être des constantes de cet `enum`.
- Pour un selector `String`, les étiquettes case doivent être des constantes de chaîne.

7.2.4 Pattern Matching dans Switch

Le switch en Java 21 prend en charge le pattern matching, y compris :

- **Type patterns** : `case String s`
- **Patterns avec garde** : `case String s when s.length() > 3`
- **Pattern null** : `case null`

Exemple :

```
String describe(Object o) {
    return switch (o) {
        case null -> "null";
        case Integer i -> "int " + i;
        case String s when s.isEmpty() -> "empty string";
        case String s -> "string (" + s.length() + ")";
        default -> "other";
    };
}
```

Points clés :

- Chaque pattern introduit une variable de pattern (comme `i` ou `s`).
- Les variables de pattern sont dans la portée uniquement à l'intérieur de leur arm (ou des chemins où le pattern est connu comme correspondant).
- L'ordre est important en raison de la **dominance** : les patterns plus spécifiques doivent précéder les plus généraux.

7.2.4.1 Noms de variables et portée entre les branches

Avec le pattern matching, la variable de pattern n'existe que dans la portée de l'arm dans lequel elle est définie. Cela signifie que vous pouvez réutiliser le même nom de variable dans différents branches case.

- Exemple :

```
switch (o) {
    case String str -> System.out.println(str.length());
    case CharSequence str -> System.out.println(str.charAt(0));
    default -> { }
}
```

> **Note**
>
> Ce dernier exemple ne retourne pas de valeur, il s'agit donc d'un **switch instruction** et non d'une expression switch.

7.2.4.2 Ordonnancement, dominance et exhaustivité dans les switch à patterns

Lorsqu'on utilise le pattern matching, l'ordre des branches est crucial en raison de la **dominance** et du potentiel **code inatteignable**.

Un pattern plus général ne doit **pas** apparaître avant un pattern plus spécifique, sinon ce dernier devient inatteignable.

- Exemple (branche inatteignable) :

```
return switch (o) {
    case Object obj -> "object";
    case String s -> "string"; // ❌ DOES NOT COMPILE: unreachable, String is already matched
};
```

- Autre exemple avec une garde :

```
return switch (o) {
    case Integer a -> "First";
    case Integer a when a > 0 -> "Second"; // ❌ DOES NOT COMPILE: unreachable, the first case
    // ...
};
```

Lorsqu'on utilise le pattern matching, les switch doivent être **exhaustifs** ; c'est-à-dire qu'ils doivent gérer toutes les valeurs possibles du selector.

Cela peut être réalisé en :

- Fournissant un case `default` qui gère toutes les valeurs non correspondantes aux autres cases.

- Fournissant une clause case finale avec un type de pattern qui correspond au type de référence du selector.
- Exemple (non exhaustif) :

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A"); // ❌ DOES NOT COMPILE: not exhaustive, selector
}
```

Pour corriger cela, vous pouvez :

- Changer le type de référence de `number` en `Short` (l'exhaustivité est alors satisfaite par le seul case).
- Ajouter une clause `default` qui couvre toutes les valeurs restantes.
- Ajouter une clause case finale couvrant le type de la variable selector, par exemple :

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A");
    case Number n -> System.out.println("B");
}
```

> **Warning**
>
> L'exemple suivant, qui utilise à la fois une clause `default` et une clause finale avec le même type que la variable selector, ne **compile pas** : le compilateur considère l'un des deux cases comme dominant toujours l'autre.

```
Number number = Short.valueOf(10);

switch (number) {
    case Short s -> System.out.println("A");
    case Number n -> System.out.println("B"); // ❌ DOES NOT COMPILE: dominated by either the
    default -> System.out.println("C");
}
```

7.3 Deux formes de `switch` : `switch` Statement vs `switch` Expression

7.3.1 L'instruction switch

Une **instruction switch** est utilisée comme construction de contrôle du flux.

Elle ne s'évalue pas, en elle-même, comme une valeur, bien que ses branches puissent contenir des instructions `return` qui retournent depuis la méthode englobante.

```
switch (mode) { // switch statement
    case "ON":
        start();
        break; // prevents fall-through
    case "OFF":
        stop();
        break;
    default:
        reset();
}
```

Points clés :

- Chaque clause `case` inclut une ou plusieurs valeurs correspondantes séparées par des virgules `,`. Un séparateur suit, qui peut être soit deux-points `:` soit, moins couramment pour les instructions, l'opérateur flèche `->`. Enfin, une expression ou un bloc (entouré de `{}`) définit le code à exécuter lorsqu'une correspondance se produit. Si vous utilisez l'opérateur flèche pour une clause, vous devez l'utiliser pour toutes les clauses de cette instruction switch.
- Le fall-through est possible pour les case de style deux-points à moins qu'une branche utilise `break`, `return` ou `throw`. Lorsqu'il est présent, `break` termine le switch après l'exécution de son case ; sans lui, l'exécution continue, dans l'ordre, vers les branches suivantes.
- Une clause `default` est optionnelle et peut apparaître n'importe où dans l'instruction switch. Elle s'exécute s'il n'y a pas de correspondance pour les cases précédents.
- Une instruction switch ne produit pas de valeur comme une expression ; vous ne pouvez pas assigner directement une instruction switch à une variable.

7.3.1.1 Comportement de fall-through

Avec des case de style deux-points, l'exécution saute à l'étiquette case correspondante.

S'il n'y a pas de `break`, elle continue dans le case suivant jusqu'à ce qu'un `break`, `return` ou `throw` soit rencontré.

```
int n = 2;

switch (n) {
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2"); // printed
    case 3:
        System.out.println("3"); // printed (fall-through)
        break;
    default:
        System.out.println("message default");
}
```

Sortie :

```
2
3
```

> **Note**
>
> Si, dans l'exemple précédent, nous supprimons le `break` sur le `case 3`, le message de la branche `default` sera également affiché.

7.3.2 L'expression switch

Une **expression switch** produit toujours une valeur unique comme résultat.

- Exemple :

```
int len = switch (s) { // switch expression
    case null -> 0;
    case "" -> 0;
    default -> s.length();
};
```

Points clés :

- Chaque clause `case` inclut une ou plusieurs valeurs correspondantes séparées par des virgules `,`, suivies de l'opérateur flèche `->`. Puis une expression ou un bloc (entouré de `{}`) définit le résultat pour cet arm.
- Lorsqu'elle est utilisée avec une assignation ou une instruction `return`, une expression switch nécessite un point-virgule de terminaison `;` après l'expression.
- Il n'y a pas de fall-through entre les arms avec flèche. Chaque arm correspondant est exécuté exactement une fois.

- Une expression switch doit être **exhaustive** : toutes les valeurs possibles du selector doivent être couvertes (via des case explicites et/ou `default`).
- Le type du résultat doit être cohérent entre toutes les branches. Par exemple, si un arm produit un `int`, les autres arms doivent produire des valeurs compatibles avec `int`.

7.3.2.1 `yield` dans les blocs d'expression switch

Lorsqu'un arm d'une expression switch utilise un bloc au lieu d'une expression unique, vous devez utiliser `yield` pour fournir le résultat de cet arm.

```
int len = switch (s) {
    case null -> 0;
    default -> {
        int l = s.trim().length();
        System.out.println("Length: " + l);
        yield l; // result of this arm
    }
};
```

> **Note**
>
> `yield` est utilisé uniquement dans les expressions switch. `break value;` n'est pas autorisé comme moyen de retourner une valeur depuis une expression switch.

7.3.2.2 Exhaustivité pour les expressions switch

Puisqu'une expression switch doit retourner une valeur, elle doit également être **exhaustive** ; en d'autres termes, elle doit gérer toutes les valeurs possibles du selector.

Vous pouvez garantir cela en :

- Fournissant un case `default`.
- Pour un selector enum : couvrant explicitement toutes les constantes enum.
- Pour des types sealed ou des pattern switch : couvrant tous les sous-types autorisés ou fournissant un `default`.

Exemple, exhaustif via `default` :

```
int val = switch (s) {
    case "one" -> 1;
    case "two" -> 2;
    default -> 0;
};
```

7.4 Gestion de null

Switch classique (sans patterns)

Si l'expression selector d'un switch classique (sans pattern matching) est évaluée à `null`, une `NullPointerException` est levée à l'exécution.

Pour éviter cela, vérifiez `null` avant d'effectuer le switch :

```
if (s == null) {
    // handle null
} else {
    switch (s) {
        case "A" -> ...
        default -> ...
    }
}
```

Pattern switch (avec `case null`)

Avec le pattern matching, vous pouvez gérer `null` directement à l'intérieur du switch :

```
int len = switch (s) {
    case null -> 0;
    default -> s.length();
};
```

> **Note**
>
> Pour les expressions switch :
>
> Si vous ne gérez pas `null` et que le selector est `null`, une `NullPointerException` est levée.
>
> L'utilisation de `case null` rend le switch explicitement sûr vis-à-vis de null.

> **Warning**
>
> Chaque fois que `case null` est utilisé dans un switch, le switch est traité comme un pattern switch, et toutes les règles applicables aux pattern switch (y compris l'exhaustivité et la dominance) s'appliquent.

◀ 6. Instanciation des types | ▲ Index | 8. Constructions de boucle en Java ▶

8. Constructions de boucle en Java

Table des matières

Java fournit plusieurs **constructions de boucle** qui permettent l'exécution répétée d'un bloc de code tant qu'une condition est vérifiée.

Les boucles sont essentielles pour l'itération, le parcours de structures de données, les calculs répétitifs et l'implémentation d'algorithmes.

8.1 La boucle `while`

La boucle `while` évalue sa **condition booléenne avant chaque itération**.
Si la condition est `false` dès le début, le corps n'est jamais exécuté.

Syntaxe

```
while (condition) {
    // loop body
}
```

- La condition doit être évaluée comme un booléen.
- La boucle peut s'exécuter zéro ou plusieurs fois.
- Les erreurs courantes incluent l'oubli de mettre à jour la variable de boucle, provoquant une boucle infinie.
- Exemple :

```
int i = 0;
while (i < 3) {
    System.out.println(i);
    i++;
}
```

Sortie :

```
0
1
2
```

8.2 La boucle `do-while`

La boucle `do-while` évalue sa condition **après** l'exécution du corps, garantissant que le corps s'exécute au moins une fois.

Syntaxe

```
do {
    // loop body
} while (condition);
```

> **Tip**
>
> `do-while` nécessite un point-virgule après la parenthèse fermante.

- Exemple :

```
int x = 5;
do {
    System.out.println(x);
    x--;
} while (x > 5); // body runs once even though condition is false
```

Sortie :

```
5
```

8.3 La boucle `for`

La boucle `for` traditionnelle convient le mieux aux boucles avec une variable compteur. Elle se compose de trois parties : initialisation, condition, mise à jour.

Syntaxe

```
for (initialization; condition; update) {
    // loop body
}
```

- L'initialisation s'exécute une fois avant le début de la boucle.
- La condition est évaluée avant chaque itération.
- La mise à jour s'exécute après chaque itération.
- L'initialisation et la mise à jour peuvent contenir plusieurs instructions séparées par des virgules.
- Les variables dans l'initialisation doivent toutes être du même type.
- Tout composant peut être omis, mais les points-virgules restent.
- Exemple :

```
for (int i = 0; i < 3; i++) {
    System.out.println(i);
}
```

Omission de parties :

```
int j = 0;
for (; j < 3;) {  // valid
    j++;
}
```

Instructions multiples :

```java
int x = 0;
for (long i = 0, c = 3; x < 3 && i < 12; x++, i++) {
    System.out.println(i);
}
```

8.4 La boucle `for-each` améliorée

Le `for` amélioré simplifie l'itération sur les tableaux et les collections.

Syntaxe

```java
for (ElementType var : arrayOrCollection) {
    // loop body
}
```

- La variable de boucle est en lecture seule par rapport à la collection sous-jacente.
- Fonctionne avec n'importe quel `Iterable` ou tableau.
- Ne peut pas supprimer des éléments sans itérateur.
- Exemple :

```java
String[] names = {"A", "B", "C"};
for (String n : names) {
    System.out.println(n);
}
```

Sortie :

```
A
B
C
```

8.5 Boucles imbriquées

Les boucles peuvent être imbriquées ; chacune conserve ses propres variables et conditions.

```java
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.println(i + "," + j);
    }
}
```

Sortie :

```
1,1
1,2
1,3
2,1
2,2
2,3
```

8.6 Boucles infinies

Une boucle est infinie lorsque sa condition est toujours évaluée à `true` ou est omise.

```java
while (true) { ... }
```

```java
for (;;) { ... }
```

> **Tip**
>
> Les boucles infinies doivent contenir `break`, `return` ou un contrôle externe.

8.7 `break` et `continue`

break

Quitte immédiatement la boucle la plus interne.

```
for (int i = 0; i < 5; i++) {
    if (i == 2) break;
    System.out.println(i);
}
```

continue

Saute le reste du corps de la boucle et passe à l'itération suivante.

```
for (int i = 0; i < 5; i++) {
    if (i % 2 == 0) continue;
    System.out.println(i);
}
```

> **Note**
>
> `break` et `continue` s'appliquent à la boucle la plus proche à moins que des étiquettes ne soient utilisées.

8.8 Boucles étiquetées

Une étiquette (identifiant + deux-points) peut être appliquée à une boucle pour permettre à break/continue d'affecter les boucles externes.

```
labelName:
for (...) {
    for (...) {
        break labelName;
    }
}
```

- Exemple :

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) break outer;
        System.out.println(i + "," + j);
    }
}
```

8.9 Portée des variables de boucle

- Les variables déclarées dans l'en-tête de la boucle sont limitées à la portée de cette boucle.
- Les variables déclarées à l'intérieur du corps existent uniquement à l'intérieur de ce bloc.

```java
for (int i = 0; i < 3; i++) {
    int x = i * 2;
}
// i and x are not accessible here
```

8.10 Code inatteignable après `break`, `continue` et `return`

Toute instruction placée **après** `break`, `continue` ou `return` dans le même bloc est considérée comme inatteignable et ne compile pas.

8.10.1 Code inatteignable après `break`

```java
for (int i = 0; i < 3; i++) {
    break;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

8.10.2 Code inatteignable après `continue`

```java
for (int i = 0; i < 3; i++) {
    continue;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

> **Note**
>
> `continue` saute à l'itération suivante, donc le code qui suit n'est jamais exécuté.

8.10.3 Code inatteignable après `return`

```java
int test() {
    return 5;
    System.out.println("Unreachable"); // ❌ Compile-time error
}
```

> **Note**
>
> `return` quitte la méthode immédiatement ; aucune instruction ne peut le suivre.

Core Standard APIs

9. Chaînes de caractères en Java

Table des matières

9.1 Chaînes & Text Blocks

9.1.1 Chaînes

9.1.1.1 Initialiser des chaînes

En Java, une **String** est un objet de la classe `java.lang.String`, utilisé pour représenter une séquence de caractères.

Les chaînes sont **immutables** : une fois créées, leur contenu ne peut pas être modifié. Toute opération qui semble modifier une chaîne en crée en réalité une nouvelle.

Vous pouvez créer et initialiser des chaînes de plusieurs façons :

```
String s1 = "Hello";                  // string literal
String s2 = new String("Hello");      // using constructor (not recommended)
String s3 = s1.toUpperCase();         // creates a new String ("HELLO")
```

> **Note**
>
> - Les littéraux de chaîne sont stockés dans le `pool de String`, une zone mémoire spéciale utilisée pour éviter de créer des objets chaîne en double.
> - L'utilisation du mot-clé `new` crée toujours un nouvel objet en dehors du pool.

9.1.1.2 Le String Pool

Comme les objets `String` sont immuables et largement utilisés, ils pourraient facilement occuper une grande quantité de mémoire dans un programme Java.

Pour réduire la duplication, Java réutilise toutes les chaînes déclarées comme littéraux (voir l'exemple ci-dessus), en les stockant dans une zone dédiée de la JVM appelée **String Pool** ou **Intern Pool**.

Veuillez consulter le paragraphe : **"6.4.3 String Pool and Equality"** dans le chapitre : Instanciation des types pour une explication et des exemples plus détaillés.

9.1.1.3 Caractères spéciaux et séquences d'échappement

Les chaînes peuvent contenir des caractères d'échappement, qui permettent d'inclure des symboles spéciaux ou des caractères de contrôle (caractères ayant une signification spéciale en Java).
Une séquence d'échappement commence par un backslash `\`.

> **Note**
>
> **Table des caractères spéciaux & séquences d'échappement dans les chaînes**

Escape	Signification	Exemple Java	Résultat
`\"`	guillemet double	`"She said \"Hi\""`	`She said "Hi"`
`\\`	backslash	`"C:\\Users\\Alex"`	`C:\Users\Alex`
`\n`	nouvelle ligne (LF)	`"Hello\nWorld"`	`Hello` + line break + `World`
`\r`	retour chariot (CR)	`"A\rB"`	`CR before B`
`\t`	tabulation	`"Name\tAge"`	`Name Age`
`\'`	guillemet simple	`"It\'s ok"`	`It's ok`
`\b`	retour arrière (backspace)	`"AB\bC"`	`AC` (le `B` est supprimé visuellement)
`\uXXXX`	unité de code Unicode	`"\u00A9"`	`©`

9.1.1.4 Règles de concaténation des chaînes

Comme introduit dans le chapitre sur Opérateurs Java, le symbole `+` représente normalement l'**addition arithmétique** lorsqu'il est utilisé avec des opérandes numériques.

Cependant, lorsqu'il est appliqué aux **String**, le même opérateur effectue la **concaténation de chaînes** — il crée une nouvelle chaîne en joignant les opérandes.

Comme l'opérateur `+` peut apparaître dans des expressions où des nombres et des chaînes sont présents, Java applique un ensemble spécifique de règles pour déterminer si `+` signifie **addition numérique** ou **concaténation de chaînes**.

9.1.1.5 Règles de concaténation

- Si les deux opérandes sont numériques, `+` effectue l'**addition numérique**.
- Si au moins un opérande est une `String`, l'opérateur `+` effectue la **concaténation de chaînes**.
- L'évaluation se fait strictement de gauche à droite, car `+` est **associatif à gauche**.

Cela signifie qu'une fois qu'une `String` apparaît sur le côté gauche de l'expression, toutes les opérations `+` suivantes deviennent des concaténations.

> **Tip**
>
> Comme l'évaluation se fait de gauche à droite, la position du premier opérande `String` détermine comment le reste de l'expression est évalué.

- Exemples

```
// *** Pure numeric addition

int a = 10 + 20;         // 30
double b = 1.5 + 2.3;    // 3.8



// *** String concatenation when at least one operand is a String

String s = "Hello" + " World";  // "Hello World"
String t = "Value: " + 10;      // "Value: 10"



// *** Left-to-right evaluation affects the result

System.out.println(1 + 2 + " apples");
// 3 + " apples"  → "3 apples"

System.out.println("apples: " + 1 + 2);
// "apples: 1" + 2 → "apples: 12"



// *** Adding parentheses changes the meaning

System.out.println("apples: " + (1 + 2));
// parentheses force numeric addition → "apples: 3"



// *** Mixed types with multiple operands

String result = 10 + 20 + "" + 30 + 40;
// (10 + 20) = 30
// 30 + ""  = "30"
// "30" + 30 = "3030"
String out = "3030" + 40; // "303040"

System.out.println(1 + 2 + "3" + 4 + 5);
// Step 1: 1 + 2 = 3
// Step 2: 3 + "3" = "33"
String r = "33" + 4;  // "334"
// Step 4: "334" + 5 = "3345"



// *** null is represented as a string when concatenated

System.out.println("AB" + null);
// ABnull
```

9.1.2 Text Blocks (depuis Java 15)

Un text block est un littéral de chaîne multi-ligne introduit pour simplifier l'écriture de grandes chaînes (comme du HTML, du JSON ou du code) sans avoir besoin de nombreuses séquences d'échappement.

Un text block commence et se termine par trois guillemets doubles (`"""`).

Vous pouvez utiliser les text blocks partout où vous utiliseriez des chaînes.

```
String html = """
    <html>
        <body>
            <p>Hello, world!</p>
        </body>
    </html>
    """;
```

> **Note**
>
> - Les text blocks incluent automatiquement les retours à la ligne et l'indentation pour la lisibilité. Les newlines sont normalisés en `\n`.
> - Les guillemets doubles à l'intérieur du bloc n'ont généralement pas besoin d'être échappés.
> - Le compilateur interprète le contenu entre les triples guillemets d'ouverture et de fermeture comme la valeur de la chaîne.

9.1.2.1 Mise en forme : espaces essentiels vs incidentels

- **Espaces essentiels** : espaces et newlines qui font partie du contenu de chaîne voulu.
- **Espaces incidentels** : indentation dans le code source que vous ne considérez pas conceptuellement comme faisant partie du texte.

```
String text = """
        Line 1
        Line 2
        Line 3
        """;
```

> **Important**
>
> - **Caractère le plus à gauche (baseline)** : la position du premier caractère non-espace sur l'ensemble des lignes (ou les `"""` de fermeture) définit la baseline d'indentation. Les espaces à gauche de cette baseline sont considérés comme incidentels et sont supprimés.
> - La ligne immédiatement après les `"""` d'ouverture n'est pas incluse dans la sortie si elle est vide (mise en forme typique).
> - Le newline avant les `"""` de fermeture est inclus dans le contenu.
>   Dans l'exemple ci-dessus, la chaîne résultante se termine par un newline après `"Line 3"` : il y a 4 lignes au total.

Sortie avec numéros de ligne (montrant la ligne vide finale) :

```
1: Line 1
2: Line 2
3: Line 3
4:
```

Pour supprimer le newline final :

- Utilisez un backslash de continuation de ligne à la fin de la dernière ligne de contenu.
- Placez les triples guillemets de fermeture sur la même ligne que le dernier contenu.

```
String textNoTrail_1 = """
        Line 1
        Line 2
        Line 3\
        """;

// OR

String textNoTrail_2 = """
        Line 1
        Line 2
        Line 3""";
```

9.1.2.2 Nombre de lignes, lignes vides et retours à la ligne

- Chaque retour à la ligne visible à l'intérieur du bloc devient `\n`.
- Les lignes vides à l'intérieur du bloc sont conservées.

```
String textNoTrail_0 = """
        Line 1
        Line 2 \n
        Line 3

        Line 4
        """;
```

Sortie :

```
1: Line 1
2: Line 2
3:
4: Line 3
5:
6: Line 4
7:
```

9.1.2.3 Text Blocks et caractères d'échappement

Les séquences d'échappement fonctionnent toujours à l'intérieur des text blocks lorsque nécessaire (par exemple, pour les backslashes ou des caractères de contrôle explicites).

```
String json = """
    {
      "name": "Alice",
      "path": "C:\\\\Users\\\\Alice"
    }\
    """;
```

Vous pouvez également formater un text block en utilisant des placeholders et `formatted()` :

```
String card = """
    Name: %s
    Age:  %d
    """.formatted("Alice", 30);
```

9.1.2.4 Erreurs courantes (avec corrections)

```
// ❌ Mismatched delimiters / missing closing triple quote
String bad = """
  Hello
World";      // ERROR — not a closing text block

// ✅ Fix
String ok = """
  Hello
  World
  """;
```

```
// ❌ Text blocks require a line break after the opening """
String invalid = """Hello""";  // ERROR

// ✅ Fix
String valid = """
    Hello
    """;
```

```
// ❌ Unescaped trailing backslash at end of a line inside the block
String wrong = """
    C:\Users\Alex\     // ERROR — backslash escapes the newline
    Documents
    """;

// ✅ Fix: escape backslashes, or avoid backslash at end of line
String correct = """
    C:\\Users\\Alex\\
    Documents\
    """;
```

9.2 Méthodes principales des chaînes

9.2.1 Indexation des chaînes

Les chaînes en Java utilisent une **indexation à base zéro**, ce qui signifie :

- Le premier caractère est à l'index `0`
- Le dernier caractère est à l'index `length() - 1`
- Accéder à un index en dehors de cette plage provoque une `StringIndexOutOfBoundsException`
- Exemple :

```
String s = "Java";
// Indexes:  0    1    2    3
// Chars:    J    a    v    a

char c = s.charAt(2); // 'v'
```

9.2.2 Méthode `length()`

`length()` renvoie le nombre de caractères dans la chaîne.

```
String s = "hello";
System.out.println(s.length());  // 5
```

Le dernier index valide est toujours `length() - 1`.

9.2.3 Règles de bornes : index de début vs index de fin

De nombreuses méthodes de `String` utilisent deux indices :

- **Index de début** — inclusif
- **Index de fin** — exclusif

Autrement dit, `substring(start, end)` inclut les caractères depuis l'index `start` jusqu'à (mais sans inclure) l'index `end`.

- L'index de début doit être `>= 0` et `<= length() - 1`
- L'index de fin peut être égal à `length()` (la "position virtuelle" après le dernier caractère).
- L'index de fin ne doit pas dépasser `length()`.
- L'index de début ne doit jamais être supérieur à l'index de fin.
- Exemple :

```
String s = "abcdef";
s.substring(1, 4); // "bcd" (indexes 1,2,3)
```

Cette règle s'applique à la plupart des méthodes basées sur substring.

9.2.4 Méthodes utilisant uniquement l'index de début (inclusif)

Méthode	Description	Paramètres	Règle d'index	Exemple
substring(int start)	Renvoie la sous-chaîne de start à la fin	start	start inclusif	"abcdef".substring(2) → "cdef"
indexOf(String)	Première occurrence	—	—	"Java".indexOf("a") → 1
indexOf(String, start)	Commence la recherche à l'index	start	start inclusif	"banana".indexOf("a", 2) → 3
lastIndexOf(String)	Dernière occurrence	—	—	"banana".lastIndexOf("a") → 5
lastIndexOf(String, fromIndex)	Recherche à rebours depuis l'index	fromIndex	fromIndex inclusif	"banana".lastIndexOf("a", 3) → 3

9.2.5 Méthodes avec début inclusif / fin exclusive

Ces méthodes suivent le même comportement de découpage : `start` inclus, `end` exclus.

Méthode	Description	Signature	Exemple
substring(start, end)	Extrait une partie de la chaîne	(int start, int end)	"abcdef".substring(1,4) → "bcd"
regionMatches	Compare des régions de sous-chaînes	(toffset, other, ooffset, len)	"Hello".regionMatches(1, "ell", 0, 3) → true
getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)	Copie des caractères dans un tableau de bytes	début inclusif, fin exclusive	Copie les caractères dans [srcBegin, srcEnd)
copyValueOf(char[] data, int offset, int count)	Crée une nouvelle chaîne	offset inclusif ; offset+count exclusif	Même règle que substring

9.2.6 Méthodes opérant sur toute la chaîne

Méthode	Description	Exemple
toUpperCase()	Version en majuscules	"java".toUpperCase() → "JAVA"
toLowerCase()	Version en minuscules	"JAVA".toLowerCase() → "java"
trim()	Supprime les espaces en début/fin	" hi ".trim() →"hi"
strip()	Trim compatible Unicode	" hi003".strip() → "hi"
stripLeading()	Supprime les espaces initiaux	" hi".stripLeading() → "hi"
stripTrailing()	Supprime les espaces finaux	"hi".stripTrailing() → "hi"
isBlank()	Vrai si vide ou uniquement des espaces	" ".isBlank() → true
isEmpty()	Vrai si length == 0	"".isEmpty() → true

9.2.7 Accès aux caractères

Méthode	Description	Exemple
charAt(int index)	Renvoie le caractère à l'index	"Java".charAt(2) → 'v'
codePointAt(int index)	Renvoie le point de code Unicode	Utile pour les emojis ou les caractères au-delà du BMP

9.2.8 Recherche

Méthode	Description	Exemple
contains(CharSequence)	Test de sous-chaîne	"hello".contains("ell") → true
startsWith(String)	Préfixe	"abcdef".startsWith("abc") → true
startsWith(String, offset)	Préfixe à l'index	"abc".startsWith("b", 1) → true
endsWith(String)	Suffixe	"abcdef".endsWith("def") → true

9.2.9 Méthodes de remplacement

Méthode	Description	Exemple
replace(char old, char new)	Remplace des caractères	"banana".replace('a','o') → "bonono"
replace(CharSequence old, CharSequence new)	Remplace des sous-chaînes	"ababa".replace("aba","X") → "Xba"
replaceAll(String regex, String replacement)	Remplacement regex global	"a1a2".replaceAll("\d","") →"aa"
replaceFirst(String regex, String replacement)	Seulement la première correspondance regex	"a1a2".replaceFirst("\d","") →"aa2"

9.2.10 Découpage et jonction

Méthode	Description	Exemple
split(String regex)	Découpe par regex	"a,b,c".split(",") → ["a","b","c"]
split(String regex, int limit)	Découpe avec limite	limit < 0 conserve toutes les chaînes vides finales

9.2.11 Méthodes retournant des tableaux

Méthode	Description	Exemple
toCharArray()	Renvoie char[]	"abc".toCharArray()
getBytes()	Renvoie byte[] en utilisant l'encodage plateforme/par défaut	"á".getBytes()

9.2.12 Indentation

Méthode	Description	Exemple
indent(int numSpaces)	Ajoute (positif) ou supprime (négatif) des espaces au début de chaque ligne ; ajoute aussi un retour à la ligne final s'il n'est pas déjà présent	str.indent(-20)
stripIndent()	Supprime tous les espaces initiaux incidentels de chaque ligne ; n'ajoute pas de retour à la ligne final	str.stripIndent()

- Exemple :

```
var txtBlock = """

                    a
                      b
                     c""";

var conc = " a\n" + " b\n" + " c";

System.out.println("length: " + txtBlock.length());
System.out.println(txtBlock);
System.out.println("");
String stripped1 = txtBlock.stripIndent();
System.out.println(stripped1);
System.out.println("length: " + stripped1.length());

System.out.println("**********");

System.out.println("length: " + conc.length());
System.out.println(conc);
System.out.println("");
String stripped2 = conc.stripIndent();
System.out.println(stripped2);
System.out.println("length: " + stripped2.length());
```

Sortie :

```
length: 9

a
  b
 c



a
  b
 c
length: 9
**********
length: 8
 a
 b
 c

a
b
c
length: 5
```

9.2.13 Exemples supplémentaires

- Exemple 1 — Extraire `[start, end)`

```java
String s = "012345";
System.out.println(s.substring(2, 5));
// includes 2,3,4 → prints "234"
```

- Exemple 2 — Recherche à partir d'un index de début

```java
String s = "hellohello";
int idx = s.indexOf("lo", 5); // search begins at index 5
```

- Exemple 3 — Pièges courants

```java
String s = "abcd";
System.out.println(s.substring(1,1)); // "" empty string
System.out.println(s.substring(3, 2)); // ❌ Exception: start index (3) > end index (2)

System.out.println("abcd".substring(2, 4)); // "cd" — includes indexes 2 and 3; 4 is excluded

System.out.println("abcd".substring(2, 5)); // ❌ StringIndexOutOfBoundsException (end index
```

◀ 8. Constructions de boucle en Java | ▲ Index | 10. Tableaux en Java ▶

10. Tableaux en Java

Table des matières

10.1 Ce qu'est un tableau

Les tableaux en Java sont des collections **à taille fixe**, **indexées**, **ordonnées** d'éléments du même type.

Ce sont des **objets**, même lorsque les éléments sont des primitifs.

10.1.1 Déclarer des tableaux

Vous pouvez déclarer un tableau de deux façons :

```
int[] a;      // preferred modern syntax
int b[];      // legal, older style
String[] names;
Person[] people;

// [] can be before or after the name: all the following declarations are equivalent.

int[] x;
int [] x1;
int []x2;
int x3[];
int x5 [];

// MULTIPLE ARRAY DECLARATIONS

int[] arr1, arr2;   // Declares two arrays of int

// WARNING:
// Here arr1 is an int[] and arr2 is just an int (NOT an array!)
int arr1[], arr2;
```

Déclarer ne crée PAS le tableau — cela crée seulement une variable capable d'en référencer un.

10.1.2 Créer des tableaux (instanciation)

Un tableau est créé en utilisant `new` suivi du type des éléments et de la longueur du tableau :

```
int[] numbers = new int[5];
String[] words = new String[3];
```

Règles clés - La longueur doit être non négative et spécifiée au moment de la création. - La longueur ne peut pas être modifiée ensuite. - La longueur du tableau peut être n'importe quelle expression `int`.

```
int size = 4;
double[] values = new double[size];
```

- Exemples illégaux de création de tableau :

```
// int length = -1;
// int[] arr = new int[-1];   // Runtime: NegativeArraySizeException

// int[] arr = new int[2.5];  // Compile error: size must be int
```

10.1.3 Valeurs par défaut dans les tableaux

Les tableaux (puisque ce sont des objets) reçoivent toujours une **initialisation par défaut** :

Type d'élément	Valeur par défaut
Numérique	0
boolean	false
char	'000'
Types référence	null

- Exemple :

```
int[] nums = new int[3];
System.out.println(nums[0]); // 0

String[] s = new String[3];
System.out.println(s[0]);    // null
```

10.1.4 Accéder aux éléments

On accède aux éléments en utilisant une indexation à base zéro :

```java
int[] a = new int[3];
a[0] = 10;
a[1] = 20;
System.out.println(a[1]); // 20
```

Exception courante

- `ArrayIndexOutOfBoundsException` (à l'exécution)

```java
// int[] x = new int[2];
// System.out.println(x[2]); // ❌ index 2 out of bounds
```

10.1.5 Raccourcis d'initialisation de tableaux

10.1.5.1 Création anonyme de tableau

```java
int[] a = new int[] {1,2,3};
```

10.1.5.2 Syntaxe courte (uniquement à la déclaration)

```java
int[] b = {1,2,3};
```

> La syntaxe courte `{1,2,3}` ne peut être utilisée qu'au moment de la déclaration.

```java
// int[] c;
// c = {1,2,3};  // ❌ does not compile
```

10.2 Tableaux multidimensionnels (tableaux de tableaux)

Java implémente les tableaux multi-dimensionnels comme des **tableaux de tableaux**.

Déclaration :

```java
int[][] matrix;
String[][][] cube;
```

10.2.1 Créer un tableau rectangulaire

```java
int[][] rect = new int[3][4]; // 3 rows, 4 columns each
```

10.2.2 Créer un tableau dentelé (irrégulier)

Vous pouvez créer des lignes de longueurs différentes :

```java
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[5];
jagged[2] = new int[1];
```

10.3 Longueur d'un tableau vs longueur d'une chaîne

- Les tableaux utilisent `.length` (champ `public final`).
- Les chaînes utilisent `.length()` (méthode).

> **Tip**
>
> C'est un piège classique: champs vs méthodes.

```
// int x = arr.length;   // OK
// int y = s.length;     // ❌ does not compile: missing ()
int yOk = s.length();
```

10.4 Affectations de références de tableaux

10.4.1 Affecter des références compatibles

```
int[] a = {1,2,3};
int[] b = a; // both now point to the same array
```

Modifier une référence affecte l'autre :

```
b[0] = 99;
System.out.println(a[0]); // 99
```

10.4.2 Affectations incompatibles (erreurs à la compilation)

```
// int[] x = new int[3];
// long[] y = x;     // ❌ incompatible types
```

Les références de tableaux suivent les règles normales d'héritage :

```
String[] s = new String[3];
Object[] o = s;      // OK: arrays are covariant
```

10.4.3 Danger d'exécution de la covariance : `ArrayStoreException`

```
Object[] objs = new String[3];
// objs[0] = Integer.valueOf(5); // ❌ ArrayStoreException at runtime
```

10.5 Comparer des tableaux

`==` compare les références (identité) :

```
int[] a = {1,2};
int[] b = {1,2};
System.out.println(a == b); // false
```

`equals()` sur les tableaux ne compare pas le contenu (il se comporte comme `==`) :

```
System.out.println(a.equals(b)); // false
```

Pour comparer le contenu, utilisez des méthodes de `java.util.Arrays` :

```
Arrays.equals(a, b);         // shallow comparison
Arrays.deepEquals(o1, o2);   // deep comparison for nested arrays
```

10.6 Méthodes utilitaires de `Arrays`

10.6.1 `Arrays.toString()`

```
System.out.println(Arrays.toString(new int[]{1,2,3})); // [1, 2, 3]
```

10.6.2 `Arrays.deepToString()` (pour les tableaux imbriqués)

```
System.out.println(Arrays.deepToString(new int[][] {{1,2},{3,4}}));
// [[1, 2], [3, 4]]
```

10.6.3 `Arrays.sort()`

```
int[] a = {4,1,3};
Arrays.sort(a); // [1, 3, 4]
```

> **Tip**
>
> - Les chaînes sont triées selon l'ordre naturel (lexicographique).
> - Les nombres sont triés avant les lettres, et les lettres majuscules sont triées avant les minuscules (nombres < majuscules < minuscules).
> - Pour les types référence, `null` est considéré plus petit que toute valeur non nulle.

```
String[] arr = {"AB", "ac", "Ba", "bA", "10", "99"};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));  // [10, 99, AB, Ba, ac, bA]
```

10.6.4 `Arrays.binarySearch()`

Exigences : le tableau doit être trié selon le même ordre ; sinon le résultat est imprévisible.

```
int[] a = {1,3,5,7};
int idx = Arrays.binarySearch(a, 5); // returns 2
```

Quand la valeur n'est pas trouvée, `binarySearch` renvoie `-(insertionPoint) - 1` :

```
int pos = Arrays.binarySearch(a, 4); // returns -3
// Insertion point is index 2 → -(2) - 1 = -3
```

10.6.5 `Arrays.compare()`

La classe `Arrays` propose un `equals()` surchargé qui vérifie si deux tableaux contiennent les mêmes éléments (et ont la même longueur) :

```
System.out.println(Arrays.equals(new int[] {200}, new int[] {100}));        // false
System.out.println(Arrays.equals(new int[] {200}, new int[] {200}));        // true
System.out.println(Arrays.equals(new int[] {200}, new int[] {100, 200}));   // false
```

Elle fournit aussi une méthode `compare()` avec ces règles :

- Si le résultat `n < 0` → le premier tableau est considéré "plus petit" que le second.
- Si le résultat `n > 0` → le premier tableau est considéré "plus grand" que le second.
- Si le résultat `n == 0` → les tableaux sont égaux.
- Exemples :

```java
int[] arr1 = new int[] {200, 300};
int[] arr2 = new int[] {200, 300, 400};
System.out.println(Arrays.compare(arr1, arr2));  // -1

int[] arr3 = new int[] {200, 300, 400};
int[] arr4 = new int[] {200, 300};
System.out.println(Arrays.compare(arr3, arr4));  // 1

String[] arr5 = new String[] {"200", "300", "aBB"};
String[] arr6 = new String[] {"200", "300", "ABB"};
System.out.println(Arrays.compare(arr5, arr6));     // Positive: "aBB" > "ABB"

String[] arr7 = new String[] {"200", "300", "ABB"};
String[] arr8 = new String[] {"200", "300", "aBB"};
System.out.println(Arrays.compare(arr7, arr8));     // Negative: "ABB" < "aBB"

String[] arr9 = null;
String[] arr10 = new String[] {"200", "300", "ABB"};
System.out.println(Arrays.compare(arr9, arr10));    // -1 (null considered smaller)
```

10.7 Boucle for améliorée avec les tableaux

```java
for (int value : new int[]{1,2,3}) {
    System.out.println(value);
}
```

Règles - Le côté droit doit être un tableau ou un `Iterable`. - Le type de la variable de boucle doit être compatible avec le type d'élément (pas d'élargissement de primitifs ici).

Erreur courante :

```java
// for (long v : new int[]{1,2}) {} // ❌ not allowed: int elements cannot be assigned to lon
```

10.8 Pièges courants

- **Accès hors limites** → lance `ArrayIndexOutOfBoundsException`.
- **Mauvaise utilisation de l'initialiseur court**

```java
// int[] x;
// x = {1,2}; // ❌ does not compile
```

- **Confondre `.length` et `.length()`**
- Oublier que les tableaux sont des objets (ils vivent sur le heap et sont référencés).
- **Mélanger des tableaux de primitifs et des tableaux de wrappers**

```java
// int[] p = new Integer[3]; // ❌ incompatible
```

- **Utiliser `binarySearch` sur des tableaux non triés** → résultats imprévisibles.
- **Exceptions d'exécution dues aux tableaux covariants** (`ArrayStoreException`).

10.9 Résumé

Les tableaux en Java sont :

- Des objets (même s'ils contiennent des primitifs).
- Des collections indexées à taille fixe.
- Toujours initialisés avec des valeurs par défaut.

- Type-safe, mais soumis aux règles de covariance (ce qui peut provoquer des exceptions à l'exécution si mal utilisé).

◄ 9. Chaînes de caractères en Java | ▲ Index | 11. Mathématiques en Java ►

11. Mathématiques en Java

Table des matières

11.1 API Math

La classe `java.lang.Math` fournit un ensemble de méthodes statiques utiles pour les opérations numériques.

Ces méthodes fonctionnent avec les types numériques primitifs.

Ci-dessous se trouve un résumé des plus fréquemment utilisées, ainsi que leurs formes surchargées.

11.1.1 Maximum et minimum entre deux valeurs

`Math.max()` et `Math.min()` comparent les deux valeurs fournies et renvoient le maximum ou le minimum entre elles.

Il existe quatre versions surchargées pour chaque méthode :

```
public static int min(int x, int y);
public static float min(float x, float y);
public static long min(long x, long y);
public static double min(double x, double y);

public static int max(int x, int y);
public static float max(float x, float y);
public static long max(long x, long y);
public static double max(double x, double y);
```

- Exemple :

```
System.out.println(Math.max(10.50, 7.5));   // 10.5
System.out.println(Math.min(10, -20));      // -20
```

11.1.2 `Math.round()`

`round()` renvoie l'entier le plus proche de son argument, en suivant les règles d'arrondi standard :
les valeurs dont la partie fractionnaire est 0.5 et au-dessus sont arrondies vers le haut ; en dessous de 0.5 elles sont arrondies vers le bas (vers l'entier le plus proche).

Surcharges

- `long round(double value)`
- `int round(float value)`
- Exemples :

```
Math.round(3.2);    // 3   (returns long)
Math.round(3.6);    // 4
Math.round(-3.5f);  // -3  (float version returns int)
```

> **Note**
>
> - La version `float` renvoie un `int`.
> - La version `double` renvoie un `long`.

11.1.3 `Math.ceil()` (Ceiling)

`ceil()` renvoie la plus petite valeur `double` qui est supérieure ou égale à l'argument.

Surcharge

- `double ceil(double value)`
- Exemples :

```
Math.ceil(3.1);   // 4.0
Math.ceil(-3.1);  // -3.0
```

11.1.4 `Math.floor()` (Floor)

`floor()` renvoie la plus grande valeur `double` qui est inférieure ou égale à l'argument.

Surcharge

- `double floor(double value)`
- Exemples :

```
Math.floor(3.9);   // 3.0
Math.floor(-3.1);  // -4.0
```

11.1.5 `Math.pow()`

`pow()` élève une valeur à une puissance.

Surcharge

- `double pow(double base, double exponent)`
- Exemples :

```
Math.pow(2, 3);      // 8.0
Math.pow(9, 0.5);    // 3.0  (square root)
Math.pow(10, -1);    // 0.1
```

11.1.6 `Math.random()`

`random()` renvoie un `double` dans l'intervalle `[0.0, 1.0)` (0.0 inclus, 1.0 exclus).

Surcharge

- `double random()`
- Exemples :

```
double r = Math.random();   // 0.0 <= r < 1.0

// Example: random int 0-9
int x = (int)(Math.random() * 10);
```

11.1.7 `Math.abs()`

`abs()` renvoie la valeur absolue (distance à zéro).

Surcharges

- `int abs(int value)`
- `long abs(long value)`
- `float abs(float value)`
- `double abs(double value)`

11.1.8 `Math.sqrt()`

`sqrt()` calcule la racine carrée et renvoie un `double`.

```
Math.sqrt(9);    // 3.0
Math.sqrt(-1);   // NaN (not a number)
```

11.1.9 Tableau récapitulatif

Méthode	Renvoie	Surcharges	Notes
`round()`	int ou long	float, double	Entier le plus proche
`ceil()`	double	double	Plus petite valeur >= argument
`floor()`	double	double	Plus grande valeur <= argument
`pow()`	double	double, double	Exponentiation
`random()`	double	none	0.0 <= r < 1.0
`min()/max()`	même type	int, long, float, double	Compare deux valeurs
`abs()`	même type	int, long, float, double	Valeur absolue
`sqrt()`	double	double	Racine carrée

11.2 BigInteger et BigDecimal

Les classes `BigInteger` et `BigDecimal` (dans `java.math`) fournissent des types numériques à précision arbitraire.

Elles sont utilisées lorsque :

- Les types primitifs (`int`, `long`, `double`, etc.) n'ont pas une plage suffisante.
- Les erreurs d'arrondi en virgule flottante de `float` / `double` ne sont pas acceptables (par exemple, dans les calculs financiers).

Les deux sont **immutables** : chaque opération renvoie une nouvelle instance.

11.2.1 Pourquoi `double` et `float` ne suffisent pas

Les types en virgule flottante (`float`, `double`) utilisent une représentation binaire. Beaucoup de fractions décimales ne peuvent pas être représentées exactement (comme 0.1 ou 0.2), ce qui produit des erreurs d'arrondi :

```
System.out.println(0.1 + 0.2); // 0.30000000000000004
```

Pour des tâches comme les calculs financiers, cela est inacceptable.

`BigDecimal` résout ce problème en représentant les nombres à l'aide d'un modèle décimal avec une échelle configurable (nombre de chiffres après la virgule).

11.2.2 BigInteger — Entiers à précision arbitraire

`BigInteger` représente des valeurs entières de taille pratiquement quelconque, limitée uniquement par la mémoire disponible.

11.2.3 Créer BigInteger

Méthodes courantes :

À partir d'un long

```
static BigInteger valueOf(long val);
```

À partir d'une String

```
BigInteger(String val);        // decimal by default
BigInteger(String val, int radix);
```

Valeur aléatoire

```
BigInteger(int numBits, Random rnd);
```

- Exemples :

```
import java.math.BigInteger;
import java.math.BigDecimal;
import java.util.Random;

BigInteger a = BigInteger.valueOf(10L);

// You can pass a long to both types, but a double only to BigDecimal

BigInteger g = BigInteger.valueOf(3000L);
BigDecimal p = BigDecimal.valueOf(3000L);
BigDecimal q = BigDecimal.valueOf(3000.00);

BigInteger b = new BigInteger("12345678901234567890"); // decimal string
BigInteger c = new BigInteger("FF", 16);               // 255 in base 16
BigInteger r = new BigInteger(128, new Random());      // random 128-bit number
```

11.2.4 Opérations (pas d'opérateurs !)

Vous ne pouvez pas utiliser les opérateurs arithmétiques standards (`+` , `-` , `*` , `/` , `%`) avec `BigInteger` ou `BigDecimal` .

À la place, vous devez appeler des méthodes (qui renvoient toutes de nouvelles instances). En voici quelques-unes courantes pour `BigInteger` :

- `add(BigInteger val)`
- `subtract(BigInteger val)`
- `multiply(BigInteger val)`
- `divide(BigInteger val)` – division entière
- `remainder(BigInteger val)`
- `pow(int exponent)`
- `negate()`
- `abs()`
- `gcd(BigInteger val)`
- `compareTo(BigInteger val)` – ordre
- Exemple :

```java
BigInteger x = new BigInteger("10000000000000000000");
BigInteger y = new BigInteger("3");

BigInteger sum = x.add(y);        // x + y
BigInteger prod = x.multiply(y);  // x * y
BigInteger div = x.divide(y);     // integer division
BigInteger rem = x.remainder(y);  // modulus

if (x.compareTo(y) > 0) {
    System.out.println("x is larger");
}
```

◀ 10. Tableaux en Java | ▲ Index | 12. Date et heure en Java ▶

12. Date et heure en Java

Table des matières

12.1 Date et heure

Java fournit une API moderne, cohérente et immuable de date/heure dans le package `java.time.*`.

Cette API remplace les anciennes classes `java.util.Date` et `java.util.Calendar`.

Selon le niveau de détail requis, Java propose quatre classes principales :

- `LocalDate` → représente une date uniquement (année–mois–jour)
- `LocalTime` → représente une heure uniquement (heure–minute–seconde–nanoseconde)
- `LocalDateTime` → combine date + heure, mais sans fuseau horaire
- `ZonedDateTime` → date + heure + décalage + fuseau horaire

> **Note**
>
> - Un **fuseau horaire** définit des règles comme les changements d'heure d'été (par exemple, `Europe/Paris`).
> - Un **décalage de zone** (zone offset) est un décalage fixe par rapport à UTC/GMT (par exemple, `+01:00`, `-07:00`).
> - Pour comparer deux instants provenant de fuseaux horaires différents, convertissez-les en UTC (GMT) en appliquant le décalage.

Obtenir la date/heure actuelle

Vous pouvez récupérer les valeurs système actuelles en utilisant les méthodes statiques `now()` :

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

- Exemple de sortie (votre système peut différer) :

```
2025-12-01
19:11:53.213856300
2025-12-01T19:11:53.213856300
2025-12-01T19:11:53.214856900+01:00[Europe/Paris]
```

- Exemple : conversion de `ZonedDateTime` vers GMT (UTC)

```
// Conceptual examples (not real code, just illustrating offsets):
// 2024-07-01T12:00+09:00[Asia/Tokyo]        ---> 12:00 minus 9 hours ---> 03:00 UTC
// 2024-07-01T20:00-07:00[America/Los_Angeles] ---> 20:00 plus 7 hours ---> 03:00 UTC
```

Les deux représentent le même instant dans le temps, simplement exprimé dans des fuseaux horaires différents.

12.1.1 Créer des dates et heures spécifiques

Vous pouvez construire des objets date/heure précis en utilisant les méthodes de fabrique `of()`.

Toutes les classes incluent plusieurs versions surchargées de `of()` (seules les plus courantes sont listées ici).

**LocalDate — formes surchargées de `of()`

- `of(int year, int month, int dayOfMonth)`
- `of(int year, Month month, int dayOfMonth)`

**LocalTime — formes surchargées de `of()`

- `of(int hour, int minute)`
- `of(int hour, int minute, int second)`
- `of(int hour, int minute, int second, int nanoOfSecond)`

**LocalDateTime — formes surchargées de `of()`

- `of(int year, int month, int day, int hour, int minute)`
- `of(int year, int month, int day, int hour, int minute, int second)`
- `of(int year, int month, int day, int hour, int minute, int second, int nano)`
- `of(LocalDate date, LocalTime time)`

**ZonedDateTime — formes surchargées de `of()`

- `of(LocalDate date, LocalTime time, ZoneId zone)`
- `of(int y, int m, int d, int h, int min, int s, int nano, ZoneId zone)`
- Exemples

```java
// Creating specific dates

var localDate1 = LocalDate.of(2025, 7, 31);
var localDate2 = LocalDate.of(2025, Month.JULY, 31);

// Creating specific times

var localTime1 = LocalTime.of(13, 21);
System.out.println(localTime1);                     // 13:21
System.out.println(LocalTime.of(13, 21, 52));       // 13:21:52
System.out.println(LocalTime.of(13, 21, 52, 200));  // 13:21:52.000000200

// Creating LocalDateTime

var localDateTime1 = LocalDateTime.of(2025, 7, 31, 13, 55, 22);
var localDateTime2 = LocalDateTime.of(localDate1, localTime1);

// Creating a ZonedDateTime

var zoned = ZonedDateTime.of(2025, 7, 31, 13, 55, 22, 0, ZoneId.of("Europe/Paris"));
```

12.1.2 Arithmétique date/heure : méthodes `plus` et `minus`

Toutes les classes du package `java.time` (comme `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, etc.) sont **immutables**.

Cela signifie que des méthodes comme `plusXxx()` et `minusXxx()` ne modifient jamais l'objet original — elles renvoient à la place une nouvelle instance avec la valeur ajustée.

12.1.3 Modèles courants

La plupart des classes date/heure prennent en charge trois types de méthodes arithmétiques :

- **Raccourcis spécifiques au type**
- `plusDays(long daysToAdd)`
- `plusHours(long hoursToAdd)`
- etc.
- **Méthodes génériques basées sur une quantité**
- `plus(TemporalAmount amount)` → par exemple `Period`, `Duration`
- `minus(TemporalAmount amount)`
- **Méthodes génériques basées sur une unité**
- `plus(long amountToAdd, TemporalUnit unit)`
- `minus(long amountToSubtract, TemporalUnit unit)`

Elles permettent une arithmétique de date/heure flexible et lisible.

12.1.4 Arithmétique `LocalDate`

`LocalDate` représente une date uniquement (pas d'heure, pas de zone).

Principales méthodes `plus` / `minus` (surcharges)

Méthode	Description
`plusDays(long days)`	Ajouter des jours
`plusWeeks(long weeks)`	Ajouter des semaines
`plusMonths(long months)`	Ajouter des mois
`plusYears(long years)`	Ajouter des années
`minusDays(long days)`	Soustraire des jours
`minusWeeks(long weeks)`	Soustraire des semaines
`minusMonths(long months)`	Soustraire des mois
`minusYears(long years)`	Soustraire des années
`plus(TemporalAmount amount)`	Ajouter une Period
`minus(TemporalAmount amount)`	Soustraire une Period
`plus(long amountToAdd, TemporalUnit unit)`	Ajouter en utilisant ChronoUnit (par ex. DAYS, MONTHS)
`minus(long amountToSubtract, TemporalUnit unit)`	Soustraire en utilisant ChronoUnit

- Exemples :

```
LocalDate date = LocalDate.of(2025, 3, 10);

LocalDate d1 = date.plusDays(5);            // 2025-03-15
LocalDate d2 = date.minusWeeks(2);          // 2025-02-24
LocalDate d3 = date.plusMonths(1);          // 2025-04-10
LocalDate d4 = date.plusYears(2);           // 2027-03-10

// Using ChronoUnit
LocalDate d5 = date.plus(10, ChronoUnit.DAYS);   // 2025-03-20

// Using Period
Period p = Period.of(1, 2, 3);  // 1 year, 2 months, 3 days
LocalDate d6 = date.plus(p);
```

12.1.5 Arithmétique `LocalTime`

`LocalTime` représente une heure uniquement (pas de date, pas de zone).

Principales méthodes `plus` / `minus` (surcharges)

Méthode	Description
`plusNanos(long nanos)`	Ajouter des nanosecondes
`plusSeconds(long seconds)`	Ajouter des secondes
`plusMinutes(long minutes)`	Ajouter des minutes
`plusHours(long hours)`	Ajouter des heures
`minusNanos(long nanos)`	Soustraire des nanosecondes
`minusSeconds(long seconds)`	Soustraire des secondes
`minusMinutes(long minutes)`	Soustraire des minutes
`minusHours(long hours)`	Soustraire des heures
`plus(TemporalAmount amount)`	Ajouter une Duration
`minus(TemporalAmount amount)`	Soustraire une Duration
`plus(long amountToAdd, TemporalUnit unit)`	Ajouter en utilisant ChronoUnit
`minus(long amountToSubtract, TemporalUnit unit)`	Soustraire en utilisant ChronoUnit

- Exemples

```
LocalTime time = LocalTime.of(13, 30);       // 13:30

LocalTime t1 = time.plusHours(2);            // 15:30
LocalTime t2 = time.minusMinutes(45);        // 12:45
LocalTime t3 = time.plusSeconds(90);         // 13:31:30

// Using ChronoUnit
LocalTime t4 = time.plus(3, ChronoUnit.HOURS);    // 16:30

// Using Duration
Duration d = Duration.ofMinutes(90);
LocalTime t5 = time.plus(d);                // 15:00
```

> **Note**
>
> Quand l'arithmétique des heures franchit minuit, la date est ignorée avec `LocalTime`. Par exemple, 23:30 + 2 heures = 01:30 (sans date impliquée).

12.1.6 Arithmétique `LocalDateTime`

`LocalDateTime` combine date + heure, mais toujours sans fuseau horaire.

Il prend en charge à la fois les méthodes de raccourci liées à la date et celles liées à l'heure.

Principales méthodes `plus` / `minus` (surcharges)

Méthode	Description
`plusYears(long years)` / `minusYears(long years)`	Ajuster les années
`plusMonths(long months)` / `minusMonths(long months)`	Ajuster les mois
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Ajuster les semaines
`plusDays(long days)` / `minusDays(long days)`	Ajuster les jours
`plusHours(long hours)` / `minusHours(long hours)`	Ajuster les heures
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Ajuster les minutes
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Ajuster les secondes
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Ajuster les nanosecondes
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Ajouter/soustraire Period ou Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	En utilisant ChronoUnit

- Exemples

```
LocalDateTime ldt = LocalDateTime.of(2025, 3, 10, 13, 30); // 2025-03-10T13:30

LocalDateTime l1 = ldt.plusDays(1);          // 2025-03-11T13:30
LocalDateTime l2 = ldt.minusHours(3);        // 2025-03-10T10:30
LocalDateTime l3 = ldt.plusMinutes(90);      // 2025-03-10T15:00

// Using ChronoUnit
LocalDateTime l4 = ldt.plus(2, ChronoUnit.WEEKS); // 2025-03-24T13:30

// Using Period and Duration
Period p = Period.ofDays(10);
Duration d = Duration.ofHours(5);

LocalDateTime l5 = ldt.plus(p);    // 2025-03-20T13:30
LocalDateTime l6 = ldt.plus(d);    // 2025-03-10T18:30
```

12.1.7 Arithmétique `ZonedDateTime`

`ZonedDateTime` représente date + heure + fuseau horaire + décalage.

Il prend en charge les mêmes méthodes `plus` / `minus` que `LocalDateTime`, mais avec une attention supplémentaire aux fuseaux horaires et à l'heure d'été (DST).

Principales méthodes `plus` / `minus` (surcharges)

Méthode	Description
`plusYears(long years)` / `minusYears(long years)`	Ajuster les années
`plusMonths(long months)` / `minusMonths(long months)`	Ajuster les mois
`plusWeeks(long weeks)` / `minusWeeks(long weeks)`	Ajuster les semaines
`plusDays(long days)` / `minusDays(long days)`	Ajuster les jours
`plusHours(long hours)` / `minusHours(long hours)`	Ajuster les heures
`plusMinutes(long minutes)` / `minusMinutes(long minutes)`	Ajuster les minutes
`plusSeconds(long seconds)` / `minusSeconds(long seconds)`	Ajuster les secondes
`plusNanos(long nanos)` / `minusNanos(long nanos)`	Ajuster les nanosecondes
`plus(TemporalAmount amount)` / `minus(TemporalAmount amount)`	Period / Duration
`plus(long amountToAdd, TemporalUnit unit)` / `minus(long amountToSubtract, TemporalUnit unit)`	En utilisant ChronoUnit

- Exemples (avec fuseaux horaires et DST) :

```
ZonedDateTime zdt = ZonedDateTime.of(
    2025, 3, 30, 1, 30, 0, 0,
    ZoneId.of("Europe/Paris")
);

// Add 2 hours across a possible DST change
ZonedDateTime z1 = zdt.plusHours(2);
System.out.println(zdt);
System.out.println(z1);
```

Selon les règles d'heure d'été pour cette date :

- L'heure locale peut passer de 02:00 à 03:00 ou similaire.
- `ZonedDateTime` ajuste le décalage et l'heure locale selon les règles de zone, mais représente toujours le bon instant sur la timeline.

> **Important**
>
> Pour `ZonedDateTime` , l'arithmétique est définie en fonction de la timeline locale et des règles de fuseau horaire, ce qui peut provoquer des décalages d'heures pendant les transitions DST.

12.1.8 Tableau récapitulatif

Classe	Méthodes plus/minus de raccourci	Méthodes génériques
LocalDate	plusDays, plusWeeks, plusMonths, plusYears (et minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalTime	plusNanos, plusSeconds, plusMinutes, plusHours (et minus)	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
LocalDateTime	Tous les raccourcis de LocalDate + LocalTime	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)
ZonedDateTime	Identiques à LocalDateTime, mais sensibles à la zone	plus/minus(TemporalAmount), plus/minus(long, TemporalUnit)

12.2 Méthodes `withXxx(...)`

Les méthodes `with...` renvoient une copie de l'objet avec un champ modifié.
Elles ne mutent jamais l'instance originale.

Classe	Méthodes with... courantes (non exhaustif)	Description
`LocalDate`	withYear(int year)	Même date, mais avec une année différente
LocalDate	LocalDate.withMonth(int month)	Même date, mois différent (1–12)
LocalDate	LocalDate.withDayOfMonth(int dayOfMonth)	Même date, jour du mois différent
LocalDate	LocalDate.with(TemporalField field, long newValue)	Ajustement générique basé sur un champ
LocalDate	LocalDate.with(TemporalAdjuster adjuster)	Utilise un adjuster (par ex. firstDayOfMonth())
`LocalTime`	withHour(int hour)	Même heure, heure différente
LocalTime	LocalTime.withMinute(int minute)	Même heure, minute différente
LocalTime	LocalTime.withSecond(int second)	Même heure, seconde différente
LocalTime	LocalTime.withNano(int nanoOfSecond)	Même heure, nanoseconde différente
LocalTime	LocalTime.with(TemporalField field, long newValue)	Ajustement générique basé sur un champ
LocalTime	LocalTime.with(TemporalAdjuster adjuster)	Ajuster via un temporal adjuster
`LocalDateTime`	withYear(int year), withMonth(int month), withDayOfMonth(int day)	Changer uniquement la partie date
LocalDateTime	withHour(int hour), withMinute(int minute), withSecond(int second)	Changer uniquement la partie heure
LocalDateTime	withNano(int nanoOfSecond)	Changer la nanoseconde
LocalDateTime	with(TemporalField field, long newValue)	Ajustement générique basé sur un champ
LocalDateTime	with(TemporalAdjuster adjuster)	Ajuster via un temporal adjuster
`ZonedDateTime`	tous les withXxx(...) de LocalDateTime	Changer les composants locaux date/heure
ZonedDateTime	withZoneSameInstant(ZoneId zone)	Même instant, zone différente (change l'heure locale)
ZonedDateTime	withZoneSameLocal(ZoneId zone)	Même date/heure locale, zone différente (change l'instant)

12.3 Conversion et méthodes `at...` (lier date, heure et zone)

Ces méthodes sont utilisées pour combiner ou convertir entre `LocalDate`, `LocalTime`, `LocalDateTime` et `ZonedDateTime`.

Depuis	Méthode	Résultat	Description
`LocalDate`	atTime(LocalTime time)	LocalDateTime	Combine cette date avec une heure donnée
LocalDate	atTime(int hour, int minute)	LocalDateTime	Surcharges de convenance avec des composantes horaires numériques
LocalDate	atTime(int hour, int minute, int second)	LocalDateTime	—
LocalDate	atTime(int hour, int minute, int second, int nano)	LocalDateTime	—
LocalDate	atStartOfDay()	LocalDateTime	Cette date à l'heure 00:00
LocalDate	atStartOfDay(ZoneId zone)	ZonedDateTime	Cette date au début de la journée dans une zone spécifique
`LocalTime`	atDate(LocalDate date)	LocalDateTime	Combine cette heure avec une date donnée
`LocalDateTime`	atZone(ZoneId zone)	ZonedDateTime	Ajoute un fuseau horaire à une date-heure locale
LocalDateTime	toLocalDate()	LocalDate	Extrait la composante date
LocalDateTime	toLocalTime()	LocalTime	Extrait la composante heure
`ZonedDateTime`	toLocalDate()	LocalDate	Supprime zone/décalage, conserve la date locale
ZonedDateTime	toLocalTime()	LocalTime	Supprime zone/décalage, conserve l'heure locale
ZonedDateTime	toLocalDateTime()	LocalDateTime	Supprime zone/décalage, conserve la date-heure locale

---

12.4 Period, Duration et Instant

Le package `java.time` fournit trois classes temporelles essentielles qui représentent des durées ou des points sur la timeline :

- **Period** → durées humaines basées sur la date (années, mois, jours)
- **Duration** → durées machine basées sur le temps (secondes, nanosecondes)
- **Instant** → un point sur la timeline UTC

12.5 `Period` — Durées humaines basées sur la date

`Period` représente une durée basée sur la date, telle que "3 ans, 2 mois et 5 jours".

Il est utilisé avec `LocalDate` et `LocalDateTime` (car ils contiennent des parties date).

Méthodes de création

Méthode	Description
Period.ofYears(int years)	Uniquement des années
Period.ofMonths(int months)	Uniquement des mois
Period.ofWeeks(int weeks)	Convertit les semaines en jours
Period.ofDays(int days)	Uniquement des jours
Period.of(int years, int months, int days)	Période complète
Period.parse(CharSequence text)	Format ISO-8601 : “P1Y2M3D”, “P7D”, “P1W”, …

Propriétés clés

- Ne prend pas en charge les heures, minutes, secondes, nanosecondes.
- Peut être négatif.
- Immuable.
- Exemples

```
Period p1 = Period.ofYears(1);              // P1Y
Period p2 = Period.of(1, 2, 3);             // P1Y2M3D
Period p3 = Period.ofWeeks(2);              // P14D (converted to days)

LocalDate base = LocalDate.of(2025, 1, 10);
LocalDate result = base.plus(p2);           // 2026-03-13
```

> **Note**
>
> `Period.parse("P1W")` est autorisé et représente une période de 7 jours (équivalente à “P7D”).

> **Tip**
>
> `Period` est basé sur le calendrier : ajouter une période de mois/années respecte la longueur des mois et les années bissextiles.

12.6 `Duration` — Durées machine basées sur le temps

`Duration` représente une durée basée sur le temps en secondes et nanosecondes.

Il est utilisé avec `LocalTime`, `LocalDateTime`, `ZonedDateTime` et `Instant`.

Méthodes de création

Méthode	Description
Duration.ofDays(long days)	Convertit les jours en secondes
Duration.ofHours(long hours)	Convertit les heures en secondes
Duration.ofMinutes(long minutes)	Convertit les minutes en secondes
Duration.ofSeconds(long seconds)	Représentation de base en secondes
Duration.ofSeconds(long seconds, long nanoAdjustment)	Secondes plus nanos supplémentaires
Duration.ofMillis(long millis)	Convertit les millisecondes en nanos
Duration.ofNanos(long nanos)	Uniquement des nanosecondes
Duration.between(Temporal start, Temporal end)	Calculer la durée entre deux instants
Duration.parse(CharSequence text)	ISO : “PT20H”, “PT15M”, “PT10S”

Caractéristiques clés

- Prend en charge des heures jusqu'aux nanosecondes, mais pas les années/mois/semaines directement.
- Idéal pour des calculs temporels au niveau machine.
- Immuable.
- Exemples

```
Duration d1 = Duration.ofHours(5);           // PT5H
Duration d2 = Duration.ofMinutes(90);        // PT1H30M

LocalTime t = LocalTime.of(10, 0);
LocalTime t2 = t.plus(d2);                   // 11:30

ZonedDateTime z1 = ZonedDateTime.of(
    2024, 3, 30, 1, 0, 0, 0,
    ZoneId.of("Europe/Paris")
);

ZonedDateTime z2 = z1.plusHours(2);          // DST-aware
ZonedDateTime z3 = z1.plus(d2);              // Duration-based
```

> **Note**
>
> `Duration.ofDays(1)` représente exactement 24 heures de temps machine. Dans une zone avec DST, 24 heures peuvent ne pas correspondre à "la même heure locale demain".

12.7 `Instant` — Point sur la chronologie UTC

`Instant` représente un seul moment dans le temps par rapport à UTC, avec une précision à la nanoseconde.

Il contient :

- Des secondes depuis l'époque (1970-01-01T00:00Z).
- Un ajustement de nanosecondes.

Méthodes de création

Méthode	Description
Instant.now()	Moment actuel en UTC
Instant.ofEpochSecond(long seconds)	À partir des secondes depuis l'époque
Instant.ofEpochSecond(long seconds, long nanos)	À partir des secondes plus nanos
Instant.ofEpochMilli(long millis)	À partir des millisecondes depuis l'époque
Instant.parse(CharSequence text)	ISO : "2024-01-01T10:15:30Z"

Conversions

Action	Méthode
Instant → heure zonée	instant.atZone(zoneId)
ZonedDateTime → Instant	zdt.toInstant()
LocalDateTime → Instant	Non autorisé directement (nécessite une zone)

- Exemple

```java
Instant i = Instant.now();

ZonedDateTime z = i.atZone(ZoneId.of("Europe/Paris"));
Instant back = z.toInstant();  // same moment

// Duration between instants
Instant start = Instant.parse("2024-01-01T10:00:00Z");
Instant end   = Instant.parse("2024-01-01T12:30:00Z");

Duration between = Duration.between(start, end); // PT2H30M
```

> **Important**
>
> `Instant` est toujours en UTC, sans information de fuseau horaire attachée. Il ne peut pas être combiné avec une `Period` ; utilisez `Duration` à la place.

12.8 Tableau récapitulatif (Period vs Duration vs Instant)

Concept	Représente	Bon pour	Fonctionne avec	Notes
Period	Années, mois, jours	Arithmétique calendaire	LocalDate, LocalDateTime	Unités humaines
Duration	Heures à nanosecondes	Calculs temporels précis	LocalTime, LocalDateTime, ZonedDateTime, Instant	Unités machine
Instant	Point exact sur la timeline UTC	Représentation d'horodatage	Convertible vers/depuis ZonedDateTime	Ne peut pas être combiné avec Period

Pièges courants

- `Period.of(1, 0, 0)` n'est pas la même chose que `Duration.ofDays(365)` (années bissextiles !).
- `Duration.ofDays(1)` peut ne pas être égal à une "journée calendaire" complète dans une zone DST.
- `LocalDateTime` ne peut pas être converti en `Instant` sans fuseau horaire.
- `Period.parse("P1W")` est valide et donne une période de 7 jours.

12.9 TemporalUnit et TemporalAmount

L'API `java.time` repose sur deux interfaces clés qui définissent comment les dates, les heures et les durées sont manipulées :

- `TemporalUnit` → représente une unité de temps (par exemple, DAYS, HOURS, MINUTES).
- `TemporalAmount` → représente une quantité de temps (par exemple, `Period`, `Duration`).

Les deux sont essentiels pour comprendre comment fonctionnent les méthodes `plus`, `minus` et `with`.

12.9.1 `TemporalUnit`

`TemporalUnit` représente une seule unité de mesure date/heure.
L'implémentation principale utilisée en Java est :

12.9.2 Enum `ChronoUnit`

Cet enum fournit les unités standard utilisées dans la chronologie ISO-8601 :

Catégorie	Unités
Unités de date	DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA, ERAS
Unités de temps	NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS
Spécial	FOREVER

Un `TemporalUnit` peut être utilisé directement avec les méthodes `plus()` et `minus()`.

- Exemples avec `ChronoUnit` :

```
LocalDate date = LocalDate.of(2025, 3, 10);

LocalDate d1 = date.plus(10, ChronoUnit.DAYS);      // 2025-03-20
LocalDate d2 = date.minus(2, ChronoUnit.MONTHS);    // 2025-01-10

LocalTime time = LocalTime.of(10, 0);
LocalTime t1 = time.plus(90, ChronoUnit.MINUTES);  // 11:30
```

> **Important**
>
> Vous ne pouvez pas utiliser des unités basées sur le temps avec `LocalDate`, ni des unités basées sur la date avec `LocalTime`.

- Exemples :

```
// ❌ UnsupportedTemporalTypeException
LocalDate d = LocalDate.now().plus(5, ChronoUnit.HOURS);

// ❌ UnsupportedTemporalTypeException
LocalTime t = LocalTime.now().plus(1, ChronoUnit.DAYS);
```

12.9.3 `TemporalAmount`

`TemporalAmount` représente une quantité de temps à unités multiples (par exemple, “2 ans, 3 mois”, ou “90 minutes”).
Elle est implémentée par :

- `Period` → années, mois, jours (basé sur la date)
- `Duration` → secondes, nanosecondes (basé sur le temps)

Les deux peuvent être passés aux objets date/heure pour les ajuster via `plus()` et `minus()`.

12.9.4 `Period` en tant que `TemporalAmount`

`Period` représente une quantité humaine : années, mois, jours.

- Exemples :

```
Period p = Period.of(1, 2, 3);  // 1 year, 2 months, 3 days

LocalDate base = LocalDate.of(2025, 3, 10);
LocalDate result = base.plus(p); // 2026-05-13
```

Notes

- `Period` ne peut pas être utilisé avec `LocalTime` (pas de composante date).
- `Period.ofWeeks(n)` est converti en interne en jours (n × 7).

12.9.5 `Duration` en tant que `TemporalAmount`

`Duration` représente un temps machine : secondes + nanosecondes.

- Exemples :

```
Duration d = Duration.ofHours(5).plusMinutes(30); // PT5H30M

LocalDateTime ldt = LocalDateTime.of(2025, 3, 10, 10, 0);
LocalDateTime result = ldt.plus(d); // 2025-03-10T15:30
```

Notes

- `Duration` peut être utilisée avec des classes qui ont des composantes de temps (`LocalTime` , `LocalDateTime` , `ZonedDateTime` , `Instant`).
- `Duration` ne peut pas être appliquée à `LocalDate` → cela lance `UnsupportedTemporalTypeException` .
- `Duration` interagit avec les zones et les transitions DST lorsqu'elle est appliquée à `ZonedDateTime` .

12.9.6 Utiliser `TemporalAmount` vs `TemporalUnit`

Utiliser un `TemporalUnit` :

```
LocalDate d1 = LocalDate.now().plus(5, ChronoUnit.DAYS);
```

Utiliser un `TemporalAmount` :

```
Period p = Period.ofDays(5);
LocalDate d2 = LocalDate.now().plus(p);
```

Les deux produisent le même résultat lorsque c'est pris en charge.

Différences

Aspect	TemporalUnit	TemporalAmount
Représente	Une seule unité (par ex. DAYS)	Une quantité structurée (par ex. 2Y, 5M, 3D)
Exemples	ChronoUnit.DAYS	Period.of(2,5,3)
Prend en charge plusieurs champs	Non	Oui
Bon pour	Incréments simples	Incréments complexes
Fréquent avec	Toutes les classes date/heure	Restreint selon le type

12.9.7 Méthodes `between(...)`

De nombreuses classes fournissent une méthode `between` via `ChronoUnit` , `Duration` ou `Period` .

Utiliser `Duration.between` (pour les classes basées sur le temps)

```
Duration d = Duration.between(
    LocalTime.of(10, 0),
    LocalTime.of(13, 30)
);
// PT3H30M
```

Utiliser `Period.between` (uniquement pour les dates)

```
Period p = Period.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 5, 10)
);
// P2M9D
```

Utiliser `ChronoUnit between`

```
long days = ChronoUnit.DAYS.between(
    LocalDate.of(2025, 3, 1),
    LocalDate.of(2025, 3, 10)
);
// 9
```

> **Important**
>
> `ChronoUnit.between(...)` renvoie toujours un `long`, tandis que `Period.between` renvoie une `Period`,
> et `Duration.between` renvoie une `Duration`.

12.9.8 Pièges courants

- Appliquer le mauvais `TemporalAmount` :

```
// LocalTime.plus(Period.ofDays(1))    // ❌ compile-time error
// LocalDate.plus(Duration.ofHours(1)) // ❌ runtime error: UnsupportedTemporalTypeException
```

- Changements DST avec `Duration` : ajouter 24 heures n'est pas toujours "demain" dans une zone avec DST.
- `Period.ofWeeks(1)` fait exactement 7 jours ; les effets DST apparaissent lorsqu'il est appliqué à des types sensibles à la zone.
- `Instant.plus(Period)` → `UnsupportedTemporalTypeException` à l'exécution ; utilisez `Duration` à la place.
- `Instant` ne peut pas être créé directement depuis un `LocalDateTime` ; vous devez d'abord appliquer une zone : `ldt.atZone(zone).toInstant()`.

12.9.9 Résumé

Fonctionnalité	TemporalUnit	TemporalAmount	ChronoUnit	Period	Duration
Représente	Une unité	Une quantité	enum d'unités	Y/M/J	S + nanos
Multi-champ	Non	Oui	Non	Oui	Non
Fonctionne avec	plus/minus	plus/minus	date/heure	LocalDate/LocalDateTime	Temps/zone
Basé sur l'humain	Non	Oui	Non	Oui	Non
Basé sur la machine	Oui	Oui	Oui	Non	Oui

◀ 11. Mathématiques en Java | ▲ Index | 13. Mise en forme et localisation en Java ▶

13. Mise en forme et localisation en Java

Table des matières

Ce chapitre fournit un traitement approfondi et pratique de la mise en forme en Java 21.

13.1 Mise en forme des chaînes

13.1.1 String.format et formatted

`String.format()` crée des chaînes formatées en utilisant des substituts de type printf.

Il est sensible au locale et retourne une nouvelle `String` immuable.

```
String result = String.format("The User: %s | Score: %d", "Bob", 42);
System.out.println(result);

// Or

System.out.println("The User: %s | Score: %d".formatted("Bob", 42));
```

Sortie :

```
The User: Bob | Score: 42
```

Caractéristiques clés :

- Utilise des spécificateurs de format tels que `%s` (tout type, généralement des chaînes), `%d` (valeurs entières), `%f` (valeurs à virgule flottante).
- Ne modifie pas les chaînes existantes.
- Lance `IllegalFormatException` si les arguments ne correspondent pas au format.
- Est sensible au locale lorsqu'un `Locale` est fourni.

```
String price = String.format(Locale.GERMANY, "%.2f", 1234.5);
// Sortie (locale allemand) : 1234,50
```

13.1.1.1 Indicateurs pour nombres à virgule flottante

`%f` est utilisé pour formater les nombres à virgule flottante (`float` , `double` , `BigDecimal`) en notation décimale.

```
System.out.printf("%f", 12.345);
```

```
12.345000
```

- **Affiche toujours 6 chiffres après le séparateur décimal** par défaut.
- Utilise l'arrondi (et non la troncature).
- Est sensible au locale pour le séparateur décimal.

13.1.1.2 Précision (.n)

La précision définit le nombre de chiffres affichés **après** le séparateur décimal.

```
System.out.printf("%.2f", 12.345);
```

```
12.35
```

- `%.0f` n'affiche aucun chiffre décimal.
- L'arrondi est appliqué.
- La précision est appliquée avant le remplissage de la largeur.

13.1.1.3 Largeur (m)

La largeur définit le nombre total minimal de caractères dans la sortie.

```
System.out.printf("%8.2f", 12.34);
```

```
   12.34
```

- **Remplit avec des espaces** par défaut.
- Si la valeur est plus longue, la largeur est ignorée (elle ne tronque jamais).
- Le remplissage est appliqué à gauche par défaut.

13.1.1.4 Indicateur de remplissage par zéro `0`

L'indicateur `0` remplace le remplissage par des espaces par des zéros.

```
System.out.printf("%08.2f", 12.34);
```

```
00012.34
```

- Nécessite une largeur.
- Les zéros sont insérés après le signe.
- Ignoré si la justification à gauche est présente (indicateur `-`).

13.1.1.5 Justification à gauche Indicateur `-`

L'indicateur `-` aligne la valeur à gauche à l'intérieur de la largeur.

```
System.out.printf("%-8.2f", 12.34);
```

```
12.34
```

- Le remplissage est déplacé vers la droite.
- Écrase le remplissage par zéro.

13.1.1.6 Signe explicite Indicateur `+`

L'indicateur `+` force l'affichage du signe pour les nombres positifs.

```
System.out.printf("%+8.2f", 12.34);
```

```
  +12.34
```

- Les nombres négatifs affichent déjà `-`.
- Écrase l'indicateur espace (qui affiche un espace en tête pour les valeurs positives).

13.1.1.7 Parenthèses pour les négatifs Indicateur `(`

L'indicateur `(` formate les nombres négatifs en utilisant des parenthèses.

```
System.out.printf("%(8.2f", -12.34);
```

```
 (12.34)
```

- N'affecte que les valeurs négatives.
- Rarement utilisé en pratique.

13.1.1.8 Combinaison des indicateurs

```
System.out.printf("%+010.2f", 12.34);
```

```
+000012.34
```

Ordre d'évaluation (simplifié) :

- La précision est appliquée.
- Le signe est traité.
- La largeur est appliquée.
- Le remplissage (espaces ou zéros) est appliqué.

13.1.1.9 Effets du Locale

```
System.out.printf(Locale.FRANCE, "%,.2f", 12345.67);
```

```
12 345,67
```

Les séparateurs décimaux et de groupement dépendent du `Locale` actif.

13.1.1.10 Erreurs courantes

- `%f` utilise 6 décimales par défaut si aucune précision n'est spécifiée.
- La largeur ne tronque jamais la sortie ; elle ne fait qu'ajouter du remplissage si nécessaire.
- L'indicateur `0` est ignoré lorsque `-` est présent.
- `+` écrase l'indicateur espace.
- Le groupement et les séparateurs dépendent du Locale.

13.1.2 Valeurs de texte personnalisées et échappement

Certains caractères ont une signification particulière dans les chaînes de format et doivent être échappés.

- `%%` → signe pourcentage littéral.
- `\n`, `\t` → échappements Java standards.

```
String msg = String.format("Completion: %d%%%nStatus: OK", 100);
System.out.println(msg);
```

Sortie :

```
Completion: 100%
Status: OK
```

> **Note**
>
> Un seul % sans spécificateur valide provoque une IllegalFormatException à l'exécution.

13.2 Mise en forme des nombres

13.2.1 NumberFormat

`NumberFormat` est une classe abstraite utilisée pour formater et analyser les nombres de manière sensible au locale.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(nf.format(1234567.89));
```

> **Important**
>
> - Les méthodes factory déterminent le style de formatage (général, entier, monnaie, pourcentage, compact, ...).
> - Le formatage dépend du `Locale` fourni.
> - `NumberFormat` (et `DecimalFormat`) ne sont pas thread-safe.

13.2.2 Localisation des nombres

La localisation des nombres affecte les séparateurs décimaux, les séparateurs de groupement et les symboles monétaires.

```
NumberFormat nfUS = NumberFormat.getInstance(Locale.US);
NumberFormat nfIT = NumberFormat.getInstance(Locale.ITALY);

System.out.println(nfUS.format(1234.56)); // 1,234.56
System.out.println(nfIT.format(1234.56)); // 1.234,56
```

13.2.3 DecimalFormat et NumberFormat

`DecimalFormat` est une sous-classe concrète de `NumberFormat` qui offre un contrôle fin de la mise en forme numérique à l'aide de patterns.

`NumberFormat` définit un formatage sensible au locale via des méthodes factory, tandis que `DecimalFormat` permet un contrôle explicite basé sur des patterns.

```
NumberFormat nf = NumberFormat.getInstance(Locale.US);
DecimalFormat df = (DecimalFormat) nf;
```

Ou directement :

```
DecimalFormat df = new DecimalFormat("#,##0.00");
```

> **Note**
>
> - `DecimalFormat` est mutable (on peut modifier le pattern, les symboles, etc.).
> - `DecimalFormat` n'est pas thread-safe.
> - Le formatage est sensible au locale via `DecimalFormatSymbols` .

13.2.4 Structure du pattern DecimalFormat

Un pattern peut contenir une sous-structure positive et une sous-structure négative optionnelle, séparées par `;` .

```
#,##0.00;(#,##0.00)
```

> **Note**
>
> - Première partie → nombres positifs.
> - Seconde partie → nombres négatifs.
> - Si la partie négative est omise, les nombres négatifs utilisent automatiquement un `-` en tête.

13.2.5 Le symbole `0` (chiffre obligatoire)

Le symbole `0` force l'affichage d'un chiffre, en complétant avec des zéros si nécessaire.

```
DecimalFormat df = new DecimalFormat("0000.00");
System.out.println(df.format(12.3));
```

```
0012.30
```

- Contrôle le nombre minimal de chiffres.
- Complète avec des zéros si le nombre contient moins de chiffres.
- Utile pour des sorties à largeur fixe ou alignées.

13.2.6 Le symbole `#` (chiffre optionnel)

Le symbole `#` affiche un chiffre uniquement s'il existe.

```
DecimalFormat df = new DecimalFormat("####.##");
System.out.println(df.format(12.3));
```

```
12.3
```

- Supprime les zéros initiaux.
- Supprime les zéros finaux inutiles.
- Adapté à un formatage « convivial ».

13.2.7 Combiner `0` et `#`

Les patterns combinent souvent les deux symboles pour plus de flexibilité.

```
DecimalFormat df = new DecimalFormat("#,##0.##");
System.out.println(df.format(12));
System.out.println(df.format(12.5));
System.out.println(df.format(12345.678));
```

```
12
12.5
12,345.68
```

Explication du pattern :

```
#,##0 . ##
 ^  ^    ^
 |  |    |
 |  |    └─ chiffres fractionnaires optionnels (#)
 |  └────── chiffre entier obligatoire (0)
 └────────── pattern de groupement (,)
```

- Au moins un chiffre entier est garanti (le `0`).
- Les chiffres sont regroupés par milliers à l'aide du séparateur de groupement.
- Les chiffres fractionnaires sont optionnels (jusqu'à deux).

13.2.8 Séparateurs décimaux et de groupement

Dans les patterns :

- `.` → séparateur décimal.
- `,` → séparateur de groupement.

Les symboles effectivement utilisés à l'exécution dépendent du `Locale` (par exemple, virgule vs point).

13.2.9 DecimalFormatSymbols : symboles de format spécifiques au Locale

```
DecimalFormatSymbols symbols =
        DecimalFormatSymbols.getInstance(Locale.FRANCE);

DecimalFormat df =
        new DecimalFormat("#,##0.00", symbols);

System.out.println(df.format(1234.5));
```

```
1 234,50
```

- Contrôle les séparateurs décimaux et de groupement.
- Contrôle le signe moins et le symbole monétaire.
- Contrôle les chaînes NaN et Infinity.

13.2.10 Patterns spéciaux de DecimalFormat

```
0.###E0   notation scientifique
###%      pourcentage
¤#,##0.00 monnaie (¤ est le symbole monétaire)
```

13.2.11 Règles et erreurs courantes

- `DecimalFormat` est une sous-classe de `NumberFormat`.
- `0` force les chiffres, `#` non.
- Les patterns contrôlent le formatage, pas le mode d'arrondi (utiliser `setRoundingMode()`).
- Le groupement fonctionne uniquement si le séparateur (généralement `,`) est présent dans le pattern.
- Le parsing peut réussir partiellement sans erreur si des caractères finaux suivent un nombre valide.
- `DecimalFormat` est mutable et non thread-safe.

13.3 Analyse (parsing) des nombres

L'analyse convertit un texte localisé en valeurs numériques. Par défaut, l'analyse est permissive.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRANCE);
Number n = nf.parse("12 345,67abc"); // analyse 12345.67
```

- L'analyse s'arrête au premier caractère invalide.
- Le texte final est ignoré sauf vérification explicite.

13.3.1 Parsing avec DecimalFormat

`DecimalFormat` peut également analyser des nombres. L'analyse est permissive par défaut.

```
DecimalFormat df = new DecimalFormat("#,##0.##");
Number n = df.parse("1,234.56abc");
```

- L'analyse s'arrête au premier caractère invalide.
- Le texte final est ignoré s'il est présent.

Pour imposer une analyse stricte :

```
df.setParseStrict(true);
```

13.3.2 CompactNumberFormat

La mise en forme compacte abrège les grands nombres pour une meilleure lisibilité humaine.

- Prend en charge les styles SHORT et LONG.
- Utilise des abréviations dépendantes du locale (par exemple K, M, « million »).

```
NumberFormat cnf =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.SHORT);

System.out.println(cnf.format(1_200));        // 1.2K
System.out.println(cnf.format(5_000_000));    // 5M

NumberFormat cnf1 =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.SHORT);

NumberFormat cnf2 =
        NumberFormat.getCompactNumberInstance(
                Locale.US, NumberFormat.Style.LONG);

System.out.println(cnf1.format(315_000_000));   // 315M
System.out.println(cnf2.format(315_000_000));   // 315 million
```

13.4 Mise en forme des dates et heures

13.4.1 DateTimeFormatter

Java 21 repose sur `java.time` et `DateTimeFormatter` pour la mise en forme moderne des dates et heures.

```
DateTimeFormatter f =
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
System.out.println(LocalDateTime.now().format(f));
```

Propriétés principales :

- Immuable.
- Thread-safe.
- Sensible au locale.

13.4.2 Symboles standard de date et d'heure

```
y   année
M   numéro du mois (ou nom avec plus de lettres)
d   jour du mois
E   nom du jour
H   heure (0-23)
h   heure (1-12)
m   minute
s   seconde
a   indicateur AM/PM
z   fuseau horaire
```

13.4.3 datetime.format vs formatter.format

Les deux méthodes sont fonctionnellement identiques :

```
date.format(formatter);
formatter.format(date);
```

- `date.format(formatter)` → préféré pour la lisibilité (données d'abord, puis formatage).
- `formatter.format(date)` → parfois pratique dans un code fonctionnel ou avec des formatters réutilisables.

13.4.4 Localisation des dates

Les styles localisés adaptent l'affichage des dates aux normes culturelles.

```
DateTimeFormatter fullIt =
        DateTimeFormatter
                .ofLocalizedDate(FormatStyle.FULL)
                .withLocale(Locale.ITALY);

DateTimeFormatter shortIt =
        DateTimeFormatter
                .ofLocalizedDate(FormatStyle.SHORT)
                .withLocale(Locale.ITALY);

LocalDate today = LocalDate.of(2025, 12, 17);

System.out.println(today.format(fullIt));
System.out.println(today.format(shortIt));
```

Sortie possible :

```
mercoledì 17 dicembre 2025
17/12/25
```

13.5 Internationalisation (i18n) et localisation (l10n)

13.5.1 Locales

Un `Locale` définit la langue, le pays et une variante optionnelle.

```
Locale l1 = Locale.US;
Locale l2 = Locale.of("fr", "FR");
Locale l3 = new Locale.Builder()
        .setLanguage("en")
        .setRegion("US")
        .build();
```

Formats de Locale :

- `en` (it, fr, etc.) : code langue en minuscules.
- `en_US` (fr_CA, it_IT, etc.) : code langue en minuscules + underscore + code pays en majuscules.

13.5.2 Catégories de Locale

Les catégories de Locale séparent la mise en forme de la langue de l'interface utilisateur.

`Locale.Category` permet à Java d'utiliser différents Locale par défaut selon les usages.

Il existe deux catégories :

Catégorie	Utilisée pour
FORMAT	Nombres, dates, monnaie, autre formatage
DISPLAY	Texte lisible (UI, noms, messages)

13.5.3 Exemple réel

Un utilisateur français vivant en Allemagne peut souhaiter :

- Nombres et dates → format allemand.
- Langue de l'interface → français.

Avant Java 7, cela n'était pas possible.

```
Locale.setDefault(Locale.Category.FORMAT, Locale.GERMANY);
Locale.setDefault(Locale.Category.DISPLAY, Locale.FRANCE);
```

Exemples d'effets :

Aspect	Résultat (exemple)
Nombres	1.234,56
Dates	31.12.2025
Monnaie	€
Texte UI	Français
Noms des mois	décembre
Noms des pays	Allemagne

13.6 Properties et Resource Bundles

Les resource bundles externalisent le texte et permettent la localisation sans modifier le code.

```
ResourceBundle rb =
        ResourceBundle.getBundle("messages", Locale.GERMAN);

String msg = rb.getString("welcome");
```

13.6.1 Règles de résolution des Resource Bundles

Java recherche les bundles selon un ordre de repli strict. Par exemple, avec le nom de base `messages` et le locale `de_DE` :

- messages_de_DE.properties
- messages_de.properties
- messages.properties

Si aucun n'est trouvé → `MissingResourceException`.

> **Note**
>
> Les fichiers `.properties` traditionnels sont spécifiés en ISO-8859-1 ; les caractères non ASCII doivent être encodés via des échappements Unicode (par exemple `\u00E9` pour é), sauf si des mécanismes de chargement alternatifs sont utilisés.

13.7 Règles et erreurs courantes

- `DateTimeFormatter` est immuable et thread-safe.
- `NumberFormat` / `DecimalFormat` sont mutables et non thread-safe.
- Modifier le `Locale` affecte la manière dont les valeurs sont formatées et analysées, et non les valeurs numériques ou temporelles sous-jacentes.
- L'analyse avec `NumberFormat` ou `DecimalFormat` peut réussir partiellement sans exception si du texte supplémentaire suit un nombre valide.
- `java.time` remplace la plupart des usages des anciennes API `java.util.Date` / `Calendar` dans le code moderne et à l'examen.

Object-Oriented Fundamentals

14. Méthodes, Attributs et Variables

Table des matières

Ce chapitre introduit des concepts fondamentaux de la Programmation Orientée Objet (OOP) en Java, en commençant par les **méthodes**, le **passage des paramètres**, la **surcharge**, les **variables locales**

vs. d'instance et les **varargs**.

14.1 Méthodes

Les `méthodes` représentent les **opérations/comportements** qui peuvent être effectués par un type de données particulier (une **classe**).

Elles décrivent *ce que l'objet peut faire* et comment il interagit avec son état interne et le monde extérieur.

Une `déclaration de méthode` est composée de composants **obligatoires** et **optionnels**.

14.1.1 Composants obligatoires d'une méthode

14.1.1.1 Modificateurs d'accès

Les `modificateurs d'accès` contrôlent la *visibilité*, pas le comportement. (Se référer au paragraphe "**Access Modifiers**" dans le chapitre : 2. Basic Language Java Building Blocks)

14.1.1.2 Type de retour

Apparaît **immédiatement avant** le nom de la méthode.

- Si la méthode retourne une valeur → le type de retour spécifie le type de la valeur.
- Si la méthode ne retourne *pas* de valeur → le mot-clé `void` **doit** être utilisé.
- Une méthode avec un type de retour non void **doit** contenir au moins une instruction `return valeur;`.
- Une méthode `void` peut :
- - omettre une instruction return
- - inclure `return;` (sans **aucune** valeur)

14.1.1.3 Nom de la méthode

Suit les mêmes règles que les identificateurs (Se référer au chapitre : 3. Java Naming Rules).

14.1.1.4 Signature de la méthode

La **signature de la méthode** en Java inclut :

- le *nom de la méthode*
- la *liste des types de paramètres* (types + ordre)

> **Note**
>
> Les **noms des paramètres NE font PAS partie de la signature**, seuls les types et l'ordre comptent.

- Exemple de signatures distinctes :

```
void process(int x)
void process(int x, int y)
void process(int x, String y)
```

- Exemple de *même* signature (surcharge illégale) :

```
// ❌ même signature : seuls les noms des paramètres diffèrent
void m(int a)
void m(int b)
```

14.1.1.5 Corps de la méthode

Un bloc `{ }` contenant **zéro ou plusieurs instructions**. Si la méthode est `abstract`, le corps doit être omis.

14.1.2 Modificateurs optionnels

Les modificateurs optionnels de méthode incluent :

- `static`
- `abstract`
- `final`
- `default` (méthodes d'interface)
- `synchronized`
- `native`
- `strictfp`

Règles :

- Les modificateurs optionnels peuvent apparaître dans **n'importe quel ordre**.
- Tous les modificateurs doivent apparaître **avant le type de retour**.
- Exemple :

```
public static final int compute() {
    return 10;
}
```

14.1.3 Déclarer des méthodes

```
public final synchronized String formatValue(int x, double y) throws IOException {
    return "Result: " + x + ", " + y;
}
```

Décomposition :

Partie	Signification
public	modificateur d'accès
final	ne peut pas être redéfinie
synchronized	modificateur de contrôle des threads
String	type de retour
formatValue	nom de la méthode
(int x, double y)	liste des paramètres
throws IOException	liste des exceptions
method body	implémentation

14.2 Java est un langage « Pass-by-Value »

Java utilise **uniquement le pass-by-value**, sans exception.

Cela signifie :

- Pour les **types primitifs** → la méthode reçoit une *copie de la valeur*.
- Pour les **types référence** → la méthode reçoit une *copie de la référence*, ce qui signifie :
- - la référence elle-même ne peut pas être modifiée par la méthode
- - l'*objet* **peut** être modifié via cette référence
- Exemple :

```
void modify(int a, StringBuilder b) {
    a = 50;           // modification de la *copie* → aucun effet à l'extérieur
    b.append("!");    // modification de l'*objet* → visible à l'extérieur
}
```

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

    int num1 = 11;
    methodTryModif(num1);
    System.out.println(num1);

}

public static void methodTryModif(int num1){
    num1 = 10;  // cette nouvelle affectation concerne uniquement le paramètre du méthode qui,
}
```

14.3 Surcharge des méthodes

La surcharge des méthodes signifie **même nom de méthode**, **signature différente**.

Deux méthodes sont considérées comme surchargées si elles diffèrent par :

- le nombre de paramètres
- les types des paramètres
- l'ordre des paramètres

La surcharge **NE dépend PAS de** :

- le type de retour
- le modificateur d'accès
- les exceptions
- Exemple :

```
void print(int x)
void print(double x)
void print(int x, int y)
```

Méthode surchargée illégale :

```
// ❌ Le type de retour ne compte pas dans la surcharge
int compute(int x)
double compute(int x)
```

14.3.1 Appeler des méthodes surchargées

Lorsque plusieurs méthodes surchargées sont disponibles, Java applique la **résolution de surcharge** pour décider quelle méthode appeler.

Le compilateur sélectionne la méthode dont les types de paramètres sont les **plus spécifiques** et **compatibles** avec les arguments fournis.

La résolution de surcharge a lieu **au moment de la compilation** (contrairement à l'overriding, qui est basé sur l'exécution).

Java suit ces règles :

14.3.1.1 La correspondance exacte l'emporte

Si un argument correspond exactement à un paramètre de méthode, cette méthode est choisie.

```
void call(int x)    { System.out.println("int"); }
void call(long x)   { System.out.println("long"); }

call(5); // affiche : int (correspondance exacte pour int)
```

14.3.1.2 — S'il n'existe pas de correspondance exacte, Java choisit le type compatible *le plus spécifique*

Java préfère :

- **l'élargissement** plutôt que l'autoboxing
- **l'autoboxing** plutôt que les varargs
- **le type de référence plus large** uniquement si un type plus spécifique n'est pas disponible
- Exemple avec des primitifs numériques :

```
void test(long x)   { System.out.println("long"); }
void test(float x)  { System.out.println("float"); }

test(5);  // littéral int : peut être élargi en long ou float
          // mais long est plus spécifique que float pour les types entiers
          // Output : long
```

14.3.1.3 — L'élargissement des primitifs bat le boxing

Si un argument primitif peut être soit élargi soit autoboxé, Java choisit l'élargissement.

```
void m(int x)       { System.out.println("int"); }
void m(Integer x)   { System.out.println("Integer"); }

byte b = 10;
m(b);               // byte → int (élargissement) l'emporte
                    // Output : int
```

14.3.1.4 — Le boxing bat les varargs

```
void show(Integer x)    { System.out.println("Integer"); }
void show(int... x)     { System.out.println("varargs"); }

show(5);                // int → Integer (boxing) préféré
                        // Output : Integer
```

14.3.1.5 — Pour les références, Java choisit le type de référence le plus spécifique

```
void ref(Object o)      { System.out.println("Object"); }
void ref(String s)      { System.out.println("String"); }

ref("abc");             // "abc" est une String → plus spécifique que Object
                        // Output : String
```

Plus spécifique signifie *plus bas dans la hiérarchie d'héritage*.

14.3.1.6 — Lorsqu'il n'y a pas de « plus spécifique » non ambigu, l'appel est une erreur de compilation

Exemple avec des classes sœurs :

```
void check(Number n)      { System.out.println("Number"); }
void check(String s)      { System.out.println("String"); }

check(null);    // String et Number acceptent null
                // String est plus spécifique car c'est une classe concrète
                // Output : String
```

Mais si deux classes non liées entrent en concurrence :

```
void run(String s)   { }
void run(Integer i)  { }

run(null);  // ❌ Erreur à la compilation : appel de méthode ambigu
```

14.3.1.7 — Surcharges mixtes primitifs + wrappers

Java évalue l'élargissement, le boxing et les varargs dans cet ordre :

`élargissement → boxing → varargs`

- Exemple :

```
void mix(long x)         { System.out.println("long"); }
void mix(Integer x)      { System.out.println("Integer"); }
void mix(int... x)       { System.out.println("varargs"); }

short s = 5;
mix(s);   // short → int → long  (élargissement)
          // boxing et varargs ignorés
          // Output : long
```

14.3.1.8 — Quand les primitifs se mélangent avec les types de référence

```
void fun(Object o)     { System.out.println("Object"); }
void fun(int x)        { System.out.println("int"); }

fun(10);                // la correspondance primitive exacte l'emporte
                        // Output : int

Integer i = 10;
fun(i);                 // référence acceptée → Object
                        // Output : Object
```

14.3.1.9 — Quand Object gagne

```
void fun(List<String> o)    { System.out.println("O"); }
void fun(CharSequence x)    { System.out.println("X"); }
void fun(Object y)          { System.out.println("Y"); }

fun(LocalDate.now());       // Output : Y
```

14.3.1.10 Tableau récapitulatif (Résolution de la surcharge)

Situation	Règle
Correspondance exacte	Toujours choisie
Élargissement primitif vs boxing	L'élargissement l'emporte
Boxing vs varargs	Le boxing l'emporte
Type de référence le plus spécifique	L'emporte
Types de référence non liés	Ambigu → erreur de compilation
Primitif + wrapper mélangés	Élargissement → boxing → varargs

14.4 Variables locales et d'instance

14.4.1 Variables d'instance

Les variables d'instance sont :

- déclarées comme membres d'une classe
- créées lorsqu'un objet est instancié
- accessibles par toutes les méthodes de l'instance

Modificateurs possibles pour les variables d'instance :

- modificateurs d'accès (`public` , `protected` , `private`)
- `final`
- `volatile`
- `transient`
- Exemple :

```
public class Person {
    private String name;          // variable d'instance
    protected final int age = 0; // final signifie ne peut pas être réaffectée
}
```

14.4.2 Variables locales

Les variables locales :

- sont déclarées *à l'intérieur* d'une méthode, d'un constructeur ou d'un bloc
- n'ont **aucune valeur par défaut** → doivent être explicitement initialisées avant utilisation
- seul modificateur autorisé : **final**
- Exemple :

```
void calculate() {
    int x;         // déclarée
    x = 10;        // doit être initialisée avant utilisation

    final int y = 5;  // légal
}
```

Deux cas particuliers :

14.4.2.1 Variables locales effectivement final

Une variable locale est *effectivement final* si elle est **assignée une seule fois**, même sans `final` .

Les variables effectivement final peuvent être utilisées dans :

- expressions lambda
- classes locales/anonimes

14.4.2.2 Paramètres comme effectivement final

Les paramètres de méthode se comportent comme des variables locales et suivent les mêmes règles.

14.5 Varargs (Listes d'arguments à longueur variable)

Les varargs permettent à une méthode d'accepter **zéro ou plusieurs** paramètres du même type.

Syntaxe :

```
void printNames(String... names)
```

Règles :

- Une méthode peut avoir **un seul** paramètre varargs.
- Il doit être le **dernier** paramètre dans la liste.
- Les varargs sont traités comme un **tableau** à l'intérieur de la méthode.
- Exemple :

```
void show(int x, String... values) {
    System.out.println(values.length);
}

show(10);                    // length = 0
show(10, "A");               // length = 1
show(10, "A", "B", "C");     // length = 3
```

> **Important**
>
> Les varargs et les tableaux participent à la surcharge des méthodes. La résolution de la surcharge peut devenir ambiguë.

14.6 Méthodes statiques, variables statiques et initialiseurs statiques

En Java, le mot-clé `static` marque des éléments qui **appartiennent à la classe elle-même**, et non aux instances individuelles. Cela signifie :

- Ils sont **chargés une seule fois** en mémoire lorsque la classe est chargée pour la première fois par la JVM.
- Ils sont partagés entre **toutes les instances**.
- Ils peuvent être accessibles **sans créer d'objet** de la classe.

Les membres statiques sont stockés dans la **method area** de la JVM (mémoire au niveau de la classe), tandis que les membres d'instance vivent dans le **heap**.

14.6.1 Variables statiques (Variables de classe)

Une **variable statique** est une variable définie au niveau de la classe et partagée par toutes les instances.

Caractéristiques :

- Créées lorsque la classe est chargée.
- Existent **même si aucune instance** de la classe n'est créée.
- Tous les objets voient la **même valeur**.
- Peuvent être marquées `final`, `volatile` ou `transient`.
- Exemple :

```
public class Counter {
    static int count = 0;     // partagée par toutes les instances
    int id;                   // variable d'instance

    public Counter() {
        count++;
        id = count;           // chaque instance obtient un id unique
    }
}
```

14.6.2 Méthodes statiques

Une **méthode statique** appartient à la classe, et non à une instance d'objet.

Règles :

- Elles peuvent être appelées en utilisant le nom de la classe : `ClassName.method()`.
- Elles **ne peuvent pas** accéder directement aux variables ou méthodes d'instance, mais uniquement via une instance de la classe.
- Elles **ne peuvent pas** utiliser `this` ou `super`.
- Elles sont couramment utilisées pour :
- - des méthodes utilitaires (ex. `Math.sqrt()`)
- - des méthodes de fabrique

- - des comportements globaux qui ne dépendent pas de l'état d'instance
- Exemple :

```
public class MathUtil {

    static int square(int x) {         // méthode statique
        return x * x;
    }

    void instanceMethod() {
        // System.out.println(count);   // OK : accès à une variable statique
        // square(5);                   // OK : méthode statique accessible
    }
}
```

Erreurs courantes :

```
// ❌ Erreur de compilation : une méthode d'instance ne peut pas être appelée directement dans
static void go() {
    run();         // run() est une méthode d'instance !
}

void run() { }
```

14.6.3 Blocs d'initialisation statique

Les blocs d'initialisation statique permettent d'exécuter du code **une seule fois**, lorsque la classe est chargée.

Syntaxe :

```
static {
    // logique d'initialisation
}
```

Utilisation :

- initialisation de variables statiques complexes
- exécution d'un setup au niveau de la classe
- exécution de code qui doit être exécuté exactement une fois
- Exemple :

```
public class Config {

    static final Map<String, String> settings = new HashMap<>();

    static {
        settings.put("mode", "production");
        settings.put("version", "1.0");
        System.out.println("Static initializer executed");
    }
}
```

> **Important**
>
> Les blocs d'initialisation statique s'exécutent **une seule fois**, dans l'ordre où ils apparaissent, avant `main()` et avant qu'une méthode statique ne soit appelée.

14.6.4 Ordre d'initialisation (Statique vs. Instance)

(Se référer au chapitre : 15. Class Loading, Initialization, and Object Construction)

14.6.5 Accès aux membres statiques

14.6.5.1 Utiliser le nom de la classe

```
Math.sqrt(16);
MyClass.staticMethod();
```

14.6.5.2 Via une référence d'instance

```
MyClass obj = new MyClass();
obj.staticMethod();
```

14.6.6 Statique et héritage

Les méthodes statiques :

- peuvent être **masquées**, pas redéfinies
- le binding est **à la compilation**, pas à l'exécution
- sont accessibles selon le **type de la référence**, et non le type de l'objet
- Exemple :

```
class A {
    static void test() { System.out.println("A"); }
}

class B extends A {
    static void test() { System.out.println("B"); }
}

A ref = new B();
ref.test();   // affiche "A" — binding statique !
```

> **Note**
>
> Règle clé : les méthodes statiques utilisent le **type de la référence**, et non le type de l'objet.

14.6.7 Pièges courants

- Tenter de référencer une variable ou une méthode d'instance depuis un contexte statique.
- Supposer que les méthodes statiques sont redéfinies → elles sont **masquées**.
- Appeler une méthode statique via une référence d'instance (légal mais déroutant).
- Confondre l'ordre d'initialisation des éléments statiques et des éléments d'instance.
- Oublier que les variables statiques sont partagées entre tous les objets.
- Ignorer que les initialiseurs statiques s'exécutent *une seule fois*, dans l'ordre de déclaration.

◀ 13. Mise en forme et localisation en Java | ▲ Index | 15. Chargement des Classes, Initialisation et Construction des Objets ▶

15. Chargement des Classes, Initialisation et Construction des Objets

Table des matières

En Java, comprendre **comment les classes sont chargées**, **comment les membres statiques et d'instance sont initialisés**, et **comment les constructeurs s'exécutent — en particulier avec l'héritage** — est essentiel pour maîtriser le langage.

Ce chapitre fournit une explication unifiée et claire de :

- Comment une classe est chargée en mémoire
- Comment les variables statiques et les blocs statiques sont exécutés
- Comment les objets sont créés étape par étape
- Comment les constructeurs s'exécutent dans une chaîne d'héritage
- Comment différentes zones mémoire (Heap, Stack, Method Area) participent

15.1 Zones Mémoire Java Pertinentes pour l'Initialisation des Classes et des Objets

Avant de comprendre l'ordre d'initialisation, il est utile de rappeler les trois principales zones mémoire impliquées :

- **Method Area (aussi appelée Class Area)** — stocke les métadonnées des classes, les variables statiques et les blocs d'initialisation statique.
- **Heap** — stocke tous les objets et les champs d'instance.
- **Stack** — stocke les appels de méthodes, les variables locales et les références.

> **Note**
>
> Les membres statiques appartiennent à la **classe** et sont créés **une seule fois** dans la Method Area. Les membres d'instance appartiennent à **chaque objet** et vivent dans le **Heap**.

15.2 Chargement des Classes (avec Héritage)

Quand un programme Java démarre, la JVM charge les classes *à la demande*.

Quand une classe est référencée pour la première fois (par exemple via `new` ou en accédant à un membre statique), **toute sa chaîne d'héritage doit d'abord être chargée**.

15.2.1 Ordre de Chargement des Classes

Étant donnée une hiérarchie de classes :

```
class A { ... }
class B extends A { ... }
class C extends B { ... }
```

Si le code exécute :

```
public static void main(String[] args) {
    new C();
}
```

Alors le chargement des classes se déroule dans cet ordre strict :

- Charger la classe A
- Initialiser les variables statiques de A (valeurs par défaut → explicites)
- Exécuter les blocs d'initialisation statique de A (du haut vers le bas)
- Charger la classe B et répéter la même logique
- Charger la classe C et répéter la même logique

15.2.2 Que se Passe-t-il Pendant le Chargement d'une Classe

- **Étape 1 : Les variables statiques sont allouées** (d'abord avec les valeurs par défaut).
- **Étape 2 : Les initialisations statiques explicites s'exécutent**.
- **Étape 3 : Les blocs d'initialisation statique** s'exécutent dans l'ordre du code source.

> **Note**
>
> Après ces étapes, la classe est complètement prête et peut maintenant être utilisée (instanciée ou référencée).

> **Warning**
>
> L'accès à un champ statique provoque l'initialisation uniquement de la classe ou de l'interface qui le déclare directement.
>
> Cela reste vrai même si le champ est référencé en utilisant le nom d'une sous-classe, d'une sous-interface ou d'une classe qui implémente l'interface.

15.3 Création d'Objets (avec Héritage)

Quand le mot-clé `new` est utilisé, **la création d'instance suit une séquence stricte et prévisible** impliquant toutes les classes parentes.

15.3.1 Ordre Complet de Création des Instances

- **1. La mémoire est allouée dans le Heap pour le nouvel objet** (les champs reçoivent des valeurs par défaut).
- **2. La chaîne des constructeurs s'exécute du parent vers l'enfant** — le sommet de la hiérarchie s'exécute en premier, puis chaque sous-classe.
- **3. Les variables d'instance reçoivent leurs initialisations explicites**.
- **4. Les blocs d'initialisation d'instance s'exécutent**.
- **5. Le corps du constructeur s'exécute** : pour chaque classe dans la chaîne d'héritage, les étapes 3–5 (initialisation des champs, blocs d'instance, corps du constructeur) s'appliquent du parent vers l'enfant.

15.4 Exemple Complet : Initialisation Statique + d'Instance à Travers l'Héritage

Considérons la hiérarchie suivante à trois niveaux :

```
class A {
    static int sa = init("A static var");

    static {
        System.out.println("A static block");
    }

    int ia = init("A instance var");

    {
        System.out.println("A instance block");
    }

    A() {
        System.out.println("A constructor");
    }

    static int init(String msg) {
        System.out.println(msg);
        return 0;
    }
}

class B extends A {
    static int sb = init("B static var");

    static {
        System.out.println("B static block");
    }

    int ib = init("B instance var");

    {
        System.out.println("B instance block");
    }

    B() {
        System.out.println("B constructor");
    }
}

class C extends B {
    static int sc = init("C static var");

    static {
        System.out.println("C static block");
    }

    int ic = init("C instance var");

    {
        System.out.println("C instance block");
    }

    C() {
        System.out.println("C constructor");
    }
}

public class Test {
    public static void main(String[] args) {
        new C();
    }
}
```

Sortie

```
A static var
A static block
B static var
B static block
C static var
C static block
A instance var
A instance block
A constructor
B instance var
B instance block
B constructor
C instance var
C instance block
C constructor
```

15.5 Diagramme de Visualisation

```
            CHARGEMENT DES CLASSES (du haut vers le bas)

                A  --->  B  --->  C
                |        |        |
      variables statiques + blocs statiques exécutés dans l'ordre

------------------------------------------------------

            CRÉATION DE L'OBJET (du parent vers l'enfant)

 new C()
    |
    +--> allocation mémoire pour C (valeurs par défaut)
    +--> appel du constructeur B()
            |
            +--> appel du constructeur A()
                    |
                    +--> initialise les variables d'instance de A
                    +--> exécute les blocs d'instance de A
                    +--> exécute le constructeur A
            +--> initialise les variables d'instance de B
            +--> exécute les blocs d'instance de B
            +--> exécute le constructeur B
    +--> initialise les variables d'instance de C
    +--> exécute les blocs d'instance de C
    +--> exécute le constructeur C
```

15.6 Règles Clés

- L'initialisation statique se produit **une seule fois** par classe.
- Les initialisateurs statiques s'exécutent dans l'ordre parent → enfant.
- L'initialisation d'instance se produit chaque fois qu'un objet est créé.
- Pour chaque classe dans la chaîne d'héritage, les champs d'instance et les blocs d'instance s'exécutent avant le corps du constructeur de cette classe.
- Globalement, l'initialisation des champs/blocs d'instance et les constructeurs s'exécutent du parent vers l'enfant.
- Les constructeurs appellent toujours le constructeur du parent (explicitement ou implicitement).

15.7 Tableau Récapitulatif

STATIQUE (Niveau Classe)	**INSTANCE (Niveau Objet)**
Une seule fois	Se produit à chaque `new`
Exécuté parent → enfant	Initialisation d'instance et constructeurs parent → enfant
variables statiques (défaut → explicites)	variables d'instance (défaut → explicites)
blocs statiques	blocs d'instance + constructeur

◀ 14. Méthodes, Attributs et Variables | ▲ Index | 16. Héritage en Java ▶

16. Héritage en Java

Table des matières

L'`Inheritance` (Héritage) est l'un des piliers fondamentaux de l'Object-Oriented Programming.

Elle permet à une classe `fille` (child), la **subclass**, d'acquérir l'état et le comportement d'une autre classe `génitrice` (parent), la **superclass**, en créant des relations hiérarchiques qui promeuvent la réutilisation du code, la spécialisation et le polymorphisme.

16.1 Définition Générale de l'Héritage

L'héritage permet à une classe d'en étendre une autre, en obtenant automatiquement ses `attributs` et ses `méthodes` accessibles.

La classe qui étend peut ajouter de nouvelles fonctionnalités ou redéfinir (faire `override`) les comportements existants, en créant des versions plus spécialisées de sa propre classe parent.

> **Note**
>
> L'Héritage exprime une relation *"is-a"* (est-un) : un Chien **is a** (est-un) Animal.

16.2 Héritage Simple et java.lang.Object

Java supporte la **single inheritance**, ce qui signifie que chaque classe peut étendre **une seule** superclasse directe.

Toutes les classes héritent en dernière analyse de `java.lang.Object`, qui se trouve au sommet de la hiérarchie.

Cela garantit que tous les objets Java partagent un comportement minimal commun (par exemple les méthodes `toString()`, `equals()`, `hashCode()`).

```
class Animal { }
class Dog extends Animal { }

// All classes implicitly extend Object
System.out.println(new Dog() instanceof Object); // true
```

16.3 Héritage Transitif

L'`Inheritance` est **transitif**.

Si la classe `C` étend `B` et `B` étend `A`, alors `C` hérite effectivement des membres accessibles à la fois de `B` *et* de `A`.

```
class A { }
class B extends A { }
class C extends B { } // C inherits from both A and B
```

16.4 Ce Qui Est Hérité, Bref Promemoria

Une subclass hérite de tous les membres **accessibles** de la classe génitrice.

Cependant, spécifiquement, cela dépend des `access modifiers`.

- **public** → toujours hérité
- **protected** → hérité si accessible via règles de package ou subclass
- **default (package-private)** → hérité seulement dans le même package
- **private** → **NON** hérité

> **Note**
>
> (Faire référence au Paragraphe "**Access Modifiers**" dans le Chapitre: Briques de base du langage Java)

16.5 Modificateurs de Classe qui influencent l'Héritage

Certains modificateurs au niveau de la classe déterminent si une classe peut être étendue.

Modifier	Signification	Effet sur l'Héritage
`final`	La classe ne peut pas être étendue	Inheritance STOP
`abstract`	La classe ne peut pas être instanciée	Doit être étendue
`sealed`	Permet seulement une liste fixe de subclass	Restreint l'inheritance
`non-sealed`	Subclass d'une sealed class qui rouvre l'inheritance	Inheritance permis
`static`	S'applique seulement aux nested classes	Se comporte comme une top-level class à l'intérieur de sa classe conteneur

> **Note**
>
> Une classe `static` en Java peut exister seulement comme **static nested class**.

16.6 Références `this` et `super`

16.6.1 La Référence `this`

La référence `this` se réfère à l'instance courante de l'objet et permet de lever l'ambiguïté d'accès aux membres courants et hérités.

Java utilise une règle de **granular scope**:

- Si une variable de méthode/locale a le même nom qu'un `instance field`, celle locale "masque" l'attribut d'instance.
- Il est nécessaire d'utiliser `this.fieldName` pour accéder donc à l'attribut d'instance.

```
public class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Si les noms diffèrent, `this` est optionnel.

```
public class Person {
    String name;

    public Person(String n) {
        name = n;
    }
}
```

> **Warning**
>
> `this` NE peut PAS être utilisé à l'intérieur de méthodes statiques parce que, dans ce contexte, aucune instance n'existe.

16.6.2 La Référence `super`

La référence `super` donne accès aux membres de la classe génitrice (parent) directe.

Utile quand:

- Le parent (genitore) et le child (figlio) définissent un attribut/méthode avec le même nom; voir section: Hériter des Membres
- Parent et child définissent un attribut avec le même nom → `variable hiding` (deux copies)
- Parent et child définissent une méthode avec la même signature → `method overriding`
- On veut appeler explicitement l'implémentation héritée

```java
class Parent { int value = 10; }

class Child extends Parent {
    int value = 20;

    void printBoth() {
        System.out.println(value);      // child value
        System.out.println(super.value); // parent value
    }
}
```

> **Note**
>
> `super` NE peut PAS être utilisé dans des contextes statiques.

16.7 Déclarer des Constructeurs dans une chaîne héréditaire

Un `constructeur` initialise un objet nouvellement créé.

Les constructeurs ne sont **jamais hérités**, mais chaque constructeur de subclass doit s'assurer que la classe parent soit initialisée.

Les `constructeurs` sont des méthodes spéciales avec un nom qui correspond au nom de la classe et qui ne déclarent aucun return type.

Une classe peut définir plusieurs constructeurs (constructor overloading), chacun avec une `signature` unique.

On peut déclarer explicitement un `no-arg constructor` ou n'importe quel constructeur spécifique ou, si on ne le fait pas, Java créera implicitement un `default no-arg constructor`.

Si on déclare explicitement un constructeur, le compilateur Java n'inclura aucun `default no-arg constructor` : cette règle s'applique indépendamment à chaque classe dans la hiérarchie.

Une classe parent continue d'avoir son propre constructeur par défaut à moins qu'elle n'en définisse aussi un.

16.8 Constructeur `no-arg` par Défaut

Si une classe ne déclare aucun constructeur, Java insère automatiquement un **default no-argument constructor**.

Ce constructeur invoquera le constructeur `super()` du parent direct, implicitement: le compilateur Java insère implicitement un appel au no-arg constructor `super()`.

```java
class Parent { }

class Child extends Parent {
    // Compiler inserts:
    // Child() { super(); }
}
```

16.9 Utiliser `this()` et Constructor Overloading

this() invoque un autre constructeur dans la même classe.

Règles:

- Doit être la **première** instruction dans le constructeur
- Ne peut pas être combiné avec `super()`
- Une seule invocation à `this()` est autorisée dans un constructeur
- Utilisé pour centraliser l'initialisation

```
class Car {
    int year;
    String model;

    Car() {
        this(2020, "Unknown");
    }

    Car(int year, String model) {
        this.year = year;
        this.model = model;
    }
}
```

16.10 Appeler le Constructeur du Parent en utilisant `super()`

Chaque constructeur doit appeler un constructeur de la superclasse, explicitement ou implicitement.

L'appel à `super()` doit apparaître comme **première** instruction dans le constructeur (à moins qu'il ne soit remplacé par `this()`).

```
class Parent {
    Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
    Child() {
        super(); // optional, compiler would insert it
        System.out.println("Child constructor");
    }
}
```

16.11 Conseils et Pièges sur le Constructeur par Défaut

- **Si la classe parent n'a pas de no-arg constructor, la classe fille DOIT invoquer le spécifique `super(args)` explicitement.**
- Si la classe fille ne définit aucun constructeur, Java ne crée pas automatiquement un constructeur par défaut pour celle-ci.
- Si on oublie d'appeler explicitement un `parent constructor` existant, le compilateur insère `super()` — lequel pourrait ne pas exister.

```
class Parent {
    Parent(int x) { }
}

class Child extends Parent {
    // ERROR → compiler inserts super(), but Parent() does not exist
    Child() { }
}
```

16.12 `super()` se Réfère Toujours au Parent le plus direct

Même dans de longues chaînes héréditaires, `super()` invoque toujours (et seulement) le constructeur de la classe génitrice **immédiate**.

```
class A {
    A() { System.out.println("A"); }
}
class B extends A {
    B() { System.out.println("B"); }
}
class C extends B {
    C() {
        super(); // Calls B(), not A()
        System.out.println("C");
    }
}
```

Output:

```
A
B
C
```

16.13 Hériter des Membres

En Java, l'**accès aux champs** et les **appels de méthodes statiques** sont résolus à la compilation, tandis que les **appels de méthodes d'instance** sont résolus à l'exécution.

La distinction essentielle est la suivante :

- La `variable` ou la `méthode statique` utilisée dépend du **type déclaré de la référence**.
- La `méthode d'instance` exécutée dépend du **type réel de l'objet à l'exécution**.

Exemple : `Accès aux Champs` (Non Polymorphique)

Les champs sont résolus en fonction du **type déclaré de la référence**, et non du type réel de l'objet.

```
class Parent {
    String name = "Parent";
}

class Child extends Parent {
    String name = "Child";
}

Parent p = new Child();
System.out.println(p.name);   // Output: Parent
```

Explication :

- La référence `p` est déclarée de type `Parent`.
- L'accès aux champs est déterminé à la compilation.
- Par conséquent, `Parent.name` est utilisé, même si l'objet est un `Child`.

> **Important**
>
> - Les champs **ne sont pas polymorphiques**.

Exemple : `Méthodes Statiques` (Non Polymorphiques)

Les méthodes statiques sont également résolues en utilisant le **type déclaré de la référence**.

```java
class Parent {
    static void print() {
        System.out.println("Parent static");
    }
}

class Child extends Parent {
    static void print() {
        System.out.println("Child static");
    }
}

Parent p = new Child();
p.print();   // Output: Parent static
```

Explication :

- Les méthodes statiques sont liées (binding) à la compilation.
- La méthode choisie dépend du type de la référence (`Parent`), et non du type réel de l'objet.

> **Important**
>
> - Ce mécanisme s'appelle le **method hiding**, et non l'override.

Exemple : `Méthodes d'Instance` (Polymorphiques)

Les méthodes d'instance sont résolues à l'exécution en fonction du **type réel de l'objet**.

```java
class Parent {
    void print() {
        System.out.println("Parent instance");
    }
}

class Child extends Parent {
    @Override
    void print() {
        System.out.println("Child instance");
    }
}

Parent p = new Child();
p.print();   // Output: Child instance
```

Explication :

- Le type de la référence est `Parent` .
- L'objet réel est de type `Child` .
- Java utilise le dynamic dispatch.
- Par conséquent, `Child.print()` est exécuté.

> **Important**
>
> - Les méthodes d'instance sont **polymorphiques**.

16.13.1 Method Overriding

Le `method overriding` est un concept fondamental de l'héritage: il permet à une classe fille de fournir une **nouvelle implémentation** pour une méthode déjà définie dans une de ses classes parent.

À runtime, la version de la méthode exécutée dépend du **type réel de l'objet**, pas du particulier `reference type` .

Ce comportement est appelé **dynamic dispatch** et c'est ce qui rend possible le polymorphisme en Java.

16.13.1.1 Définition et Rôle dans l'Héritage

Une méthode dans une subclass fait **override** d'une méthode d'une de ses superclass si:

- la méthode de la superclass est `méthode d'instance` (non statique).
- la méthode de la subclass a le **même nom**, la **même liste de paramètres** et un **return type qui est du même type** ou d'un **sous-type** du return type dans la méthode héritée.
- Lorsque le type de retour de la méthode redéfinie (c'est-à-dire la méthode dans la classe de base/superclasse) est un type **primitif**, le type de retour de la méthode qui la redéfinit (c'est-à-dire la méthode dans la sous-classe) doit correspondre exactement au type de retour de la méthode redéfinie.
- les deux méthodes sont accessibles (non privées) et la méthode de la subclass n'est pas moins visible que celle de la superclass.
- La méthode en overriding **ne peut pas déclarer de nouvelles ou plus larges checked exceptions**.

L'Overriding est utilisé pour spécialiser le comportement: une subclass peut adapter ou affiner le comportement de la classe parent, tout en pouvant être utilisée via une référence du type parent.

```
class Animal {
    void speak() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {

    @Override
    void speak() {
        System.out.println("Woof!");
    }
}

public class TestOverride {
    public static void main(String[] args) {
        Animal a = new Dog(); // reference type = Animal, object type = Dog
        a.speak(); // prints "Woof!" (Dog implementation)
    }
}
```

On ne peut pas redéfinir une méthode d instance avec une méthode statique, ni redéfinir une méthode statique avec une méthode d instance.

Cependant, une sous-interface est autorisée à redéclarer une méthode statique d une superinterface comme méthode `default`.

- Exemple :

```
class Alpha {
    static void a() { }
    void b() { }
    static void c() { }
    void d() { }
}

class Beta extends Alpha {
    void a() { }        // NE COMPILE PAS (impossible de redéfinir une méthode statique avec u
    static void b() { } // NE COMPILE PAS (impossible de redéfinir une méthode d instance avec
    static void c() { } // VALIDE, c() dans Alpha est masquée
    void d() { }        // VALIDE, d() dans Alpha est redéfinie
}
```

16.13.1.2 Utiliser super pour appeler l'Implémentation du Parent

Quand une subclass fait override d'une méthode, elle peut quand même accéder à l'implémentation "originelle" de la superclass, via la référence `super`.

Cela est utile si on veut réutiliser ou étendre le comportement défini dans la classe parent.

```java
class Person {
    void introduce() {
        System.out.println("I am a person.");
    }
}

class Student extends Person {
    @Override
    void introduce() {
        super.introduce(); // calls Person.introduce()
        System.out.println("I am also a student.");
    }
}
```

Si la classe parent et la classe child déclarent toutes deux un membre (attribut ou méthode) avec le même nom, le child peut accéder aux deux:

- la version en overriding (default)
- la version du parent via `super`

```java
class Base {
    int value = 10;

    void show() {
        System.out.println("Base value = " + value);
    }
}

class Derived extends Base {
    int value = 20; // hides Base.value

    @Override
    void show() {
        System.out.println("Derived value = " + value);          // 20
        System.out.println("Base value via super = " + super.value); // 10
    }
}
```

16.13.1.3 Règles de Overriding (Instance Methods)

- **Même signature (signature)**: même nom de méthode, mêmes types et ordre des paramètres.
- **return type covariant**: la méthode en overriding peut restituer (retourner) le même type du parent, ou un **subtype** du return type du parent.
- **Accessibilité**: la méthode en overriding ne peut pas être moins accessible que la méthode originelle (par exemple, on ne peut pas passer de `public` à `protected` ou `private`). Elle peut seulement maintenir la même visibilité ou l'augmenter.
- **Checked exceptions**: la méthode en overriding ne peut pas déclarer de nouvelles ou plus larges `checked exceptions` par rapport au `parent method`; elle peut en déclarer moins, déclarer des checked exceptions plus spécifiques ou, éventuellement, les enlever complètement.
- **Unchecked exceptions**: elles peuvent être ajoutées ou enlevées sans restrictions.
- **final methods**: elles ne peuvent pas participer à l'`override`.

```java
class Parent {
    Number getValue() throws Exception {
        return 42;
    }
}

class Child extends Parent {
@Override
    // Covariant return type: Integer is a subclass of Number
    Integer getValue() throws RuntimeException {
        return 100;
    }
}
```

16.13.1.4 Masquer Static Methods (Method Hiding)

Les méthodes statiques ne participent pas à l'`overriding`; elles sont au contraire, éventuellement, masquées (**hidden**).

Si une subclass définit un static method avec la même signature d'un static method de la classe parent, la méthode statique de la subclass **masque** celle de la classe génitrice.

Si l'une des méthodes est marquée comme `static` et l'autre non, le code ne compilera pas.

La sélection de la méthode pour les méthodes statiques arrive à **compile time** et est basée sur le `reference type` : pas sur l'`object type`.

```
class A {
    static void show() {
        System.out.println("A.show()");
    }
}

class B extends A {
    static void show() {
        System.out.println("B.show()");
    }
}

public class TestStatic {
    public static void main(String[] args) {
        A a = new B();
        B b = new B();

        a.show(); // A.show()  (reference type A)
        b.show(); // B.show()  (reference type B)
    }
}
```

> **Important**
>
> - méthodes statiques **final** ne peuvent pas être `hidden` (masquées); méthodes d'instance déclarées **final** ne peuvent pas être `overriden`.
> - Si on essaye de les redéfinir dans une subclass, le code ne compilera pas.

16.13.2 Abstract Classes

16.13.2.1 Définition et But

Une **abstract class** est une classe qui ne peut pas être instanciée directement et est destinée à être étendue.

Elle peut contenir:

- méthodes abstract (déclarées sans body);
- méthodes concrètes (avec implémentation);
- attributs, constructeurs, membres statiques, et aussi static initializers.

Les abstract classes sont utilisées quand on veut définir un comportement commun (et un contrat) de **base**, mais laisser certains détails à implémenter aux subclasses concrètes.

16.13.2.2 Règles pour les Abstract Classes

- Une classe avec au moins une méthode abstraite **doit** être déclarée `abstract`.
- Une `abstract class` **ne peut pas** être instanciée directement.
- Les méthodes abstraites n'ont pas de body et terminent avec un point-virgule.
- Les **abstract methods ne peuvent pas être** `final`, `static` **ou** `private`, parce qu'elles doivent être redéfinissables `overridable`.
- La première subclass concrète (non-abstract) dans la hiérarchie, doit implémenter tous les `abstract methods` hérités, sinon elle doit être déclarée elle aussi `abstract`.

```java
abstract class Shape {

    abstract double area(); // must be implemented by concrete subclasses

    void describe() {
        System.out.println("I am a shape.");
    }

    Shape() {
        System.out.println("Shape constructor");
    }
}

class Circle extends Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
}
```

> **Note**
>
> - Bien qu'une `abstract class` ne puisse pas être instanciée, ses constructeurs sont quand même appelés quand on crée des instances de classes filles concrètes.
> - Le flux des instanciations, dans la `chaîne héréditaire`, part toujours du sommet de la hiérarchie et se déplace vers le bas.

16.13.3 Créer des Objets Immutables

16.13.3.1 Qu'est-ce qu'un Objet `Immutable`

Un objet est **immutable** si, après qu'il a été créé, son état **ne peut pas changer**.

Tous les attributs qui représentent son état, restent constants pour l'ensemble du cycle de vie de cet objet.

Les `immutable objects` sont simples à comprendre, intrinsèquement `thread safe` (si conçus correctement), et largement utilisés dans la Java Standard Library (par exemple `String`, wrapper classes comme `Integer`, et beaucoup de classes dans `java.time`).

16.13.3.2 Lignes Directrices pour Concevoir des Classes Immutable

- Déclarer une classe **final** afin qu'elle ne puisse pas être étendue (ou bien rendre tous les constructeurs privés et fournir des factory methods protégés).
- Rendre tous les attributs qui représentent son état **private** et **final**.
- Ne fournir aucune méthode `mutator` (setter).
- Initialiser tous les attributs dans les constructeurs (ou dans les factory methods) et ne jamais les exposer de façon `mutable`.
- Si un attribut se réfère à un objet mutable, faire des **defensive copies** (copies défensives) en phase de construction et quand on le restitue via des `getters`.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public final class Person {
    private final String name; // String is immutable
    private final int age;
    private final List<String> hobbies; // List is mutable, we must protect it

    public Person(String name, int age, List<String> hobbies) {
        this.name = name;
        this.age = age;
        // Defensive copy on input
        this.hobbies = new ArrayList<>(hobbies);
    }

    public String getName() {
        return name; // safe (String is immutable)
    }

    public int getAge() {
        return age;
    }

    public List<String> getHobbies() {
        // Defensive copy or unmodifiable view on output
        return Collections.unmodifiableList(hobbies);
    }
}
```

Dans cet exemple:

- `Person` est **final**: elle ne peut pas être étendue et son comportement ne peut pas être altéré via `inheritance`.
- Tous les attributs sont `private` et `final`, définis une seule fois dans le constructeur.
- La liste des `hobbies` est copiée défensivement dans la phase de construction et wrappée comme `unmodifiable` (non modifiable) dans la `méthode getter`, afin qu'aucun code externe ne puisse modifier l'état interne.

Concevoir des `immutable objects` est particulièrement important dans des contextes multithread et quand on passe des objets à travers les différents layers d'une application.

◀ 15. Chargement des Classes, Initialisation et Construction des Objets | ▲ Index | 17. Au-delà des Classes ▶

17. Au-delà des Classes

Table des matières

Ce chapitre présente plusieurs mécanismes de type (type) avancés, au-delà de celui, déjà étudié, de la classe : **interfaces**, **enum**, **classes sealed / non-sealed**, **records** et **classes imbriquées**.

17.1 Interfaces

Une **interface** en Java est un type de référence qui définit un contrat de méthodes qu'une classe accepte d'implémenter.

Une `interface` est implicitement `abstract` et ne peut pas être marquée `final` : comme pour les classes top-level, une interface peut déclarer une visibilité `public` ou `default` (package-private).

Une classe Java peut implémenter un nombre quelconque d'interfaces via le mot-clé `implements`.

Une `interface` peut à son tour étendre plusieurs interfaces en utilisant le mot-clé `extends`.

Les interfaces permettent l'abstraction, un couplage faible et l'héritage multiple de type.

17.1.1 Ce que les Interfaces Peuvent Contenir

- **Méthodes abstraites** (implicitement `public` et `abstract`)
- **Méthodes concrètes**
  - **Méthodes default** (contiennent du code et sont implicitement `public`)
  - **Méthodes static** (déclarées `static`, contiennent du code et sont implicitement `public`)
  - **Méthodes private** (Java 9+) pour la réutilisation interne
- **Constantes** → implicitement `public static final` et initialisées à la déclaration

```
interface Calculator {

    int add(int a, int b);                  // abstract

    default int mult(int a, int b) {        // default method
        return a * b;
    }

    static double pi() { return 3.14; }     // static method
}
```

> **Warning**
>
> Puisque les méthodes abstraites des interfaces sont implicitement `public`, **vous ne pouvez pas** réduire le niveau d'accès sur une méthode d'implémentation.

17.1.2 Implémenter une Interface

```
class BasicCalc implements Calculator {
    public int add(int a, int b) { return a + b; }
}
```

> **Note**
>
> **Chaque** méthode abstraite doit être implémentée, sauf si la classe est elle-même abstraite.

17.1.3 Héritage Multiple

Une classe peut implémenter plusieurs interfaces.

```
interface A { void a(); }
interface B { void b(); }

class C implements A, B {
    public void a() {}
    public void b() {}
}
```

17.1.4 Héritage des Interfaces et Conflits

Si deux interfaces fournissent des méthodes `default` avec la même signature, la classe qui implémente doit redéfinir (override) la méthode.

```
interface X { default void run() { } }
interface Y { default void run() { } }

class Z implements X, Y {
    public void run() { } // mandatory
}
```

Si vous voulez tout de même accéder à une implémentation particulière de la méthode `default` héritée, vous pouvez utiliser la syntaxe suivante :

```
interface X { default int run() { return 1; } }
interface Y { default int run() { return 2; } }

class Z implements X, Y {

    public int useARun(){
        return Y.super.run();
    }

}
```

17.1.5 Méthodes `Default`

Une méthode `default` (déclarée avec le mot-clé `default`) est une méthode qui définit une implémentation et peut être redéfinie par une classe qui implémente l interface.

- Une méthode default contient du code et est implicitement `public` ;
- Une méthode default ne peut pas être `abstract`, `static` ou `final` ;
- Une interface peut redéclarer une méthode `default` et fournir une implémentation différente ;
- Une sous-interface est autorisée à redéclarer une méthode statique d'une superinterface comme méthode `default`.
- Comme nous l'avons vu juste au-dessus, si deux interfaces fournissent des méthodes default avec la même signature, la classe implémentante doit redéfinir la méthode ;
- Une classe implémentante peut bien sûr s'appuyer sur l'implémentation fournie par la méthode `default` sans la redéfinir ;
- La méthode `default` peut être invoquée sur une instance de la classe implémentante et NON comme une méthode `static` de l interface qui la contient ;
- Une classe (ou une interface) peut invoquer explicitement une méthode default d'une interface qui est directement mentionnée dans sa clause `implements` (ou `extends`) en utilisant la syntaxe `InterfaceName.super.methodName()` ; cela est généralement utilisé pour lever l'ambiguïté entre plusieurs méthodes default héritées ;
- Cette syntaxe ne peut être utilisée que si l'interface est explicitement mentionnée dans la clause `implements` (ou `extends`) ; elle ne peut pas être utilisée pour invoquer une méthode default

provenant d'une interface héritée indirectement ;

- La syntaxe `InterfaceName.super.methodName()` s'applique uniquement aux méthodes `default` et ne peut pas être utilisée pour des méthodes abstraites, des méthodes statiques, des méthodes privées d'interface ou des champs.

Example: Utilisation correcte

```
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

class B implements A {

    @Override
    public void hello() {
        A.super.hello();  // ✅ allowed
        System.out.println("Hello from B");
    }
}
```

Example: Utilisation incorrecte

```
interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}

interface B extends A {
}

class C implements B {

    public void test() {
        A.super.hello();  // ❌ compilation error
    }
}
```

> **Note**
>
> - Une sous-interface est autorisée à redéclarer une méthode statique d'une superinterface comme méthode `default`.

Exemple :

```
interface Parent {
    static void p() { }
}

interface Child extends Parent {
    default void p() { } // VALID, static method redeclared as default
}
```

> **Note**
>
> - Une interface est autorisée à redéclarer une méthode `default` héritée d une superinterface et à la transformer en méthode `abstract`.
>
> Lorsque cela se produit, l implémentation `default` provenant de la superinterface est effectivement supprimée dans la sous-interface. En conséquence, toute classe qui implémente la sous-interface N héritera PAS de l implémentation `default` originale et devra fournir sa propre implémentation.

Exemple :

```java
interface Parent {
    default void greet() {
        System.out.println("Hello from Parent");
    }
}

interface Child extends Parent {
    void greet();   // redéclarée comme abstraite
}

class Demo implements Child {
    public void greet() {   // obligatoire
        System.out.println("Hello from Demo");
    }
}
```

Explication :

- `Parent` fournit une implémentation par défaut de `greet()`.
- `Child` redéclare `greet()` sans `default`, la rendant à nouveau abstraite.
- `Demo` ne peut pas hériter de l implémentation par défaut de `Parent`.
- Par conséquent, `Demo` doit implémenter explicitement `greet()`.

17.1.6 Méthodes `static`

- Une interface peut fournir des `static methods` (via le mot-clé `static`) qui sont implicitement `public` ;
- Les méthodes static doivent inclure un corps de méthode et sont accessibles via le nom de l'interface ;
- Les méthodes static ne peuvent pas être `abstract` ou `final` ;

17.1.7 Méthodes `private` dans les interfaces

Parmi toutes les méthodes concrètes qu'une interface peut implémenter, nous avons aussi :

- **Méthodes `private`** : visibles uniquement à l'intérieur de l'interface déclarante et qui ne peuvent être invoquées que depuis un contexte `non-static` (méthodes `default` ou autres `non-static private methods`).
- **Méthodes `private static`** : visibles uniquement à l'intérieur de l'interface déclarante et qui peuvent être invoquées par n'importe quelle méthode de l'interface englobante.

17.2 `Types sealed`, `non-sealed` et `final`

Les classes et interfaces `sealed` (Java 17+) restreignent quelles autres classes (ou interfaces) peuvent les étendre ou les implémenter.

Un `sealed type` est déclaré en plaçant le modificateur `sealed` juste avant le mot-clé class (ou interface), et en ajoutant, après le nom du Type, le mot-clé `permits` suivi de la liste des types qui peuvent l'étendre (ou l'implémenter).

```java
public sealed class Shape permits Circle, Rectangle { }

final class Circle extends Shape { }

non-sealed class Rectangle extends Shape { }
```

17.2.1 Règles

- Un type sealed doit déclarer tous les sous-types autorisés.
- Un sous-type autorisé doit être **final**, **sealed** ou **non-sealed** ; puisque les interfaces ne peuvent pas être final, elles ne peuvent être marquées que `sealed` ou `non-sealed` lorsqu'elles étendent une interface sealed.

- Si une sealed class appartient à un `named` module, alors toutes les classes listées dans sa permits clause doivent également appartenir à ce `même module`.
- Si une sealed class appartient à un `unnamed module`, alors toutes les classes listées dans sa permits clause doivent être déclarées dans le `même package`.

17.3 Enum

Les **enum** définissent un ensemble fixe de valeurs constantes.

Les `enum` peuvent déclarer des `attributs`, des `constructeurs` et des `méthodes` comme des classes ordinaires mais **ne peuvent pas être étendues**.

La liste des valeurs de l'enum doit se terminer par un point-virgule `(;)` dans le cas des `Enum Complexes`, mais ce n'est pas obligatoire pour les `Enum Simples`.

17.3.1 Définition d'une Enum `Simple`

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN } // point-virgule omis
```

17.3.2 Enum `Complexes` avec État et Comportement

```
enum Level {
    LOW(1), MEDIUM(5), HIGH(10); // point-virgule obligatoire

    private int code;

    Level(int code) { this.code = code; }

    public int getCode() { return code; }
}

public static void main(String[] args) {
    Level.MEDIUM.getCode();     // invoking a method
}
```

17.3.3 Méthodes des Enum

- `values()` – renvoie un tableau de toutes les valeurs constantes utilisables, par exemple, dans une boucle `for-each`
- `valueOf(String)` – renvoie la constante par son nom
- `ordinal()` – index (int) de la constante

17.3.4 Règles

- **Les constructeurs d'enum sont implicitement** `private` ;
- Les enum peuvent contenir des méthodes `static` et `instance` ;
- Les enum peuvent implémenter des `interfaces` ;
- Les énumérations ne peuvent pas être étendues.

17.4 Records (Java 16+)

Un **record** est une classe spéciale conçue pour modéliser des données immuables : ils sont en effet implicitement **final**.

Vous ne pouvez pas étendre un record, mais il est permis d'implémenter une interface `normale` ou `sealed`.

Il fournit automatiquement :

- **champs private final** pour chaque composant ;
- **constructeur** avec des paramètres dans le même ordre que la déclaration du record ;

- **getters** (portant le nom des attributs) ;
- **`equals()`, `hashCode()`, `toString()`** : il est également permis de redéfinir (override) ces méthodes ;
- Les **Records** peuvent inclure `nested classes`, `interfaces`, `records`, `enums` et `annotations`.

```
public record Point(int x, int y) { }

var element = new Point(11, 22);

System.out.println(element.x);
System.out.println(element.y);
```

Si vous avez besoin de validation ou de transformation supplémentaire des champs fournis, vous pouvez définir un `constructeur long` ou un `constructeur compact`.

17.4.1 Résumé des Règles de Base pour les Records

Un record peut être déclaré à trois emplacements :

- Comme **record top-level** (directement dans un package)
- Comme **record member** (à l'intérieur d'une classe ou d'une interface)
- Comme **record local** (à l'intérieur d'une méthode)

Toutes les classes record `member` et `local` sont implicitement `static`.

- Un record member peut déclarer `static` de manière redondante.
- Un record local ne doit pas déclarer `static` explicitement.

Chaque classe record est implicitement `final`.

- Déclarer `final` explicitement est autorisé mais redondant.
- Un record ne peut pas être déclaré `abstract`, `sealed` ou `non-sealed`.

La superclasse directe de chaque record est `java.lang.Record`.

- Un record ne peut pas déclarer de clause `extends`.
- Un record ne peut étendre aucune autre classe.

La sérialisation des records diffère de celle des classes sérialisables ordinaires.

- Lors de la désérialisation, le constructeur canonique est invoqué.

Le corps d'un record peut contenir :

- Des constructeurs
- Des méthodes
- Des champs statiques
- Des blocs d'initialisation statiques

Le corps d'un record NE doit PAS contenir :

- Des déclarations de champs d'instance
- Des blocs d'initialisation d'instance
- Des méthodes `abstract`
- Des méthodes `native`

17.4.2 Constructeur Long

```
public record Person(String name, int age) {

    public Person (String name, int age){
        if (age < 0) throw new IllegalArgumentException();
        this.name = name;
        this.age = age;
    }
}
```

Vous pouvez aussi définir des constructeurs en surcharge (overload), à condition qu'ils délèguent finalement au constructeur canonique via `this(...)` :

```
public record Point(int x, int y) {

    // Overloaded constructor (NOT canonical)
    public Point(int value) {
        this(value, value); // doit invoquer, comme première instruction, un autre constructeu
    }
}
```

> **Note**
>
> - Le compilateur n'insérera pas de constructeur si vous en fournissez manuellement un avec la même liste de paramètres dans l'ordre défini ;
> - Dans ce cas, vous devez définir explicitement chaque champ manuellement ;

17.4.3 Constructeur Compact

Vous pouvez définir un `constructeur compact` qui initialise implicitement tous les champs, tout en vous permettant d'effectuer des validations et des transformations sur des champs spécifiques.

Java exécutera le constructeur complet, initialisant tous les champs, après que le constructeur compact a terminé.

```
public record Person(String name, int age) {

    public Person {
        if (age < 0) throw new IllegalArgumentException();

        name = name.toUpperCase(); // This transformation is still (at this level of initializ

        // this.name = name; // ❌ Does not compile.
    }
}
```

> **Warning**
>
> - Si vous essayez de modifier un attribut de Record dans un Constructeur Compact, votre code ne compilera pas

17.4.4 Pattern Matching pour les Records

Quand vous utilisez le pattern matching avec `instanceof` ou avec `switch`, un record pattern doit spécifier :

- Le type du record ;
- Un pattern pour chaque champ du record (correspondant au bon nombre de composants, et avec des types compatibles) ;

Exemple record :

```
Object obj = new Point(3, 5);

if (obj instanceof Point(int a, int b)) {
    System.out.println(a + b);   // 8
}
```

17.4.5 Nested Record Patterns et Matching des Records avec `var` et Generics

Les nested record patterns permettent de déstructurer des records qui contiennent d'autres records ou des types complexes, en extrayant récursivement des valeurs directement dans le pattern.

Ils combinent la puissance de la déconstruction des `record` avec le pattern matching, vous donnant une manière concise et expressive de naviguer dans des structures de données hiérarchiques.

17.4.5.1 Nested Record Pattern de Base

Si un record contient un autre record, vous pouvez déstructurer les deux en une seule fois :

```
record Address(String city, String country) {}
record Person(String name, Address address) {}

void printInfo(Object obj) {

    switch (obj) {
        case Person(String n, Address(String c, String co)) -> System.out.println(n + " lives
        default -> System.out.println("Unknown");
    }
}
```

Dans l'exemple ci-dessus, le pattern `Person` inclut un pattern `Address` imbriqué.

Les deux sont appariés structurellement.

17.4.5.2 Nested Record Patterns avec `var`

Au lieu de spécifier des types exacts pour chaque champ, vous pouvez utiliser `var` dans le pattern pour laisser le compilateur inférer le type.

```
    switch (obj) {
        case Person(var name, Address(var city, var country)) -> System.out.println(name + " -
    }
```

`var` dans les patterns fonctionne comme `var` dans les variables locales : cela signifie "inférer le type".

> **Warning**
>
> - Vous avez toujours besoin du type du record englobant (Person, Address) ;
> - seuls les types des champs peuvent être remplacés par `var`.

17.4.5.3 Nested Record Patterns et Generics

Les record patterns fonctionnent aussi avec des records génériques.

```
record Box<T>(T value) {}
record Wrapper(Box<String> box) {}

static void test(Object o) {
    switch (o) {
        case Wrapper(Box<String>(var v)) -> System.out.println("Boxed string: " + v);
        default -> System.out.println("Something else");
    }
}
```

Dans cet exemple :

- Le pattern exige exactement `Box<String>`, pas `Box<Integer>`.

- Dans le pattern, `var v` capture la valeur générique unboxed.

17.4.5.4 Erreurs Courantes avec les Nested Record Patterns

Structure de record non correspondante

```
// ❌ ERROR: pattern does not match record structure
case Person(var n, var city) -> ...
```

`Person` a 2 champs, mais l'un d'eux est un record. Vous devez déstructurer correctement.

Nombre incorrect de composants

```
// ❌ ERROR: Address has 2 components, not 1
case Person(var n, Address(var onlyCity)) -> ...
```

Incompatibilité générique

```
// ❌ ERROR: expecting Box<String> but found Box<Integer>
case Wrapper(Box<Integer>(var v)) -> ...
```

Placement illégal de `var`

```
// ❌ var cannot replace the record type itself
case var(Person(var n, var a)) -> ...
```

> **Note**
>
> - `var` ne peut pas remplacer l'ensemble du pattern, seulement les composants individuels.

17.5 Classes Imbriquées en Java

Java supporte plusieurs types de **classes imbriquées** — des classes déclarées à l'intérieur d'une autre classe.

Ce sont des outils fondamentaux pour l'encapsulation, l'organisation du code, les patterns d'event-handling et la représentation de hiérarchies logiques.

Une classe imbriquée appartient toujours à une **classe englobante** et a des règles particulières d'accessibilité et d'instanciation selon sa catégorie.

Java définit quatre types de classes imbriquées :

- **Static Nested Classes** – déclarées avec `static` à l'intérieur d'une autre classe.
- **Inner Classes** (non-static **nested** classes).
- **Local Classes** – déclarées dans un bloc (méthode, constructeur ou initializer).
- **Anonymous Classes** – classes sans nom créées inline, généralement pour redéfinir une méthode ou implémenter une interface.

> **Warning**
>
> - `static` s'applique uniquement aux classes **membres imbriquées**
> - Les classes `Top-level` → ne peuvent pas être static
> - Les classes `Local` (déclarées dans les méthodes) → ne peuvent pas être static
> - Les classes `Anonymous` → ne peuvent pas être static
> - Une classe `static nested` ne peut pas accéder aux membres d'instance sans une référence explicite à un objet externe.

17.5.1 Static Nested Classes

Une **static nested class** se comporte comme une classe top-level dont le namespace est à l'intérieur de sa classe englobante.
Elle ne **peut pas** accéder aux membres d'instance de la classe externe mais **peut** accéder aux membres statiques.
Elle ne conserve pas de référence vers une instance de la classe englobante. Une classe imbriquée `static` peut contenir des variables membres non statiques.

17.5.1.1 Syntaxe et Règles d'Accès

- Déclarée via `static class` à l'intérieur d'une autre classe.
- Peut accéder uniquement aux membres **static** de la classe externe.
- N'a pas de référence implicite vers l'instance englobante.
- Peut être instanciée sans instance externe.
- Peut contenir des variables membres non statiques.

```
class Outer {
    static int version = 1;

    static class Nested {
        void print() {
            System.out.println("Version: " + version); // OK: accessing static member
        }
    }
}

class Test {
    public static void main(String[] args) {
        Outer.Nested n = new Outer.Nested(); // No Outer instance required
        n.print();
    }
}
```

17.5.1.2 Erreurs Courantes

- Les static nested classes **ne peuvent pas accéder aux variables d'instance** :

```
class Outer {
    int x = 10;
    static class Nested {
        void test() {
            // System.out.println(x); // ❌ Compile error
        }
    }
}
```

17.5.2 Inner Classes (Non-Static Nested Classes)

Une **inner class** est associée à une instance de la classe externe et peut accéder à **tous les membres** de la classe externe, y compris ceux **private**.

17.5.2.1 Syntaxe et Règles d'Accès

- Déclarée sans `static`.
- Possède une référence implicite vers l'instance englobante.
- Peut accéder aux membres statiques et aux membres d'instance de la classe externe.
- Comme elle n'est pas statique, elle doit être créée via une instance de la classe englobante.

```java
class Outer {
    private int value = 100;

    class Inner {
        void print() {
            System.out.println("Value = " + value); // OK: accessing private
        }
    }

    void make() {
        Inner i = new Inner(); // OK inside the outer class
        i.print();
    }
}

class Test {
    public static void main(String[] args) {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner(); // MUST be created from an instance
        i.print();
    }
}
```

À l'intérieur d'une classe interne `non-static`, on peut faire référence à l'objet englobant en utilisant `OuterClass.this`.
L'expression `InnerClass.this`, qui est équivalente à `this`, fait référence à l'objet `Inner` courant.

- Exemple :

```java
class Outer {
    int x = 10;

    class Inner {
        int x = 20;

        void print() {
            System.out.println(x);              // 20 (Inner.this.x)
            System.out.println(this.x);         // 20
            System.out.println(Outer.this.x);   // 10
        }
    }
}
```

17.5.2.2 Erreurs Courantes

- Les inner classes **ne peuvent pas déclarer de membres statiques** sauf les **static final constants**.

```java
class Outer {
    class Inner {
        // static int x = 10;     // ❌ Compile error
        static final int OK = 10; // ✓ Allowed (constant)
    }
}
```

> **Warning**
>
> - Instancier une inner class SANS instance externe est illégal.

17.5.3 Classes Locales

Une **classe locale** est une classe imbriquée définie à l'intérieur d'un bloc — le plus souvent une méthode.

Elle n'a pas de modificateur d'accès et n'est visible qu'à l'intérieur du bloc où elle est déclarée.

17.5.3.1 Caractéristiques

- Déclarée à l'intérieur d'une méthode, d'un constructeur ou d'un initializer.
- Peut accéder aux membres de la classe externe.
- Peut accéder aux variables locales si elles sont **effectively final**.

- Ne peut pas déclarer de membres statiques (sauf static final constants).

```
class Outer {
    void compute() {
        int base = 5; // must be effectively final

        class Local {
            void show() {
                System.out.println(base); // OK
            }
        }

        new Local().show();
    }
}
```

Une classe locale, tout comme une classe interne membre, possède une référence implicite vers l'instance englobante via `OuterClass.this`.
Elle dispose également de `LocalClass.this`, équivalent à `this`, qui est valide à l'intérieur du corps de la classe locale.

- Exemple :

```
class Outer {
    int x = 10;

    void method() {
        class Local {
            void print() {
                System.out.println(Outer.this.x);  // ✓ valide

                System.out.println(Local.this);    // ✓ valide
            }
        }
    }
}
```

17.5.3.2 Erreurs Courantes

- `base` doit être effectively final ; le modifier casse la compilation.

```
void compute() {
    int base = 5;
    base++; // ❌ Now base is NOT effectively final
    class Local {}
}
```

17.5.4 Classes Anonymes

Une **classe anonyme** est une classe one-off créée inline, généralement pour implémenter une interface ou redéfinir une méthode sans nommer une nouvelle classe.

17.5.4.1 Syntaxe et Utilisation

- Créée via `new` + type + body.
- Ne peut pas avoir de constructeurs (pas de nom).
- Souvent utilisée pour event handling, callbacks, comparators.

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous running");
    }
};
```

17.5.4.2 Classe Anonyme qui Étend une Classe

```
Button b = new Button("Click");
b.onClick(new ClickHandler() {
    @Override
    public void handle() {
        System.out.println("Handled!");
    }
});
```

17.5.5 Comparaison des Types de Classes Imbriquées

Un tableau rapide qui résume tous les types de classes imbriquées.

Type	A une Instance Externe ?	Peut Accéder aux Membres d'Instance Externe ?	Peut Avoir des Membres Statiques ?	Usage Typique
Static Nested	Non	Non	Oui	Namespacing, helpers
Inner Class	Oui	Oui	Non (sauf constantes)	Comportement lié à l'objet
Local Class	Oui	Oui	Non	Classes temporaires avec scope
Anonymous Class	Oui	Oui	Non	Personnalisation inline

17.6 Imbrication des Interfaces en Java

En Java, une interface peut être déclarée à différents emplacements et suit des règles spécifiques concernant l'imbrication et les membres autorisés.

17.6.1 Où une Interface peut être Déclarée

Une interface peut être :

- **Top-level** (directement dans un package)
- **Interface membre imbriquée** (déclarée à l'intérieur d'une classe ou d'une autre interface)
- **Interface locale** ❌ (non autorisée)
- **Interface anonyme** ❌ (n'existe pas comme déclaration, seulement des implémentations anonymes)

En Java, il n'est **pas permis de déclarer une interface locale** (c'est-à-dire à l'intérieur d'une méthode ou d'un bloc).
Les interfaces peuvent être uniquement `top-level` ou `member`.

17.6.2 Interfaces Imbriquées

Une interface imbriquée peut être déclarée dans :

17.6.2.1 Interface Imbriquée dans une Classe

- Elle est implicitement `static`
- Elle ne peut pas être déclarée `non-static`
- Elle peut être déclarée `public`, `protected`, `private` ou `package-private`
- Exemple :

```java
class Outer {
    interface InnerInterface {
        void test();
    }
}
```

Le mot-clé `static` est implicite :

```java
class Outer {
    static interface InnerInterface {   // autorisé mais redondant
        void test();
    }
}
```

17.6.2.2 Interface Imbriquée dans une autre Interface

- Elle est implicitement `public` et `static`
- Elle ne peut pas être `private` ou `protected`

```java
interface A {
    interface B {
        void test();
    }
}
```

17.6.3 Règles dAccès

Une `interface imbriquée` :

- N'a pas de référence implicite à une instance de la classe englobante
- Ne peut pas accéder directement aux membres d'instance de la classe englobante
- **Peut accéder uniquement aux membres `static` de la classe englobante**

17.6.4 Types Imbriqués dans les Interfaces

Une interface peut contenir :

- Des classes imbriquées (implicitement `public static`)
- Des records imbriqués (implicitement `public static`)
- Des enums imbriqués (implicitement `public static`)
- D'autres interfaces imbriquées (implicitement `public static`)

17.6.5 Résumé Essentiel

- Les interfaces imbriquées sont toujours `static`
- Les interfaces locales n'existent pas
- Les champs sont toujours `public static final`
- Les méthodes sont implicitement `public abstract` (sauf default/static/private)
- Elles peuvent contenir d'autres types imbriqués

◀ 16. Héritage en Java | ▲ Index | 18. Generics en Java ▶

18. Generics en Java

Table des matières

Java `Generics` permettent de créer des classes, des interfaces et des méthodes qui travaillent avec des types spécifiés par l'utilisateur, en garantissant que seuls des objets du type correct sont utilisés.

Tous les contrôles de type sont effectués par le compilateur à compile-time.

Pendant la compilation, le compilateur vérifie les types puis supprime les informations génériques (processus identifié comme **type erasure**), en les remplaçant par les types réels ou par Object lorsque nécessaire.

Le bytecode résultant ne contient pas de generics: il contient seulement les types concrets et, si nécessaire, des casts insérés automatiquement par le compilateur.

De cette manière, les erreurs de type sont interceptées avant l'exécution, rendant le code plus sûr, lisible et réutilisable.

Les Generics s'appliquent à:

- `Classes`
- `Interfaces`
- `Méthodes` (méthodes génériques)
- `Constructeurs`

18.1 Bases des Types Génériques

Une classe ou interface générique introduit un ou plusieurs **paramètres de type**, encadrés par des chevrons.

```
class Box<T> {
    private T value;
    void set(T v) { value = v; }
    T get()       { return value; }
}

Box<String> b = new Box<>();

b.set("hello");

String x = b.get(); // aucun cast nécessaire
```

Plusieurs paramètres de type sont permis:

```
class Pair<K, V> {
    K key;
    V value;
}
```

18.2 Pourquoi les Generics Existent

```
List list = new ArrayList();          // pre-generics
list.add("hi");

Integer x = (Integer) list.get(0);    // ClassCastException à runtime
```

Avec les generics:

```
List<String> list = new ArrayList<>();
list.add("hi");

String x = list.get(0);               // type-safe, aucun cast
```

18.3 Méthodes Génériques

Une **méthode générique** introduit ses propres paramètres de type, indépendants de la classe.

```
class Util {

    static <T> T pick(T a, T b) { return a; }

}

String s = Util.<String>pick("A", "B"); // explicite
String t = Util.pick("A", "B");         // l'inférence fonctionne
```

18.4 Type Erasure

La `Type erasure` est le processus par lequel le compilateur Java supprime toutes les informations sur les types génériques avant de générer le bytecode.

Cela garantit la compatibilité avec les JVM précédentes à Java 5.

À `compile time`, les generics sont complètement contrôlés: bounds sur les types, variance, surcharge de méthodes génériques, etc.

Cependant, à runtime, toutes les informations génériques disparaissent.

18.4.1 Comment Fonctionne la Type Erasure

- Remplacer toutes les variables de type (comme `T`) par leur type erasure.
- Insérer des casts lorsque nécessaire.
- Supprimer tous les arguments de type générique (ex. `List<String>` → `List`).

18.4.2 Erasure des Paramètres de Type Sans Bound

Si une variable de type n'a pas de bound:

```
class Box<T> {
    T value;
    T get() { return value; }
}
```

L'erasure de `T` est `Object`.

```
class Box {
    Object value;
    Object get() { return value; }
}
```

18.4.3 Erasure des Paramètres de Type Avec Bound

Si le paramètre de type a un bound:

```
class TaskRunner<T extends Runnable> {

    void run(T task) { task.run(); }

}
```

Alors l'erasure de `T` est le premier bound trouvé par le compilateur: dans ce cas spécifique `Runnable`.

```
class TaskRunner {
    void run(Runnable task) { task.run(); }
}
```

18.4.4 Bounds Multiples: Le Premier Bound Détermine l'Erasure

Java permet des bounds multiples:

```
<T extends Runnable & Serializable & Cloneable>
```

> **Important**
>
> L'erasure de `T` est toujours le **premier bound**, qui doit être une classe ou une interface.

Puisque `Runnable` est le premier bound, le compilateur effectue l'erasure de `T` à `Runnable`.

- Exemple avec Bounds Multiples (Entièrement Développé)

```
public static <T extends Runnable & Serializable & Cloneable>
void runAll(List<T> list) {
    for (T t : list) {
        t.run();
    }
}
```

Version avec Erasure:

```
public static void runAll(List list) {
    for (Object obj : list) {
        Runnable t = (Runnable) obj;   // cast inséré par le compilateur
        t.run();
    }
}
```

Que se passe-t-il avec les autres bounds (Serializable, Cloneable)?

- Ils sont appliqués seulement à compile time.
- Ils n'apparaissent PAS dans le bytecode.
- Aucune interface supplémentaire n'est associée au type avec erasure.

18.4.5 Pourquoi Seulement le Premier Bound Devient le Type à Runtime?

Parce que la JVM doit opérer en utilisant un seul type de référence concret pour chaque variable ou paramètre.

Les instructions bytecode à runtime comme `invokevirtual` exigent une seule classe ou interface, pas un type composé comme "Runnable & Serializable & Cloneable".

> **Note**
>
> Java sélectionne le **premier bound** comme type à runtime, et utilise les bounds restants seulement pour la **validation à compile-time**.

18.4.6 Un Exemple Plus Complexe

```
interface A { void a(); }
interface B { void b(); }

class C implements A, B {
    public void a() {}
    public void b() {}
}

class Demo<T extends A & B> {
    void test(T value) {
        value.a();
        value.b();
    }
}
```

Version avec Erasure:

```
class Demo {
    void test(A value) {
        value.a();
        // value.b();   // ❌ non disponible après l'erasure: le type est A, pas B
    }
}
```

> **Note**
>
> Le compilateur peut insérer des casts supplémentaires ou des méthodes bridge dans des scénarios d'héritage plus complexes, mais l'erasure utilise toujours seulement le premier bound (A dans ce cas).

18.4.7 Redéfinition (Overriding) et Génériques

Lorsque les génériques interagissent avec l'héritage, deux règles fondamentales doivent être clairement comprises :

> **Important**
>
> **La redéfinition est vérifiée après l'effacement des types (type erasure).**
>
> **La compatibilité des types est vérifiée avant l'effacement.**

Ces deux étapes expliquent pourquoi certaines méthodes redéfinissent correctement, tandis que d'autres provoquent des erreurs de compilation.

18.4.7.1 Comment le compilateur valide une redéfinition

Lorsqu'une sous-classe déclare une méthode qui *pourrait* redéfinir une méthode de la superclasse, le compilateur effectue deux vérifications :

1. **Avant l'effacement**
    - La méthode doit être compatible avec celle de la classe parente :
        - Même nom
        - Même types de paramètres (y compris les arguments génériques)
        - Type de retour compatible (covariance autorisée)
2. **Après l'effacement**
    - Les signatures effacées doivent correspondre exactement.
        - Les deux conditions doivent être satisfaites.

18.4.7.2 Paramètres génériques et redéfinition

Les arguments de type générique font partie de la signature de la méthode **à la compilation**, mais disparaissent après l'effacement.

Par conséquent :

- Il est permis **d'effacer l'information générique dans la méthode redéfinie**
- Il est interdit **d'ajouter une nouvelle spécificité générique**
- Si les deux méthodes utilisent des types paramétrés, ils doivent correspondre exactement

18.4.7.3 Redéfinition valide — Suppression de la spécificité générique

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set data) {}   // ✓ autorisé (type brut)
}
```

Explication :

- Avant l'effacement : `Set` est compatible par affectation avec `Set<Integer>`
- Après l'effacement : les deux deviennent `Set`

✓ Redéfinition valide.

18.4.7.4 Redéfinition invalide — Ajout de spécificité générique

```
class Parent {
    void process(Set data) {}
}

class Child extends Parent {
    void process(Set<Integer> data) {}   // ❌ erreur de compilation
}
```

Explication :

- Avant l'effacement : `Set<Integer>` n'est PAS compatible par affectation avec `Set`
- Le compilateur rejette la méthode avant même d'appliquer l'effacement

18.4.7.5 Redéfinition valide — Paramétrage identique

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    @Override
    void process(Set<Integer> data) {}   // ✓ correspondance exacte
}
```

Les deux vérifications réussissent : - Compatible avant l'effacement - Identique après l'effacement

18.4.7.6 Redéfinition invalide — Changement d'argument générique

```
class Parent {
    void process(Set<Integer> data) {}
}

class Child extends Parent {
    void process(Set<String> data) {}   // ❌ erreur de compilation
}
```

Explication :

- Avant l'effacement : `Set<String>` n'est pas compatible avec `Set<Integer>`
- Après l'effacement : les deux deviennent `Set`
- Collision + incompatibilité → erreur de compilation

18.4.7.7 Pourquoi cette règle existe

Java doit garantir :

- **La sûreté des types à la compilation**
- **Le polymorphisme à l'exécution après effacement**

Comme les génériques disparaissent à l'exécution, la JVM ne voit que les signatures effacées.
Le compilateur doit donc assurer la compatibilité avant l'effacement et la cohérence après l'effacement.

18.4.7.8 Modèle mental

Considérez la redéfinition avec génériques comme une vérification en deux phases :

```
Phase 1 → Les types au niveau source sont-ils compatibles ?
Phase 2 → Les signatures effacées correspondent-elles ?
```

Si l'une des phases échoue → erreur de compilation.

18.4.7.9 Retours covariants et génériques

Une méthode redéfinie (c'est-à-dire une méthode déclarée dans une sous-classe) est autorisée à retourner un **sous-type** du type de retour déclaré dans la méthode redéfinie (c'est-à-dire la méthode de la superclasse).

Ceci est connu sous le nom de **règle des retours covariants**.

La première étape lors de la validation d'une redéfinition consiste donc à :

- Vérifier si le type de retour de la méthode redéfinissante est un sous-type du type de retour déclaré dans la superclasse.

> **Important**
>
> - Si la méthode redéfinie retourne `List`, la méthode redéfinissante peut retourner `ArrayList`.
> - Elle ne peut **pas** retourner `Object`, car `Object` est un supertype et non un sous-type.

Lorsque les génériques sont impliqués, la validation du type de retour devient plus subtile.

Il faut alors évaluer les relations de sous-typage en utilisant les règles de hiérarchie des types génériques.

Supposons que `S` soit un sous-type de `T`.

Il existe deux hiérarchies génériques importantes à retenir.

Hiérarchie 1 (wildcards bornés supérieurs) :

`A<S>` est un sous-type de `A<? extends S>` qui est lui-même un sous-type de `A<? extends T>`

- Exemple :

Puisque `Integer` est un sous-type de `Number` :

- `List<Integer> <<< List<? extends Integer>`
- `List<? extends Integer> <<< List<? extends Number>`

Donc, si une méthode redéfinie retourne :

`List<? extends Integer>`

la méthode redéfinissante peut retourner :

- `List<Integer>`

mais ne peut **pas** retourner :

- `List<Number>`
- `List<? extends Number>`

Hiérarchie 2 (wildcards bornés inférieurs) :

`A<T>` est un sous-type de `A<? super T>` qui est lui-même un sous-type de `A<? super S>`

- Exemple :
- `List<Number> <<< List<? super Number>`
- `List<? super Number> <<< List<? super Integer>`

Donc, si une méthode redéfinie retourne :

`List<? super Number>`

la méthode redéfinissante peut retourner :

- `List<Number>`

mais ne peut **pas** retourner :

- `List<Integer>`
- `List<? super Integer>`

Point crucial à retenir :

Même si `Integer` est un sous-type de `Number`,
`List<Integer>` n'est **pas** un sous-type de `List<Number>`.

Les types génériques en Java sont invariants, sauf en présence de wildcards.

Ces règles expliquent pourquoi certaines méthodes qui semblent compatibles sont rejetées par le compilateur.
La spécificité générique doit respecter les hiérarchies formelles de sous-typage avant que la validation de redéfinition ne passe aux vérifications basées sur l'effacement (voir 18.4.7.10).

18.4.7.10 Règles récapitulatives

- La redéfinition est validée **après l'effacement**
- La compatibilité est validée **avant l'effacement**
- Il est possible d'effacer l'information générique dans la sous-classe
- Il est interdit d'introduire une nouvelle spécificité générique
- Si les deux méthodes sont paramétrées, les arguments doivent correspondre exactement
- Après l'effacement, les signatures doivent être identiques
- Les types de retour covariants exigent que le type de retour de la méthode redéfinissante soit un véritable sous-type.
- Avec les génériques, les relations de sous-typage doivent respecter les règles de hiérarchie des wildcards.
- Les relations logiques apparentes entre arguments de type (par exemple `Integer` et `Number`) ne se traduisent pas automatiquement en relations de sous-typage entre types paramétrés.

Cela explique pourquoi certaines méthodes qui *semblent* être des surcharges sont rejetées :
après l'effacement, elles entrent en collision, et si elles ne constituent pas une redéfinition valide, le compilateur les bloque.

18.4.8 Surcharge d'une Méthode Générique — Pourquoi Certaines Surcharges Sont Impossibles

Quand Java compile du code générique, il applique la type erasure: les paramètres de type comme T sont supprimés, et le compilateur les remplace par leur type erasure (habituellement Object ou le premier bound).

Pour cette raison, deux méthodes qui semblent différentes au niveau source peuvent devenir identiques après l'erasure.

Si les `signature` avec erasure sont les mêmes, Java ne peut pas les distinguer, donc le code ne compile pas.

- Exemple: Deux Méthodes qui S'effondrent sur la Même `Signature`

```
public class Demo {
    public void testInput(List<Object> inputParam) {}

    // public void testInput(List<String> inputParam) {}   // ❌ Erreur de compilation: après
}
```

Explication

```
List<Object> et List<String> sont tous deux effacés en List.
```

À runtime les deux méthodes apparaîtraient comme:

```
void testInput(List inputParam)
```

Java ne permet pas deux méthodes avec des signatures identiques dans la même classe, donc la surcharge est rejetée à compile time.

18.4.9 Surcharge d'une Méthode Générique Héritée d'une Classe Parent

La même règle s'applique quand une subclass tente d'introduire une méthode qui, après erasure, a la même signature qu'une dans la superclass.

```
public class SubDemo extends Demo {
    public void testInput(List<Integer> inputParam) {}
    // ❌ Erreur de compilation: erasure → testInput(List), identique au parent
}
```

Encore une fois, le compilateur rejette la surcharge parce que les signatures avec erasure entrent en collision.

Quand la Surcharge Fonctionne

L'erasure supprime seulement les paramètres génériques, pas la classe réelle utilisée comme paramètre de méthode.

Donc, si deux paramètres diffèrent dans le type raw (non générique), la surcharge est légale.

```
public class Demo {
    public void testInput(List<Object> inputParam) {}
    public void testInput(ArrayList<String> inputParam) {}  // ✓ Compile
}
```

Pourquoi ça fonctionne

Même si `ArrayList<String>` devient `ArrayList`, et `List<Object>` devient `List`, ce sont des classes différentes (ArrayList vs List), donc les signatures restent distinctes:

```
void testInput(List inputParam)
void testInput(ArrayList inputParam)
```

Aucune collision → surcharge légale.

18.4.10 Retourner des Types Génériques — Règles et Restrictions

Quand on retourne une valeur depuis une méthode, Java suit une règle rigide:

Le type de retour d'une méthode en overriding doit être un sous-type du type de retour du parent, et tout argument générique doit rester type-compatible (même s'il est effacé à runtime).

Cela confond souvent les programmeurs, parce que les generics sur les types de retour causent des conflits similaires à ceux des paramètres.

Points Clés:

- La **covariance du type de retour s'applique seulement au type raw**, pas aux arguments génériques.
- Les arguments génériques doivent rester compatibles après l'erasure (ils doivent coïncider).
- **Deux méthodes ne peuvent pas différer seulement par le paramètre générique dans le type de retour**.

Exemple: substitution Illégale du Type de Retour à Cause d'Incompatibilité Générique

```
class A {
    List<String> getData() { return null; }
}

class B extends A {
    // List<Integer> n'est pas un type de retour covariant de List<String>
    // ❌ Erreur de compilation
    List<Integer> getData() { return null; }
}
```

Explication:

Même si les generics sont effacés, Java impose quand même la type safety au niveau source:

```
List<Integer> n'est pas un sous-type de List<String>.
```

Les deux deviennent List, mais Java rejette l'override qui casse la compatibilité de type.

- Exemple: Type de Retour Covariant Légal

```
class A {
    Collection<String> getData() { return null; }
}

class B extends A {
    List<String> getData() { return null; }  // ✓ List est un sous-type de Collection
}
```

Ceci est permis parce que:

- Les types raw sont covariants (List étend Collection).
- Les arguments génériques coïncident (String vs String).
- Exemple: Surcharge Illégale Basée Seulement sur le Type de Retour

```
class Demo {
    List<String> getList() { return null; }

    // List<Integer> getList() { return null; }
    // ❌ Erreur de compilation: le type de retour seul ne distingue pas les méthodes
}
```

Java n'utilise pas le type de retour pour distinguer les méthodes en surcharge.

18.4.11 Récapitulatif des Règles d'Erasure

- `T sans bound` → erasure à Object.
- `T extends X` → erasure à X.
- `T extends X & Y & Z` → erasure à X.
- Tous les paramètres génériques sont effacés dans les signatures des méthodes.
- Des casts sont insérés pour préserver la typisation à compile-time.
- Des méthodes bridge peuvent être générées pour préserver le polymorphisme.

18.5 Bounds sur les Paramètres de Type

Cette section introduit les **bornes sur les paramètres de type et les jokers (wildcards)** dans les génériques Java.
Les bornes restreignent l'ensemble des types pouvant être utilisés avec un paramètre de type générique ou un joker.

Elles sont utilisées pour imposer des **contraintes de type** et pour exprimer des relations entre types dans du code générique.

Les bornes apparaissent sous deux formes principales :

- **Bornes sur les paramètres de type** utilisant `extends`
- **Bornes sur les jokers** utilisant `?`, `? extends` et `? super`

Ces mécanismes permettent aux API génériques de spécifier quels types sont acceptables et quelles opérations sont sûres du point de vue du typage.

Règles

- `T extends Type` → le paramètre de type doit être `Type` ou une sous-classe.
- `T extends Classe & Interface1 & Interface2` → plusieurs bornes sont autorisées.
- Dans des bornes multiples, la classe doit apparaître en premier.

- `?` représente un type inconnu.
- `? extends Type` → accepte des types qui sont `Type` ou des sous-classes.
- `? super Type` → accepte des types qui sont `Type` ou des superclasses.
- `? extends` permet la **lecture (extraction)** mais interdit l'insertion.
- `? super` permet l'**écriture (insertion)** mais la lecture retourne `Object`.

Tableau récapitulatif

Syntaxe	Signification	Compatibilité d'affectation	Lecture	Écriture
`<T extends Number>`	Le paramètre de type doit être `Number` ou une sous-classe	Borne de déclaration générique	`T`	`T`
`<T extends Classe & Interface>`	Bornes multiples	Borne de déclaration générique	`T`	`T`
`List<?>`	Type d'élément inconnu	Tout `List<T>`	`Object`	❌
`List<? extends Number>`	Sous-type inconnu de `Number`	`List<Integer>`, `List<Double>`, etc.	`Number`	❌
`List<? super Integer>`	`Integer` ou supertype	`List<Integer>`, `List<Number>`, `List<Object>`	`Object`	`Integer`

18.5.1 Upper Bounds: extends

`<T extends Number>` signifie que **T doit être Number ou une sous-classe**.

```
class Stats<T extends Number> {
    T num;
    Stats(T num) { this.num = num; }
}
```

18.5.2 Bounds Multiples

Syntaxe : `T extends Classe & Interface1 & Interface2 ...`

La classe doit apparaître en premier.

```
class C<T extends Number & Comparable<T>> { }
```

18.5.3 Wildcard: `?`, `? extends`, `? super`

18.5.3.1 Wildcard Non Limitée `?`

À utiliser lorsque l'on veut accepter une liste de type inconnu :

```
void printAll(List<?> list) { ... }
```

18.5.3.2 Wildcard avec Upper Bound `? extends`

```
List<? extends Number> nums = List.of(1, 2, 3);
Number n = nums.get(0);    // OK
// nums.add(5);            // ❌ impossible d'ajouter : sécurité de type
```

> **Vous ne pouvez pas ajouter d'éléments (sauf null) à ? extends** car vous ne connaissez pas le sous-type exact.

18.5.3.3 Wildcard avec Lower Bound `? super`

`<? super Integer>` signifie que **le type doit être Integer ou une superclasse de Integer**.

```
List<? super Integer> list = new ArrayList<Number>();
list.add(10);    // OK
Object o = list.get(0); // retourne Object (supertype commun le plus bas)
```

> **Important**
>
> - `Super` accepte **l'insertion**
> - `extends` accepte **l'extraction**.

18.6 Generics et Héritage

> I generics ne participent PAS à l'héritage.
> Un `List<String>` n'est pas un sous-type de `List<Object>`; les types paramétrés sont invariants.

```
List<String> ls = new ArrayList<>();
List<Object> lo = ls;      // ❌ erreur de compilation
```

Au contraire:

```
List<? extends Object> ok = ls;   // fonctionne
```

18.7 Type Inference (Opérateur Diamond)

```
Map<String, List<Integer>> map = new HashMap<>();
```

Le compilateur déduit les arguments génériques à partir de l'affectation.

18.8 Raw Types (Compatibilité Legacy)

Un **raw type** désactive les generics, réintroduisant des comportements non sûrs.

```
List raw = new ArrayList();
raw.add("x");
raw.add(10);   // permis, mais non sûr
```

> Les raw types devraient être évités.

18.9 Tableaux Génériques (Non Autorisés)

Tu ne peux pas créer des tableaux de types paramétrés:

```
List<String>[] arr = new List<String>[10];   // ❌ erreur de compilation
```

Parce que les tableaux appliquent la type safety à runtime tandis que les generics se basent seulement sur des contrôles à compile-time.

18.10 Bounded Type Inference

```
static <T extends Number> T identity(T x) { return x; }

int v = identity(10);   // OK
// String s = identity("x"); // ❌ n'est pas un Number
```

18.11 Wildcard vs Paramètres de Type

Utilise les **wildcards** quand tu as besoin de flexibilité dans les paramètres. Utilise les **paramètres de type** quand la méthode doit retourner ou maintenir des informations de type.

- Exemple — wildcard trop faible:

```
List<?> copy(List<?> list) {
   return list;  // perd des informations de type
}
```

Mieux:

```
<T> List<T> copy(List<T> list) {
    return list;
}
```

18.12 Règle PECS (Producer Extends, Consumer Super)

Utilise **? extends** quand le paramètre **produit** des valeurs. Utilise **? super** quand le paramètre **consomme** des valeurs.

```
List<? extends Number> listExtends = List.of(1, 2, 3);
List<? super Integer> listSuper = new ArrayList<Number>();

// ? extends → lecture sûre
Number n = listExtends.get(0);

// ? super → écriture sûre
listSuper.add(10);
```

18.13 Pièges Communs

- Trier des listes avec wildcard: `List<? extends Number>` ne peut pas accepter d'insertions.
- Mal comprendre que `List<Object>` N'EST PAS un supertype de List.
- Oublier que les tableaux génériques sont illégaux.
- Penser que les types génériques sont préservés à runtime (ils sont effacés).
- Essayer de surcharger des méthodes en utilisant seulement des paramètres de type différents.

18.14 Tableau Récapitulatif des Wildcards

Syntaxe	Signification
`?`	type inconnu (lecture seule sauf méthodes Object)
`? extends T`	lire T en sécurité, on ne peut pas ajouter (sauf null)
`? super T`	on peut ajouter T, la lecture retourne Object

18.15 Récapitulatif des Concepts

```
Generics = type safety à compile-time
Bounds = limitent les types légaux
Wildcard = flexibilité dans les paramètres
Type Inference = le compilateur déduit les types
Type Erasure = les generics disparaissent à runtime
Bridge Methods = maintiennent le polymorphisme
```

18.16 Exemple Complet

```java
class Repository<T extends Number> {
    private final List<T> store = new ArrayList<>();

    void add(T value) { store.add(value); }

    T first() { return store.isEmpty() ? null : store.get(0); }

    // méthode générique avec wildcard
    static double sum(List<? extends Number> list) {
        double total = 0;
        for (Number n : list) total += n.doubleValue();
        return total;
    }
}
```

◀ 17. Au-delà des Classes | ▲ Index | 19. Exceptions et Gestion des Erreurs ▶

19. Exceptions et Gestion des Erreurs

Table des matières

Les `Exceptions` constituent le mécanisme structuré de Java pour gérer les conditions anormales à runtime. Elles permettent de séparer le flux normal d'exécution de la logique de gestion des erreurs, améliorant la robustesse, la lisibilité et l'exactitude du programme.

19.1 Hiérarchie et types dexceptions

Toutes les exceptions dérivent de `Throwable`. La hiérarchie définit quelles conditions sont récupérables, lesquelles doivent être déclarées, et lesquelles représentent des défaillances système fatales.

```
java.lang.Object
└── java.lang.Throwable
    ├── java.lang.Error
    └── java.lang.Exception
        └── java.lang.RuntimeException
```

19.1.1 Throwable

- Classe de base pour toutes les erreurs et exceptions
- Fournit message, cause et stack trace
- Seuls `Throwable` et ses sous-classes peuvent être lancés ou capturés

19.1.2 Error (unchecked)

- Représente des problèmes graves de la JVM ou du système
- Non destiné à être capturé ou géré
- Exemples: `OutOfMemoryError`, `StackOverflowError`

> **Note**
>
> Les erreurs indiquent des conditions dont l'application ne peut généralement pas se remettre.

19.1.3 Exceptions Checked (`Exception`)

- Sous-classes de `Exception` **à lexclusion** de `RuntimeException`
- Représentent des conditions que l'application peut vouloir gérer
- Doivent être soit **capturées** soit **déclarées**
- Exemples: `IOException`, `SQLException`

19.1.4 Exceptions Unchecked (`RuntimeException`)

- Sous-classes de `RuntimeException`
- Ne doivent pas obligatoirement être déclarées ou capturées
- Représentent généralement des erreurs de programmation
- Exemples: `NullPointerException`, `IllegalArgumentException`

19.2 Déclarer et lancer des exceptions

19.2.1 Déclarer des exceptions avec throws

Une méthode déclare les exceptions checked avec la clause `throws`. Cela fait partie du contrat API de la méthode.

```
void readFile(Path p) throws IOException {
    Files.readString(p);
}
```

> **Note**
>
> - Seules les **exceptions checked** doivent être déclarées.
> - Les exceptions unchecked peuvent l'être, mais sont généralement omises.

19.2.2 Lancer des exceptions

Les exceptions sont créées avec `new` et lancées explicitement avec `throw`.

```
if (value < 0) {
    throw new IllegalArgumentException("value must be >= 0");
}
```

- `throw` lance exactement une instance dexception
- `throws` déclare les exceptions possibles dans la signature de la méthode

19.3 Redéfinition de méthodes et règles sur les exceptions

Lors de la redéfinition dune méthode, les règles sur les exceptions sont strictement appliquées :

- Une méthode redéfinie peut lancer **moins** d'exceptions checked ou des exceptions plus **spécifiques**
- Elle peut lancer n'importe quelles exceptions unchecked
- Elle ne peut lancer **aucune nouvelle** exception checked plus large

```java
class Parent {
    void work() throws IOException {}
}

class Child extends Parent {
    @Override
    void work() throws FileNotFoundException {} // OK
}
```

> **Note**
>
> Modifier uniquement les exceptions unchecked ne viole jamais le contrat de redéfinition.

> **Important**
>
> Rappel : les `constructeurs` suivent une règle différente.
>
> Un `Constructeur` doit déclarer toutes les exceptions checked déclarées dans le constructeur de base (ou les superclasses de ces exceptions checked).
>
> Il peut également déclarer des exceptions checked supplémentaires. Ce comportement est l'opposé de celui de l'overriding de méthodes.
>
> Une méthode qui override ne peut pas lancer d'exception checked autre que celles déclarées par la méthode overridée. Elle ne peut lancer que des sous-classes de ces exceptions.

19.4 Gestion des exceptions: try, catch, finally

19.4.1 Syntaxe de base try-catch

```java
try {
    riskyOperation();
} catch (IOException e) {
    handle(e);
}
```

- Un bloc `try` doit être suivi d'au moins un `catch` ou d'un `finally`
- Les `catch` sont évalués de haut en bas

19.4.2 Plusieurs blocs catch

```java
try {
    process();
} catch (FileNotFoundException e) {
    recover();
} catch (IOException e) {
    log();
}
```

> **Note**
>
> Les exceptions plus spécifiques doivent précéder les plus générales, sinon la compilation échoue. Si un `catch` pour une superclasse précède celui d'une sous-classe, ce dernier devient inatteignable.

19.4.3 Multi-catch Java-7

```java
try {
    process();
} catch (IOException | SQLException e) {
    log(e);
}
```

- Les types d'exception doivent être non liés (pas parent/enfant)

- La variable capturée est implicitement `final`

19.4.4 Bloc finally

Le bloc `finally` s'exécute qu'il y ait exception ou non, sauf en cas d'arrêt extrême de la JVM.

```
try {
    open();
} finally {
    close();
}
```

- Utilisé pour la logique de nettoyage
- S'exécute même si `return` est utilisé dans try ou catch

> **Note**
>
> Un bloc `finally` peut écraser une valeur de retour ou avaler une exception. Cela est déconseillé car cela complique le flux de contrôle.

> **Important**
>
> Lorsquun bloc `catch` et un bloc `finally` lancent tous deux une exception, lexception lancée dans le bloc `finally` est celle qui est propagée par la méthode.
>
> Lexception lancée dans le bloc `catch` est perdue et **nest pas** ajoutée à la liste des exceptions supprimées.

```
try {
    throw new RuntimeException("try");
} catch (RuntimeException e) {
    throw new RuntimeException("catch");
} finally {
    throw new RuntimeException("finally");
}
```

Dans ce cas, seule lexception `"finally"` est lancée.

19.5 Gestion automatique des ressources try-with-resources

Le `try-with-resources` permet la fermeture automatique des ressources implémentant `AutoCloseable`. Il élimine le besoin d'un bloc `finally` explicite dans la plupart des cas.

19.5.1 Syntaxe de base

```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
}
```

- Les ressources sont fermées automatiquement
- La fermeture a lieu même en cas d'exception
- Les ressources sont fermées avant l'exécution de tout bloc `catch` ou `finally`.

```
try (Resource a = new Resource()) {
    a.read();
} finally {
    a.close();  // ❌ Compile-time error: a est hors de portée ici
}
```

19.5.2 Déclarer plusieurs ressources

```
try (InputStream in = Files.newInputStream(p);
        OutputStream out = Files.newOutputStream(q)) {
    in.transferTo(out);
}
```

- Les ressources sont fermées en **ordre inverse** de déclaration

19.5.3 Portée des ressources

- Les ressources sont visibles uniquement dans le bloc `try`
- Elles sont implicitement `final`
- Depuis Java 9, on peut déclarer les ressources avant le try-with-resources si elles sont `final` ou effectivement finales

```
final var firstWriter = Files.newBufferedWriter(filePath);

try (firstWriter; var secondWriter = Files.newBufferedWriter(filePath)) {
    // CODE
}
```

> **Note**
>
> Tenter de réaffecter une variable ressource provoque une erreur de compilation.

```
Resource a = new Resource();
try(a){ // since Java 9
  ...
}finally{
   a.close(); // ce code compile, mais la ressource référencée par la référence `a` a déjà été
}
```

19.6 Exceptions supprimées

Lorsque le bloc `try` et la méthode `close()` d'une ressource lancent tous deux une exception, Java conserve l'exception principale et **supprime** les autres.

```
try (BadResource r = new BadResource()) {
    throw new RuntimeException("main");
}
```

Si `close()` lance aussi une exception, elle devient **supprimée**.

```
catch (Exception e) {
    for (Throwable t : e.getSuppressed()) {
        System.out.println(t);
    }
}
```

- L'exeption principale est lancée
- Les exceptions secondaires sont accessibles via `getSuppressed()`

> **Important**
>
> Les exceptions supprimées sont générées uniquement par le bloc `finally` **implicite** créé par le `try-with-resources`.
>
> En revanche, les exceptions lancées dans un bloc `finally` **explicite** ne sont pas supprimées : elles remplacent toute exception précédente et deviennent la seule exception propagée.

19.7 Résumé des exceptions

- Les exceptions checked doivent être capturées ou déclarées
- Les méthodes redéfinies ne peuvent pas élargir les exceptions checked
- Utiliser multi-catch pour une logique de gestion commune
- Préférer try-with-resources au nettoyage via finally
- Les ressources se ferment en ordre inverse
- Les exceptions supprimées préservent le contexte complet de défaillance

---

◀ 18. Generics en Java | ▲ Index | 20. Programmation Fonctionnelle en Java ▶

Module 05

Functional Programming

20. Programmation Fonctionnelle en Java

Table des matières

La `programmation fonctionnelle` est un paradigme qui se concentre sur ce qui doit être fait plutôt que sur la manière de le faire.

À partir de Java 8, le langage a ajouté plusieurs fonctionnalités qui permettent un style "fonctionnel" : `lambda expressions`, `functional interfaces` et `method references`.

Ces fonctionnalités permettent d'écrire du code plus expressif, concis et réutilisable, en particulier lorsqu'on travaille avec des collections, des API de concurrence et des systèmes event-driven.

20.1 Interfaces Fonctionnelles

En Java, une **interface fonctionnelle** est une interface qui contient **exactement une** méthode abstraite.

Les interfaces fonctionnelles permettent les **Lambda Expressions** et les **Method References**, et constituent le cœur du modèle de programmation fonctionnelle de Java.

> **Note**
>
> Java considère automatiquement comme interface fonctionnelle toute interface ayant une seule méthode abstraite. L'annotation `@FunctionalInterface` est optionnelle mais recommandée.

20.1.1 Règles pour les Interfaces Fonctionnelles

- **Exactement une méthode abstraite** (SAM = Single Abstract Method).

- Une interface peut déclarer un nombre quelconque de méthodes **default**, **static** ou **private**.
- Elle peut redéfinir des méthodes de `Object` (`toString()`, `equals(Object)`, `hashCode()`) sans affecter le décompte SAM.
- La méthode fonctionnelle peut provenir d'une **super-interface**.

Exemple :

```
@FunctionalInterface
interface Adder {
    int add(int a, int b);   // single abstract method
    static void info() {}
    default void log() {}
}
```

20.1.2 Interfaces Fonctionnelles Courantes (java.util.function)

Ci-dessous, un résumé des interfaces fonctionnelles les plus importantes.

Functional Interface	Returns	Method	Parameters
`Supplier<T>`	T	get()	0
`Consumer<T>`	void	accept(T)	1
`BiConsumer<T,U>`	void	accept(T,U)	2
`Function<T,R>`	R	apply(T)	1
`BiFunction<T,U,R>`	R	apply(T,U)	2
`UnaryOperator<T>`	T	apply(T)	1 (mêmes types)
`BinaryOperator<T>`	T	apply(T,T)	2 (mêmes types)
`Predicate<T>`	boolean	test(T)	1
`BiPredicate<T,U>`	boolean	test(T,U)	2

- Exemples

```
Supplier<String> sup = () -> "Hello!";

Consumer<String> printer = s -> System.out.println(s);

Function<String, Integer> length = s -> s.length();

UnaryOperator<Integer> square = x -> x * x;

Predicate<Integer> positive = x -> x > 0;
```

20.1.3 Méthodes de Commodité sur les Interfaces Fonctionnelles

De nombreuses interfaces fonctionnelles proposent des méthodes utilitaires permettant l'enchaînement et la composition.

Interface	Method	Description
Function	andThen()	applique la fonction, puis l'autre spécifiée dans cette méthode additionnelle
Function	compose()	applique la fonction spécifiée dans cette méthode additionnelle, puis la fonction
Function	identity()	renvoie une fonction x -> x
Predicate	and()	ET logique
Predicate	or()	OU logique
Predicate	negate()	NON logique
Consumer	andThen()	chaîne des consumers
BinaryOperator	minBy()	minimum basé sur un comparator
BinaryOperator	maxBy()	maximum basé sur un comparator

- Exemples

```
Function<Integer, Integer> times2 = x -> x * 2;
Function<Integer, Integer> plus3  = x -> x + 3;

var result1 = times2.andThen(plus3).apply(5);    // (5*2)+3 = 13
var result2 = times2.compose(plus3).apply(5);    // (5+3)*2 = 16

Predicate<String> longString = s -> s.length() > 5;
Predicate<String> startsWithA = s -> s.startsWith("A");

boolean ok = longString.and(startsWithA).test("Amazing");  // true
```

20.1.4 Interfaces Fonctionnelles Primitives

Java fournit des versions spécialisées des interfaces fonctionnelles pour les types primitifs afin d'éviter le coût du boxing/unboxing.

Functional Interface	Return Type	Single Abstract Method	# Parameters
IntSupplier	int	getAsInt()	0
LongSupplier	long	getAsLong()	0
DoubleSupplier	double	getAsDouble()	0
BooleanSupplier	boolean	getAsBoolean()	0
IntConsumer	void	accept(int)	1 (int)
LongConsumer	void	accept(long)	1 (long)
DoubleConsumer	void	accept(double)	1 (double)
IntPredicate	boolean	test(int)	1 (int)
LongPredicate	boolean	test(long)	1 (long)
DoublePredicate	boolean	test(double)	1 (double)
IntUnaryOperator	int	applyAsInt(int)	1 (int)
LongUnaryOperator	long	applyAsLong(long)	1 (long)
DoubleUnaryOperator	double	applyAsDouble(double)	1 (double)
IntBinaryOperator	int	applyAsInt(int, int)	2 (int,int)
LongBinaryOperator	long	applyAsLong(long, long)	2 (long,long)
DoubleBinaryOperator	double	applyAsDouble(double,double)	2
IntFunction	R	apply(int)	1 (int)
LongFunction	R	apply(long)	1 (long)
DoubleFunction	R	apply(double)	1 (double)
ToIntFunction	int	applyAsInt(T)	1 (T)
ToLongFunction	long	applyAsLong(T)	1 (T)
ToDoubleFunction	double	applyAsDouble(T)	1 (T)
ToIntBiFunction<T,U>	int	applyAsInt(T,U)	2 (T,U)
ToLongBiFunction<T,U>	long	applyAsLong(T,U)	2 (T,U)
ToDoubleBiFunction<T,U>	double	applyAsDouble(T,U)	2 (T,U)
ObjIntConsumer	void	accept(T,int)	2 (T,int)
ObjLongConsumer	void	accept(T,long)	2 (T,long)
ObjDoubleConsumer	void	accept(T,double)	2 (T,double)
DoubleToIntFunction	int	applyAsInt(double)	1
DoubleToLongFunction	long	applyAsLong(double)	1
IntToDoubleFunction	double	applyAsDouble(int)	1
IntToLongFunction	long	applyAsLong(int)	1
LongToDoubleFunction	double	applyAsDouble(long)	1
LongToIntFunction	int	applyAsInt(long)	1

- Exemple

```
IntSupplier dice = () -> (int)(Math.random() * 6) + 1;

IntPredicate even = x -> x % 2 == 0;

IntUnaryOperator doubleIt = x -> x * 2;
```

20.1.5 Résumé

- Les interfaces fonctionnelles contiennent exactement une méthode abstraite (SAM).
- Elles sont le support des Lambdas et des Method References.
- Java propose de nombreuses FI intégrées dans `java.util.function`.
- Les variantes primitives améliorent les performances en supprimant le boxing.

20.2 Expressions Lambda

Une expression lambda est une manière compacte d'écrire une fonction.

Les expressions lambda offrent une façon concise de définir des implémentations d'interfaces fonctionnelles.

Une lambda est essentiellement un petit bloc de code qui prend des paramètres et renvoie une valeur, sans nécessiter une déclaration complète de méthode.

Elles représentent le comportement comme une donnée et constituent un élément clé du modèle de programmation fonctionnelle en Java.

20.2.1 Syntaxe des Expressions Lambda

La syntaxe générale est :

`(parameters) -> expression`
ou
`(parameters) -> { statements }`

> **Important**
>
> Une expression lambda n'introduit pas un nouveau scope pour les variables. Par conséquent, les noms de variables déjà présents dans le contexte environnant ne peuvent pas être redéclarés comme paramètres de l'expression lambda.

20.2.2 Exemples de Syntaxe Lambda

Zéro paramètre

```
Runnable r = () -> System.out.println("Hello");
```

Un paramètre (parenthèses optionnelles)

```
Consumer<String> c = s -> System.out.println(s);
```

Plusieurs paramètres

```
BinaryOperator<Integer> add = (a, b) -> a + b;
```

Avec un corps en bloc

```
Function<Integer, String> f = (x) -> {
    int doubled = x * 2;
    return "Value: " + doubled;
};
```

20.2.3 Règles pour les Expressions Lambda

- Les types des paramètres peuvent être omis (inférence de type).
- Si un paramètre a un type, alors **tous** les paramètres doivent spécifier un type.
- Un seul paramètre ne nécessite pas de parenthèses.
- Plusieurs paramètres nécessitent des parenthèses.
- Si le corps est une seule expression (sans `{ }`), `return` est interdit ; l'expression elle-même est la valeur de retour.
- Si le corps utilise `{ }` (un bloc), `return` doit apparaître si une valeur est renvoyée.
- Les expressions lambda ne peuvent être assignées qu'à des interfaces fonctionnelles (types SAM).

20.2.4 Inférence de Type

Le compilateur déduit le type de la lambda à partir du contexte de l'interface fonctionnelle cible.

```
Predicate<String> p = s -> s.isEmpty();  // s déduit comme String
```

Si le compilateur ne peut pas déduire le type, il faut le préciser explicitement.

```
BiFunction<Integer, Integer, Integer> f = (Integer a, Integer b) -> a * b;
```

20.2.5 Restrictions dans les Corps des Lambda

Les lambdas ne peuvent capturer que des variables locales `final` ou effectively final (non réassignées).

```
int x = 10;
Runnable r = () -> {
    // x++;   // ❌ erreur de compilation — x doit être effectively final
    System.out.println(x);
};
```

Elles peuvent en revanche modifier l'état d'un objet (seules les références doivent être effectively final).

```
var list = new ArrayList<>();
Runnable r2 = () -> list.add("OK");  // autorisé
```

20.2.6 Règles de Type de Retour

Si le corps est une expression : l'expression est la valeur de retour.

```
Function<Integer, Integer> f = x -> x * 2;
```

Si le corps est un bloc : il faut inclure `return`.

```
Function<Integer, Integer> g = x -> {
    return x * 2;
};
```

20.2.7 Lambdas vs Classes Anonymes

- Les lambdas ne créent PAS une nouvelle portée : elles partagent la portée englobante.
- `this` dans une lambda fait référence à l'objet englobant, pas à la lambda.

```
class Test {
    void run() {
        Runnable r = () -> System.out.println(this.toString());
    }
}
```

Dans une classe anonyme, `this` fait référence à l'instance de la classe anonyme.

20.2.8 Erreurs Courantes

Types de retour incohérents

```
x -> { if (x > 0) return 1; }  // ❌ manque un return pour le cas négatif
```

Mélanger paramètres typés et non typés

```
(a, int b) -> a + b   // ❌ illégal
```

Renvoyer une valeur pour une lambda ciblant void

```
Runnable r = () -> 5;  // ❌ Runnable.run() retourne void
```

Résolution d'overload ambiguë

```
void m(IntFunction<Integer> f) {}
void m(Function<Integer, Integer> f) {}

m(x -> x + 1);  // ❌ ambigu
```

20.3 Références de Méthodes

Les références de méthodes (method references) fournissent une syntaxe abrégée pour utiliser une méthode existante comme implémentation d'une interface fonctionnelle.

Elles sont équivalentes aux expressions lambda, mais plus concises, plus lisibles, et souvent préférées lorsque la méthode cible existe déjà.

Il existe quatre catégories de références de méthodes en Java :

- Référence à une méthode statique (`ClassName::staticMethod`)
- Référence à une méthode d'instance d'un objet particulier (`instance::method`)
- Référence à une méthode d'instance d'un objet arbitraire d'un type donné (`ClassName::instanceMethod`)
- Référence à un constructeur (`ClassName::new`)

20.3.1 Référence à une Méthode Statique

Une référence à méthode statique remplace une lambda qui appelle une méthode statique.

```
class Utils {
    static int square(int x) { return x * x; }
}

Function<Integer, Integer> f1 = x -> Utils.square(x);
Function<Integer, Integer> f2 = Utils::square;  // method reference
```

`f1` et `f2` se comportent de manière identique.

20.3.2 Référence à une Méthode d'Instance d'un Objet Particulier

Utilisée lorsque vous avez déjà une instance d'objet et que vous voulez référencer l'une de ses méthodes.

```
String prefix = "Hello, ";

UnaryOperator<String> op1 = s -> prefix.concat(s);
UnaryOperator<String> op2 = prefix::concat;   // method reference

System.out.println(op2.apply("World"));
```

La référence `prefix::concat` lie `concat` à **cet objet spécifique**.

20.3.3 Référence à une Méthode d'Instance d'un Objet Arbitraire d'un Type Donné

C'est la forme la plus "piégeuse".

Le premier paramètre de l'interface fonctionnelle devient le receiver de la méthode (`this`).

```
BiPredicate<String, String> p1 = (s1, s2) -> s1.equals(s2);
BiPredicate<String, String> p2 = String::equals;   // method reference

System.out.println(p2.test("abc", "abc"));  // true
```

> **Note**
>
> Cette forme applique la méthode au *premier argument* de la lambda.

20.3.4 Référence à un Constructeur

Les références de constructeurs remplacent des lambdas qui appellent `new` .

```
Supplier<ArrayList<String>> sup1 = () -> new ArrayList<>();
Supplier<ArrayList<String>> sup2 = ArrayList::new; // method reference

Function<Integer, ArrayList<String>> sup3 = ArrayList::new;
// appelle le constructeur ArrayList(int capacity)
```

20.3.5 Tableau Récapitulatif des Types de Method Reference

Le tableau ci-dessous résume toutes les catégories de références de méthodes.

Type	Syntax Example	Equivalent Lambda
Static method	Class::staticMethod	x -> Class.staticMethod(x)
Instance method of specific object	instance::method	x -> instance.method(x)
Instance method of arbitrary object	Class::method	(obj, x) -> obj.method(x)
Constructor	Class::new	() -> new Class()

20.3.6 Pièges Fréquents

- Une référence de méthode doit correspondre *exactement* à la signature de l'interface fonctionnelle.
- Les overloads peuvent rendre une référence de méthode ambiguë.
- La référence à méthode d'instance (`Class::method`) décale le receiver sur le paramètre 1.
- Une référence de constructeur échoue s'il n'existe pas de constructeur compatible.

```
// ❌ Ambigu : quel println() ? (println(int), println(String)...)
Consumer<String> c = System.out::println; // OK uniquement parce que le paramètre FI est Strin

// ❌ Constructeur non compatible : mauvaise interface fonctionnelle
Supplier<Integer> s = Integer::new;          // ✓ OK : appelle Integer()
Function<String, Long> f = Integer::new;     // ❌ ERREUR : le constructeur retourne Integer,
```

En cas de doute, réécrivez la method reference en lambda : si la lambda fonctionne mais pas la method reference, le problème est généralement un mismatch de signature.

◀ 19. Exceptions et Gestion des Erreurs | ▲ Index | 21. Java Optional et Streams ▶

21. Java Optional et Streams

Table des matières

21.1 Optional (Optional, OptionalInt, OptionalLong, OptionalDouble)

`Optional<T>` est un objet conteneur qui peut contenir, ou non, une valeur non nulle.

Il a été conçu pour rendre explicite « l'absence d'une valeur » et pour réduire le risque de `NullPointerException` en forçant les appelants à gérer le cas d'absence.

> **Note**
>
> - `Optional` est principalement destiné aux **types de retour**.
> - Il est généralement déconseillé pour les attributs, les paramètres de méthode et les contextes de sérialisation (sauf si un contrat API spécifique l'exige).

21.1.1 Créer des Optional

Il existe trois méthodes factory principales pour créer des Optional.

- `Optional.of(value)` → value doit être non nulle ; sinon une `NullPointerException` est levée
- `Optional.ofNullable(value)` → retourne empty si value est null
- `Optional.empty()` → un Optional explicitement vide

```
Optional<String> a = Optional.of("x");
Optional<String> b = Optional.ofNullable(null); // Optional.empty
Optional<String> c = Optional.empty();
```

21.1.2 Lire des valeurs en toute sécurité

Les Optional fournissent plusieurs moyens d'accéder à la valeur encapsulée.

- `isPresent()` / `isEmpty()` → test de présence
- `get()` → retourne la valeur ou lève `NoSuchElementException` si absente (déconseillé)
- `orElse(defaultValue)` → retourne la valeur ou la valeur par défaut (évaluée immédiatement)
- `orElseGet(supplier)` → retourne la valeur ou le résultat du supplier (supplier évalué de manière lazy)
- `orElseThrow()` → retourne la valeur ou lève `NoSuchElementException`
- `orElseThrow(exceptionSupplier)` → retourne la valeur ou lève une exception personnalisée

```
Optional<String> opt = Optional.of("java");

String v1 = opt.orElse("default");
String v2 = opt.orElseGet(() -> "computed");
String v3 = opt.orElseThrow(); // ok car opt est présent
```

> **Note**
>
> - Un piège courant : `orElse(...)` évalue son argument même si l'Optional est présent.
> - Utilisez `orElseGet(...)` lorsque la valeur par défaut est coûteuse à calculer.

21.1.3 Transformer des Optional

Les Optional prennent en charge des transformations fonctionnelles similaires aux streams, mais avec une sémantique « 0 ou 1 élément ».

- `map(fn)` → transforme la valeur si elle est présente
- `flatMap(fn)` → transforme en un Optional aplati, sans imbrication
- `filter(predicate)` → conserve la valeur uniquement si le predicate est true

```java
Optional<String> name = Optional.of("Alice");

Optional<Integer> len =
    name.map(String::length); // Optional[5]

Optional<String> filtered =
    name.filter(n -> n.startsWith("A")); // Optional[Alice]

System.out.println(len.orElse(0));
System.out.println(filtered.orElseGet(() -> "11"));
```

Sortie :

```
5
Alice
```

> **Note**
>
> - `map` encapsule le résultat dans un Optional.
> - Si votre fonction de mapping retourne déjà un Optional, utilisez `flatMap` pour éviter l'imbrication `Optional<Optional<T>>`.

21.1.4 Optional et Streams

Un pattern de pipeline très courant consiste à effectuer un `map` vers un Optional puis à supprimer les éléments absents.

Depuis Java 9, `Optional` fournit `stream()` pour convertir « présent → un élément » et « vide → zéro élément ».

```java
        Stream<String> words = Stream.of("a", "bb", "ccc");

        words.map(w -> w.length() > 1 ? Optional.of(w.length()) : Optional.<Integer>empty())
             .flatMap(Optional::stream) // supprime les éléments vides
             .forEach(System.out::println);
```

Sortie :

```
2
3
```

> **Note**
>
> Avant Java 9, ce pattern nécessitait `filter(Optional::isPresent)` plus `map(Optional::get)`.

21.1.5 Optional pour les types primitifs

Les streams primitifs utilisent des optional primitifs pour éviter le boxing : `OptionalInt`, `OptionalLong`, `OptionalDouble`.

Ils reflètent l'API principale de Optional avec des getters primitifs comme `getAsInt()`.

- `OptionalInt.getAsInt()` / `OptionalLong.getAsLong()` / `OptionalDouble.getAsDouble()`
- `orElse(...)` / `orElseGet(...)` / `orElseThrow(...)`

```java
OptionalInt m = IntStream.of(3, 1, 2).min(); // OptionalInt[1]
int value = m.orElse(0); // 1
```

21.1.6 Pièges courants

- Ne pas utiliser `get()` sans vérifier la présence ; préférer `orElseThrow` ou les transformations

- Éviter de retourner `null` au lieu de `Optional.empty()` ; une référence Optional elle-même ne devrait pas être null
- Se souvenir que `average()` sur les streams primitifs retourne toujours `OptionalDouble` (même pour `IntStream` et `LongStream`)
- Utiliser `orElseGet` lorsque le calcul de la valeur par défaut est coûteux en termes de performances

---

21.2 Qu'est-ce qu'un Stream (et ce que ce n'est pas)

Un `Stream Java` représente une séquence d'éléments (un pipeline) prenant en charge des opérations de style fonctionnel.

Les streams sont conçus pour le traitement des données, et non pour leur stockage.

Caractéristiques clés :

- Un stream ne stocke pas de données
- Un stream est lazy — rien ne se produit tant qu'une opération terminale n'est pas invoquée
- Un stream ne peut être consommé qu'une seule fois
- Les streams encouragent des opérations sans effets de bord

> **Note**
>
> Les streams sont conceptuellement similaires aux requêtes de bases de données : ils décrivent ce qu'il faut calculer, et non comment itérer.

21.3 Architecture du pipeline Stream

Chaque pipeline de stream se compose de trois phases distinctes :

- 1 **Source**
- 2 Zéro ou plusieurs **Opérations intermédiaires**
- 3 Exactement une **Opération terminale**

21.3.1 Sources de Stream

Les sources courantes de stream incluent :

- Collections : `collection.stream()`
- Tableaux : `Arrays.stream(array)`
- Canaux I/O et fichiers
- Streams infinis : `Stream.iterate`, `Stream.generate`

```
List<String> names = List.of("Ana", "Bob", "Carla");

Stream<String> s = names.stream();
```

21.3.2 Opérations intermédiaires

Opérations intermédiaires :

- Retournent un nouveau stream
- Sont évaluées de manière lazy
- Ne déclenchent pas l'exécution

21.3.2.1 Tableau des opérations intermédiaires courantes

Method	Common input Params	Return value	Desctiption
`filter`	Predicate	`Stream<T>`	filtre le stream selon une correspondance du predicate
`map`	Function	`Stream<R>`	transforme un stream par un mapping un-à-un entrée/sortie
`flatMap`	Function	`Stream<R>`	aplatit des streams imbriqués en un seul stream
`sorted`	(none) or Comparator	`Stream<T>`	trie par ordre naturel ou selon le Comparator fourni
`distinct`	(none)	`Stream<T>`	supprime les éléments dupliqués
`limit` / `skip`	long	`Stream<T>`	limite la taille ou saute des éléments
`peek`	Consumer	`Stream<T>`	exécute une action avec effet de bord pour chaque élément (debugging)

- Exemple :

```
        List<String> names = List.of("Ana", "Bob", "Carla", "Mario");

        names.stream()
             .filter(n -> n.length() > 3)
             .map(String::toUpperCase)
             .forEach(System.out::println);
```

Sortie :

```
CARLA
MARIO
```

> **Note**
>
> Les opérations intermédiaires décrivent uniquement le calcul. Aucun élément n'est encore traité.

21.3.3 Opérations terminales

Opérations terminales :

- Déclenchent l'exécution
- Consomment le stream
- Produisent un résultat ou un effet de bord

21.3.3.1 Tableau des opérations terminales

Method	Return value	behaviour for infinite streams
`forEach`	**void**	ne termine pas
`collect`	varie	ne termine pas
`reduce`	varie	ne termine pas
`findFirst` / `findAny`	`Optional<T>`	termine
`anyMatch` / `allMatch` / `noneMatch`	**boolean**	peut terminer tôt (court-circuit)
`min` / `max`	`Optional<T>`	ne termine pas
`count`	**long**	ne termine pas

21.4 Évaluation paresseuse et court-circuitage

```
var newNames = new ArrayList<String>();

newNames.add("Bob");
newNames.add("Dan");

// Les streams sont évalués de manière paresseuse : ceci ne parcourt pas encore les données,
// cela crée uniquement une description du pipeline liée à la source.
var stream = newNames.stream();

newNames.add("Erin");

// L'opération terminale déclenche l'évaluation. Le stream voit la source mise à jour,
// donc le count inclut "Erin".
stream.count(); // 3
```

> **Note**
>
> Un stream est lié à sa *source* (`newNames`), et le pipeline n'est pas exécuté tant qu'une opération terminale n'est pas invoquée.
> Pour cette raison, si vous **modifiez la collection avant l'opération terminale**, l'opération terminale « voit » les nouveaux éléments (ici, `Erin`).
> En général, toutefois, **modifier la source pendant qu'un pipeline de stream est en cours d'utilisation est une mauvaise pratique** et peut conduire à un comportement non déterministe (ou à une `ConcurrentModificationException` avec certaines sources/opérations).
> La règle pratique est : *construire la source, puis créer et exécuter le stream sans la modifier*.

Les streams traitent les éléments **un par un**, en circulant « verticalement » à travers le pipeline plutôt que étape par étape.

Ci-dessous, nous modifions l'exemple pour utiliser une opération terminale **à court-circuit** : `findFirst()` .

```
Stream.of("a", "bb", "ccc")
    .filter(s -> {
        System.out.println("filter " + s);
        return s.length() > 1;
    })
    .map(s -> {
        System.out.println("map " + s);
        return s.toUpperCase();
    })
    .findFirst()
    .ifPresent(System.out::println);
```

Ordre d'exécution :

> **Note**
>
> Seul le nombre minimal d'éléments requis par l'opération terminale est traité.

```
filter a
filter bb
map bb
BB
```

`findFirst()` est satisfait dès qu'il trouve le **premier** élément qui traverse avec succès le pipeline (ici `"bb"`), donc :

- `"ccc"` n'est jamais traité (ni `filter` ni `map`) ;
- l'évaluation paresseuse évite un travail inutile par rapport à une opération terminale qui consomme tous les éléments (comme `forEach` ou `count`).

> **Important**
>
> `allMatch`, `noneMatch`, `anyMatch`, `findFirst` et `findAny` sont des **opérations terminales à court-circuit** (*short-circuiting terminal operations*).
>
> Cela signifie que le prédicat fourni **n'est pas nécessairement évalué pour chaque élément du stream**.
> L'opération peut s'arrêter dès que le résultat final peut déjà être déterminé.
>
> Par exemple, avec `allMatch`, si le prédicat retourne `false` pour le **premier élément**, le résultat global de l'opération est déjà connu comme étant `false`. Comme `allMatch` exige que tous les éléments satisfassent le prédicat, trouver un seul élément qui ne correspond pas suffit pour déterminer le résultat.
>
> Par conséquent, dès qu'un tel élément est rencontré, **les éléments restants du stream n'ont plus besoin d'être testés**, et le traitement du stream s'arrête immédiatement.

21.5 Opérations stateless vs stateful

21.5.1 Opérations stateless

Des opérations comme `map` et `filter` traitent chaque élément indépendamment.

21.5.2 Opérations stateful

Des opérations comme `distinct`, `sorted` et `limit` nécessitent le maintien d'un état interne.

> **Note**
>
> Les opérations stateful peuvent avoir un impact sévère sur les performances des streams parallèles.

21.6 Ordonnancement des Streams et déterminisme

Les streams peuvent être :

- Ordonnés (ex. `List.stream()`)
- Non ordonnés (ex. `HashSet.stream()`)

Certaines opérations respectent l'ordre de parcours :

- `forEachOrdered`
- `findFirst`

> **Note**
>
> Dans les streams parallèles, `forEach` ne garantit pas l'ordre.

21.7 Streams parallèles

Les streams parallèles divisent le travail entre threads en utilisant `ForkJoinPool.commonPool()`.

```
int sum =
IntStream.range(1, 1_000_000)
    .parallel()
    .sum();
```

Règles pour des streams parallèles sûrs :

- Aucun effet de bord
- Aucun état partagé mutable
- Uniquement des opérations associatives

> **Note**
>
> Les streams parallèles peuvent être plus lents pour des charges de travail légères.

21.8 Opérations de réduction

21.8.1 `reduce()` : combiner un stream en un seul objet

Il existe trois signatures de méthode pour cette opération :

- public `Optional<T>` **reduce**(`BinaryOperator<T> accumulator`);
- public `T` **reduce**(`T identity`, `BinaryOperator<T> accumulator`);
- public `<U> U` **reduce**(`U identity`, `BiFunction<U, ? super T, U> accumulator`, `BinaryOperator<U> combiner`);

```
int sum = Stream.of(1, 2, 3)
    .reduce(0, Integer::sum);
```

La réduction requiert :

- **Identity** : valeur initiale pour chaque réduction partielle ; doit être un élément neutre ; exemple : 0 pour la somme, 1 pour la multiplication, collection vide pour la collecte ;
- **Accumulator** : incorpore un élément du stream dans un résultat partiel ;
- (Optionnel) **Combiner** : fusionne deux résultats partiels ; utilisé uniquement lorsque le stream est parallèle ; ignoré pour les streams séquentiels

> **Note**
>
> L'accumulator doit être associatif et stateless.

21.8.1.1 Modèle mental correct

- Accumulator : résultat + élément
- Combiner : résultat + résultat

Exemple 1 : Utilisation correcte (somme des longueurs)

```
int totalLength =
    Stream.of("a", "bb", "ccc")
          .parallel()
          .reduce(
              0,                              // identity
              (sum, s) -> sum + s.length(), // accumulator
              (left, right) -> left + right // combiner
          );
```

Ce qui se passe en parallèle

Supposons que le stream soit divisé :

- Thread 1 : "a", "bb" → 0 + 1 + 2 = 3
- Thread 2 : "ccc" → 0 + 3 = 3

Ensuite, le combiner fusionne les résultats partiels :

```
3 + 3 = 6
```

Exemple 2 : Combiner ignoré dans les streams séquentiels

```
int result =
    Stream.of("a", "bb", "ccc")
          .reduce(
              0,
              (sum, s) -> sum + s.length(),
              (x, y) -> {
                  throw new RuntimeException("Never called");
              }
          );
```

Exemple 3 : Combiner incorrect

```
int result =
    Stream.of(1, 2, 3, 4)
          .parallel()
          .reduce(
              0,
              (a, b) -> a - b,    // accumulator
              (x, y) -> x - y     // combiner
          );
```

Pourquoi ceci est incorrect

La soustraction n'est pas associative.

Exécution possible :

- Thread 1 : 0 - 1 - 2 = -3
- Thread 2 : 0 - 3 - 4 = -7

Combiner :

```
-3 - (-7) = 4
```

Le résultat séquentiel serait :

```
(((0 - 1) - 2) - 3) - 4 = -10
```

> **Warning**
>
> ❌ Les résultats parallèles et séquentiels diffèrent → réduction illégale

21.8.2 `collect()`

`collect` est une réduction mutable optimisée pour le regroupement et l'agrégation.

C'est l'outil standard de l'API Stream pour la « réduction mutable » : vous accumulez des éléments dans un conteneur mutable (comme une List, Set, Map, StringBuilder, objet résultat personnalisé), puis, éventuellement, vous fusionnez les conteneurs partiels lors de l'exécution en parallèle.

La forme générale est :

- `public <R> R **collect**(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)` ;

Une version couramment utilisée est :

- `public <R, A> R **collect**(Collector<? super T, A, R> collector)` ;

où `Collectors.*` fournit des collectors préconstruits (grouping, mapping, joining, counting, etc.).

Signification :

- **supplier** : crée un nouveau conteneur de résultat vide (ex. `new ArrayList<>()`)
- **accumulator** : ajoute un élément dans ce conteneur (ex. `list::add`)
- **combiner** : fusionne deux conteneurs (ex. `list1.addAll(list2)`)

21.8.3 Pourquoi `collect()` est différent de `reduce()`

- **Intention** : mutation vs immutabilité
  - `reduce()` est conçu pour une réduction de style immuable : combiner des valeurs en une nouvelle valeur (ex. somme, min, max).
  - `collect()` est conçu pour des conteneurs mutables : construire une List, Map, StringBuilder, etc.
- **Correction** en parallèle
  - `reduce()` exige que l'opération soit :
    - associative
    - stateless
    - compatible avec les règles d'identity/combiner
  - `collect()` est conçu pour supporter le parallélisme en toute sécurité grâce à :
    - la création d'un conteneur par thread (supplier)
    - l'accumulation locale (accumulator)
    - la fusion finale (combiner)
- **Performance**
  - `collect()` peut être optimisé car le runtime du stream sait que vous construisez des conteneurs :
    - il peut éviter des copies inutiles
    - il peut pré-dimensionner ou utiliser des implémentations spécialisées (selon le collector)
    - c'est l'approche idiomatique et attendue
    - utiliser `reduce()` pour construire une collection crée souvent des objets supplémentaires ou force une mutation non sûre.
- Exemple : « collecter dans une List » de la bonne manière

```
List<String> longNames =
    names.stream()
         .filter(s -> s.length() > 3)
         .collect(Collectors.toList());
```

- Exemple : groupingBy avec explication

```
Map<Integer, List<String>> byLength =
    names.stream()
         .collect(Collectors.groupingBy(String::length));
```

Ce qui se passe conceptuellement :

- Le collector crée une `Map<Integer, List<String>>` vide
- Pour chaque nom :
    - calcule la clé (`String::length`)
    - l'ajoute dans la liste du bucket approprié
- En parallèle :
    - chaque thread construit ses propres maps partielles
    - le combiner fusionne les maps en fusionnant les listes par clé

21.9 Pièges courants des Streams

- Réutiliser un stream déjà consommé → `IllegalStateException`
- Modifier des variables externes à l'intérieur des lambda
- Supposer l'ordre d'exécution dans les streams parallèles
- Utiliser `peek` pour la logique au lieu du debugging

21.10 Streams primitifs

Java fournit trois types de streams spécialisés pour éviter le surcoût du boxing et pour permettre des opérations centrées sur les nombres :

- `IntStream` pour `int`
- `LongStream` pour `long`
- `DoubleStream` pour `double`

Les streams primitifs restent des streams (pipelines lazy, opérations intermédiaires + terminales, usage unique), mais ils ne sont pas **génériques** et utilisent des interfaces fonctionnelles spécialisées pour les primitifs (ex. `IntPredicate`, `LongUnaryOperator`, `DoubleConsumer`).

> **Note**
>
> Utilisez les streams primitifs lorsque les données sont naturellement numériques ou lorsque la performance compte : ils évitent le surcoût de boxing/unboxing et fournissent des opérations terminales numériques supplémentaires.

21.10.1 Pourquoi les streams primitifs sont importants

- Performance : éviter l'allocation d'objets wrapper et le boxing/unboxing répété dans de grands pipelines
- Commodité : réductions numériques intégrées comme `sum()`, `average()`, `summaryStatistics()`
- Pièges courants : comprendre quand les résultats sont primitifs vs `OptionalInt` / `OptionalLong` / `OptionalDouble`

21.10.2 Méthodes courantes de création

Les méthodes suivantes sont les plus fréquemment utilisées pour créer des streams primitifs. De nombreuses questions de certification commencent par identifier le type de stream créé par une méthode factory.

Sources
IntStream.of(int…)
IntStream.range(int startInclusive, int endExclusive)
IntStream.rangeClosed(int startInclusive, int endInclusive)
IntStream.iterate(int seed, IntUnaryOperator f) // infini sauf limitation
IntStream.iterate(int seed, IntPredicate hasNext, IntUnaryOperator f)
IntStream.generate(IntSupplier s) // infini sauf limitation
LongStream.of(long…)
LongStream.range(long startInclusive, long endExclusive)
LongStream.rangeClosed(long startInclusive, long endInclusive)
LongStream.iterate(long seed, LongUnaryOperator f)
LongStream.iterate(long seed, LongPredicate hasNext, LongUnaryOperator f)
LongStream.generate(LongSupplier s)
DoubleStream.of(double…)
DoubleStream.iterate(double seed, DoubleUnaryOperator f)
DoubleStream.iterate(double seed, DoublePredicate hasNext, DoubleUnaryOperator f)
DoubleStream.generate(DoubleSupplier s)

> **Important**
>
> - Seuls `IntStream` et `LongStream` fournissent `range()` et `rangeClosed()`.
> - Il n'existe pas de `DoubleStream.range` car le comptage avec des doubles pose des problèmes d'arrondi.

21.10.3 Méthodes de mapping spécialisées pour les primitifs

Les streams primitifs fournissent des opérations de mapping **uniquement pour primitifs** afin d'éviter le boxing :

- `IntStream.map(IntUnaryOperator)` → `IntStream`
- `IntStream.mapToLong(IntToLongFunction)` → `LongStream`
- `IntStream.mapToDouble(IntToDoubleFunction)` → `DoubleStream`
- `LongStream.map(LongUnaryOperator)` → `LongStream`
- `LongStream.mapToInt(LongToIntFunction)` → `IntStream`
- `LongStream.mapToDouble(LongToDoubleFunction)` → `DoubleStream`
- `DoubleStream.map(DoubleUnaryOperator)` → `DoubleStream`
- `DoubleStream.mapToInt(DoubleToIntFunction)` → `IntStream`
- `DoubleStream.mapToLong(DoubleToLongFunction)` → `LongStream`

21.10.4 Tableau de mapping entre `Stream<T>` et les streams primitifs

Ce tableau résume les principales conversions entre streams d'objets et streams primitifs.

La colonne « From » indique quelles méthodes sont disponibles et le type de stream cible résultant.

From (source)	To (target)	Primary method(s)
`Stream<T>`	`Stream<R>`	`map(Function<? super T, ? extends R>)`
`Stream<T>`	`Stream<R>` (flatten)	`flatMap(Function<? super T, ? extends Stream<? extends R>>)`
`Stream<T>`	`IntStream`	`mapToInt(ToIntFunction<? super T>)`
`Stream<T>`	`LongStream`	`mapToLong(ToLongFunction<? super T>)`
`Stream<T>`	`DoubleStream`	`mapToDouble(ToDoubleFunction<? super T>)`
`Stream<T>`	`IntStream` (flatten)	`flatMapToInt(Function<? super T, ? extends IntStream>)`
`Stream<T>`	`LongStream` (flatten)	`flatMapToLong(Function<? super T, ? extends LongStream>)`
`Stream<T>`	`DoubleStream` (flatten)	`flatMapToDouble(Function<? super T, ? extends DoubleStream>)`
`IntStream`	`Stream<Integer>`	`boxed()`
`LongStream`	`Stream<Long>`	`boxed()`
`DoubleStream`	`Stream<Double>`	`boxed()`
`IntStream`	`Stream<U>`	`mapToObj(IntFunction<? extends U>)`
`LongStream`	`Stream<U>`	`mapToObj(LongFunction<? extends U>)`
`DoubleStream`	`Stream<U>`	`mapToObj(DoubleFunction<? extends U>)`
`IntStream`	`LongStream`	`asLongStream()`
`IntStream`	`DoubleStream`	`asDoubleStream()`
`LongStream`	`DoubleStream`	`asDoubleStream()`

> **Important**
>
> - Il n'existe pas d'opération **`unboxed()`**.
> - Pour passer des wrappers aux primitifs, vous devez partir de `Stream<T>` et utiliser `mapToInt` / `mapToLong` / `mapToDouble`.

21.10.5 Opérations terminales et leurs types de résultat

Les streams primitifs disposent de plusieurs opérations terminales qui sont uniques ou qui ont des types de retour spécifiques aux primitifs.

Terminal operation	IntStream returns	LongStream returns	DoubleStream returns
`count()`	long	long	long
`sum()`	int	long	double
`min()` / max()	OptionalInt	OptionalLong	OptionalDouble
`average()`	OptionalDouble	OptionalDouble	OptionalDouble
`findFirst()` / findAny()	OptionalInt	OptionalLong	OptionalDouble
`reduce(op)`	OptionalInt	OptionalLong	OptionalDouble
`reduce(identity, op)`	int	long	double
`summaryStatistics()`	IntSummaryStatistics	LongSummaryStatistics	DoubleSummaryStatistics

> **Warning**
>
> - Même pour `IntStream` et `LongStream`, **`average()`** retourne `OptionalDouble` (et non `OptionalInt` ou `OptionalLong`).

- Exemple 1 : `Stream<String>` → `IntStream` → opérations terminales primitives

```
List<String> words = List.of("a", "bb", "ccc");

int totalLength = words.stream()
    .mapToInt(String::length) // IntStream
    .sum(); // int

// totalLength = 1 + 2 + 3 = 6
```

- Exemple 2 : `IntStream` → `Stream<Integer>` boxé (boxing introduit)

```
Stream<Integer> boxed = IntStream.rangeClosed(1, 3) // 1,2,3
    .boxed(); // Stream<Integer>
```

- Exemple 3 : stream primitif → stream d'objets via `mapToObj`

```
Stream<String> labels = IntStream.range(1, 4) // 1,2,3
    .mapToObj(i -> "N=" + i); // Stream<String>
```

21.10.6 Pièges et gotchas courants

- Ne pas confondre `Stream<Integer>` avec `IntStream` : leurs méthodes de mapping et interfaces fonctionnelles diffèrent
- `IntStream.sum()` retourne `int` mais `IntStream.count()` retourne `long`
- `average()` retourne toujours `OptionalDouble` pour tous les types de streams primitifs
- Utiliser `boxed()` réintroduit le boxing ; ne le faire que si l'API en aval requiert des objets (ex. collecte dans `List<Integer>`)
- Attention aux conversions de narrowing : `LongStream.mapToInt` et `DoubleStream.mapToInt` peuvent tronquer les valeurs

---

21.11 Collectors (collect(), Collector et les méthodes factory de Collectors)

Un `Collector` décrit comment accumuler des éléments de stream dans un résultat final.

L'opération terminale `collect(...)` exécute cette recette.

La classe utilitaire `Collectors` fournit des collectors prêts à l'emploi pour des tâches courantes d'agrégation.

21.11.1 collect() vs Collector

Il existe deux manières principales de collecter :

- `collect(Collector)` → la forme courante utilisant `Collectors.*`
- `collect(supplier, accumulator, combiner)` → réduction mutable explicite (plus bas niveau)

```
List<String> list =
Stream.of("a", "b")
    .collect(Collectors.toList());

StringBuilder sb =
Stream.of("a", "b")
    .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append);
```

> **Note**
>
> Utilisez `collect(supplier, accumulator, combiner)` lorsque vous avez besoin d'un conteneur mutable personnalisé et que vous ne souhaitez pas implémenter un `Collector` complet.

21.11.2 Collectors principaux

Voici les collectors les plus fréquemment utilisés et les plus susceptibles d'apparaître dans les questions d'examen.

- `toList()` → `List<T>` (aucune garantie sur la mutabilité ou l'implémentation)
- `toSet()` → `Set<T>`
- `toCollection(supplier)` → type de collection spécifique (ex. `TreeSet`)
- `joining(delim, prefix, suffix)` → `String` à partir d'éléments `CharSequence`
- `counting()` → comptage `Long`
- `summingInt` / `summingLong` / `summingDouble` → sommes numériques
- `averagingInt` / `averagingLong` / `averagingDouble` → moyennes numériques
- `minBy(comparator)` / `maxBy(comparator)` → `Optional<T>`
- `mapping(mapper, downstream)` → transforme puis collecte avec un downstream
- `filtering(predicate, downstream)` → filtre à l'intérieur du collector (Java 9+)

21.11.3 Collectors de regroupement

`groupingBy` classe les éléments dans des buckets à l'aide d'une fonction classifier.

Il produit une `Map<K, V>` où `V` dépend du collector downstream.

```
Map<Integer, List<String>> byLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length));
System.out.println("byLen: " + byLen.toString());
```

Sortie :

```
byLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

Avec un collector downstream, vous contrôlez ce que contient chaque bucket :

```java
Map<Integer, Long> countByLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length, Collectors.counting()));
System.out.println("countByLen: " + countByLen.toString());

Map<Integer, Set<String>> setByLen =
Stream.of("a", "bb", "ccc", "dd")
    .collect(Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println("setByLen: " + setByLen.toString());
```

Sortie :

```
countByLen: {1=1, 2=2, 3=1}
setByLen: {1=[a], 2=[bb, dd], 3=[ccc]}
```

> **Warning**
>
> Faites attention au type de la valeur de la map résultante. Exemple : `groupingBy(..., counting())` produit `Map<K, Long>` (et non `int`).

21.11.4 partitioningBy

`partitioningBy` divise le stream en exactement deux groupes à l'aide d'un `Predicate` booléen. Il retourne toujours une map avec les clés `true` et `false`.

```java
Map<Boolean, List<String>> parts =
Stream.of("a", "bb", "ccc")
    .collect(Collectors.partitioningBy(s -> s.length() > 1));
System.out.println("parts: " + parts.toString());
```

Sortie :

```
parts: {false=[a], true=[bb, ccc]}
```

> **Note**
>
> `partitioningBy` crée toujours deux buckets, tandis que `groupingBy` peut en créer plusieurs. Les deux prennent en charge des collectors downstream.

21.11.5 toMap et règles de fusion

`toMap` lève une exception en cas de clés dupliquées sauf si vous fournissez une fonction de fusion.

```java
Map<Integer, String> m1 =
Stream.of("aa", "bb")
    .collect(Collectors.toMap(String::length, s -> s)); // ❌ Exception in thread "main" java

Map<Integer, String> m2 =
Stream.of("aa", "bb", "cc")
    .collect(Collectors.toMap(String::length, s -> s, (oldV, newV) -> oldV + "," + newV)); //
```

Sortie :

```
m2: {2=aa,bb,cc}
```

21.11.6 collectingAndThen

`collectingAndThen(downstream, finisher)` permet d'appliquer une transformation finale après la collecte (ex. rendre la liste non modifiable).

```java
List<String> unmodifiable =
Stream.of("a", "b", "c")
    .collect(Collectors.collectingAndThen(Collectors.toList(), List::copyOf));
```

21.11.7 Comment les collectors se rapportent aux streams parallèles

Les collectors sont conçus pour fonctionner avec des streams parallèles en utilisant supplier/accumulator/combiner en interne. En parallèle, chaque worker construit un conteneur de résultat partiel puis fusionne les conteneurs.

- L'accumulator modifie un conteneur par thread (aucun état partagé mutable)
- Le combiner fusionne les conteneurs (requis pour l'exécution parallèle)
- Certains collectors sont « concurrent » ou possèdent des caractéristiques influençant les performances et l'ordonnancement

> **Note**
>
> Préférez `collect(Collectors.toList())` à l'utilisation de `reduce` pour construire des collections. `reduce` est destiné aux réductions de style immuable ; `collect` est destiné aux conteneurs mutables.

◀ 20. Programmation Fonctionnelle en Java | ▲ Index | 22. Introduction au Framework des Collections ▶

Module 06

Collections Framework

22. Introduction au Framework des Collections

Table des matières

Le `Java Collections Framework (JCF)` est un ensemble **d'interfaces, de classes et d'algorithmes** conçu pour stocker, manipuler et traiter des groupes de données de manière efficace.

Il fournit une architecture unifiée pour gérer les collections, permettant aux développeurs d'écrire du code réutilisable et interopérable avec des comportements prévisibles et des caractéristiques de performance.

Ce chapitre introduit les concepts fondamentaux nécessaires avant d'étudier List, Set, Queue, Map et les Sequenced Collections, explorés en détail dans les chapitres suivants.

22.1 Qu'est-ce que le Framework des Collections ?

Le Framework des Collections fournit :

- Un **ensemble d'interfaces** (Collection, List, Set, Queue, Deque, Map…)
- Un **ensemble d'implémentations** (ArrayList, HashSet, TreeSet, LinkedList…)
- Un **ensemble d'algorithmes utilitaires** (tri, recherche, copie, inversion…) dans java.util.Collections et java.util.Arrays.
- Un langage commun pour les attentes de performance (complexité Big-O).

Toutes les principales structures de collection partagent une conception cohérente, de sorte que le code fonctionnant avec une implémentation peut souvent être réutilisé avec une autre.

22.2 Les Interfaces Principales

Au cœur du Java Collections Framework se trouve un petit ensemble **d'interfaces racines** qui définissent des comportements génériques de gestion des données.

- **List** : une collection `ordonnée` d'éléments qui autorise les `doublons` ;
- **Set** : une collection qui n'autorise pas les `doublons` ;
- **Queue** : une collection conçue pour contenir des éléments en cours de traitement, typiquement FIFO (first-in-first-out), avec des variantes comme les priority queues et les deques.
- **Map** : une structure qui associe des clés à des valeurs, où les clés dupliquées ne sont pas autorisées ; chaque clé peut être associée à au plus une valeur.

22.2.1 Principales interfaces de Collection

Ci-dessous se trouve la hiérarchie conceptuelle.

```
java.util
├─ Collection<E>
│ ├─ SequencedCollection<E> (Java 21+)
│ │ ├─ List<E>
│ │ │   ├─ ArrayList<E>
│ │ │   └─ LinkedList<E> (also implements Deque<E>)
│ │ └─ Deque<E> (also extends Queue<E>)
│ │     ├─ ArrayDeque<E>
│ │     └─ LinkedList<E>
│ ├─ Set<E>
│ │     ├─ SequencedSet<E> (Java 21+)
│ │     │       └─ LinkedHashSet<E>
│ │     ├─ SortedSet<E>
│ │     │       └─ NavigableSet<E>
│ │     │           └─ TreeSet<E>
│ │     ├─ HashSet<E>
│ │     └─ (other Set implementations)
│ ├─ Queue<E>
│ │     ├─ Deque<E> (already under SequencedCollection<E>)
│ │     ├─ PriorityQueue<E>
│ │     └─ (other Queue implementations)
│ └─ (other Collection implementations)
│
└─ Map<K,V> (not a Collection)
    ├─ SequencedMap<K,V> (Java 21+)
    │   └─ LinkedHashMap<K,V>
    ├─ SortedMap<K,V>
    │   └─ NavigableMap<K,V>
    │   └─ TreeMap<K,V>
    ├─ HashMap<K,V>
    ├─ Hashtable<K,V>
    └─ (other Map/ConcurrentMap implementations)
```

L'interface **Map** n'étend pas Collection, car une map stocke des paires clé/valeur plutôt que des valeurs uniques.

22.2.2 Hiérarchie de Map

```
java.util
└─ Map<K,V>
    ├─ SequencedMap<K,V> (Java 21+)
    │   └─ LinkedHashMap<K,V>
    ├─ SortedMap<K,V>
    │   └─ NavigableMap<K,V>
    │       └─ TreeMap<K,V>
    ├─ HashMap<K,V>
    ├─ Hashtable<K,V>
    └─ ConcurrentMap<K,V> (java.util.concurrent)
        └─ ConcurrentHashMap<K,V>
```

22.3 Collections Sequenced (Java 21+)

Java 21 introduit la nouvelle interface `SequencedCollection`, qui formalise l'idée qu'une collection maintient un **ordre de parcours défini**. Cela était déjà vrai pour List, LinkedHashSet, LinkedHashMap, Deque, etc., mais ce comportement est désormais standardisé.

- `SequencedCollection` définit des méthodes telles que `getFirst()`, `getLast()`, `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()` et `reversed()`.
- SequencedSet et SequencedMap étendent ce concept aux sets et aux maps.

Cela simplifie considérablement la spécification des comportements d'ordonnancement et sera utilisé dans tous les chapitres suivants.

22.4 Pourquoi le Framework des Collections existe

- Éviter de réinventer les structures de données
- Fournir des algorithmes bien testés et hautement performants

- Améliorer l'interopérabilité grâce à des interfaces partagées
- Prendre en charge les types génériques pour des collections sûres du point de vue du type

Avant Java 1.2, les structures de données étaient ad hoc, incohérentes et non typées.

Le Collections Framework a unifié tout cela dans une API cohérente.

22.5 Les deux côtés du Framework : Collections vs Maps

« Map étend-elle Collection ? » **Non.** Une Map stocke des **paires**, tandis qu'une Collection stocke des **éléments uniques**.

- Collection = List, Set, Queue, Deque, SequencedCollection
- Map = stockage clé/valeur de type dictionnaire

22.6 Types génériques dans le Framework des Collections

Les collections sont presque toujours utilisées avec des generics. L'utilisation de raw types est déconseillée.

```
List<String> names = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();
```

> **Note**
>
> Les generics dans les collections fonctionnent via le `type erasure` : se référer au paragraphe "**18.4 Type Erasure**" dans le chapitre : Generics in Java.

22.7 Mutabilité vs Immutabilité

De nombreuses méthodes de la Collections API renvoient des collections **unmodifiable** :

```
List<String> immutable = List.of("a", "b");
immutable.add("c"); // ❌ UnsupportedOperationException
```

Java fournit plusieurs moyens de créer des collections immuables :

- `List.of()`, `Set.of()`, `Map.of()`
- `List.copyOf(collection)`
- wrappers `Collections.unmodifiableList(...)`
- `Records` utilisés comme conteneurs de valeurs immuables

> **Note**
>
> La méthode `Arrays.asList(varargs)`, qui est construite sur un tableau, se comporte différemment : voir les exemples ci-dessous.

```java
String[] vargs = new String[] {"u", "v", "z"};
List<String> fromAsList = Arrays.asList(vargs);

List<String> immutable1 = List.of(vargs);
immutable1.add("c"); // ❌ UnsupportedOperationException

List<String> immutable2 = List.copyOf(fromAsList);
immutable2.set(0, "k"); // ❌ UnsupportedOperationException


// Nous ne pouvons pas ADD ou REMOVE des éléments de "fromAsList" mais nous pouvons les rempla
// soit en modifiant le tableau sous-jacent "vargs", soit en mutant la liste elle-même :


fromAsList.set(0, "k");  // la mise à jour sera également reflétée dans le tableau sous-jacent
```

> **Note**
>
> `Arrays.asList(...)` renvoie une vue List de taille fixe, mais **mutable**, supportée par le tableau d'origine. Vous ne pouvez pas ajouter/supprimer des éléments, mais vous pouvez remplacer ceux existants.

22.8 Attentes de Performance Big-O

Comprendre la complexité des types de collection est essentiel. Voici quelques exemples courants :

Type	Methods	Complexity
ArrayList	`get()`, `add()`, `remove()`	`O(1)`, `O(1) amorti`, `O(n)`
LinkedList	`get()`, `add/remove first/last`	`O(n)`, `O(1)`
HashSet	`add()`, `contains()`, `remove()`	~ `O(1)`
TreeSet	`add()`, `contains()`, `remove()`	`O(log n)`
HashMap	`get()/put()`	~ `O(1) en moyenne`
TreeMap	`get()/put()`	`O(log n)`
Deque	`add/remove first/last`	`O(1)`

> **Note**
>
> Ces valeurs sont des moyennes ; le pire cas peut être différent (en particulier pour les structures basées sur le hachage).

22.9 Résumé

- Le Collection Framework est construit sur un petit ensemble d'interfaces principales.
- Java 21 ajoute les Sequenced Collections pour unifier le comportement d'ordonnancement.
- Les Map ne sont pas des Collection — elles forment une hiérarchie parallèle.
- Les collections font un usage intensif des generics.
- La mutabilité est importante — les méthodes factory renvoient souvent des collections immuables.
- Les caractéristiques de performance sont prévisibles.

---

23. Opérations Partagées des Collections & Égalité

Table des matières

Ce chapitre couvre les opérations fondamentales partagées dans toute la Java Collections API, y compris la manière dont l'égalité est déterminée à l'intérieur des collections.

Ces concepts s'appliquent à toutes les principales familles de collections basées sur Collection (List, Set, Queue, Deque et leurs variantes Sequenced).

Map partage plusieurs comportements conceptuels (itération, égalité) mais n'hérite pas de Collection.

Maîtriser ces opérations est essentiel, car elles expliquent comment les collections se comportent lors de l'ajout, de la recherche, de la suppression, de la comparaison, de l'itération et du tri des éléments.

23.1 Méthodes Fondamentales des Collections (Disponibles pour la Majorité des Collections)

Les méthodes suivantes proviennent de l'interface `Collection<E>` et sont héritées par **toutes** les principales collections à l'exception de `Map` (qui possède sa propre famille d'opérations).

> **Note**
>
> `Map` n'implémente pas `Collection`, mais ses vues `keySet()`, `values()` et `entrySet()` **l'implémentent**, et exposent donc ces opérations partagées.

23.1.1 Opérations de Mutation

- `boolean add(E e)` — Ajoute un élément (les listes autorisent les doublons).
- `boolean remove(Object o)` — Supprime le premier élément correspondant.
- `void clear()` — Supprime tous les éléments.
- `boolean addAll(Collection<? extends E> c)` — Insertion bulk.
- `boolean removeAll(Collection<?> c)` — Supprime tous les éléments contenus dans la collection fournie.
- `boolean retainAll(Collection<?> c)` — Conserve uniquement les éléments correspondants.

23.1.2 Opérations de Requête

- `int size()` — Nombre d'éléments.
- `boolean isEmpty()` — Indique si la collection contient zéro élément.
- `boolean contains(Object o)` — S'appuie sur les règles d'égalité des éléments.
- `Iterator<E> iterator()` — Retourne un itérateur (fail-fast).
- `Object[] toArray()` et `<T> T[] toArray(T[] a)` — Copie dans un tableau.

23.2 Égalité

Une implémentation personnalisée de la méthode `equals()` permet de comparer le type et le contenu de deux collections.

L'implémentation diffère selon que l'on traite des List ou des Set.

- Exemple

```
List<Integer> firstList = List.of(10, 11, 22);
List<Integer> secondList = List.of(10, 11, 22);
List<Integer> thirdList = List.of(22, 11, 10);

System.out.println("firstList.equals(secondList): " + firstList.equals(secondList));
System.out.println("secondList.equals(thirdList): " + secondList.equals(thirdList));

Set<Integer> firstSet = Set.of(10, 11, 22);
Set<Integer> secondSet = Set.of(10, 11, 22);
Set<Integer> thirdSet = Set.of(22, 11, 10);

System.out.println("firstSet.equals(secondSet): " + firstSet.equals(secondSet));
System.out.println("secondSet.equals(thirdSet): " + secondSet.equals(thirdSet));
```

Sortie

```
firstList.equals(secondList): true
secondList.equals(thirdList): false
firstSet.equals(secondSet): true
secondSet.equals(thirdSet): true
```

> **Note**
>
> - Les List comparent la taille, l'ordre et l'égalité des éléments un par un.
> - Les Set comparent uniquement la taille et l'appartenance — l'ordre de parcours est sans importance.
> - Deux sets contenant les mêmes éléments logiques sont égaux même s'ils maintiennent des ordres d'itération internes différents.

23.3 Comportement Fail-Fast

La plupart des itérateurs de collections (à l'exception des collections concurrentes) sont `fail-fast` : modifier structurellement une collection pendant l'itération déclenche une `ConcurrentModificationException`.

```
List<Integer> list = new ArrayList<>(List.of(1,2,3));
for (Integer i : list) {
    list.add(99); // ❌ ConcurrentModificationException
}
```

> **Note**
>
> Utilisez `Iterator.remove()` lorsque vous devez supprimer des éléments pendant l'itération. Le comportement fail-fast **n'est pas garanti** — l'exception est levée selon un principe de best-effort. Vous ne devez pas compter sur sa capture pour assurer la correction du programme.

23.4 Opérations Bulk

- `removeIf(Predicate<? super E> filter)` — Supprime tous les éléments correspondants.
- `replaceAll(UnaryOperator<E> op)` — Remplace chaque élément.

- `forEach(Consumer<? super E> action)` — Applique une action à chaque élément.
- `stream()` — Retourne un stream pour les opérations de pipeline.

23.5 Types de Retour et Exceptions Courantes

- `add(E)` retourne **boolean** — toujours `true` pour `ArrayList`, peut être `false` pour les `Set` si aucune modification n'a lieu.
- `remove(Object)` retourne boolean (pas l'élément supprimé).
- `get(int)` lève `IndexOutOfBoundsException`.
- `iterator().remove()` lève `IllegalStateException` s'il est appelé deux fois sans `next()`.
- `toArray()` retourne toujours un `Object[]` — jamais un `T[]`.

23.6 Tableau de Synthèse — Opérations Partagées

Opération	S'applique à	Notes
`add(e)`	Toutes les collections sauf Map	Les List autorisent les doublons
`remove(o)`	Toutes les collections sauf Map	Supprime la première occurrence
`contains(o)`	Toutes les collections sauf Map	Utilise equals()
`size(), isEmpty()`	Toutes les collections	Temps constant pour la majorité
`iterator()`	Toutes les collections	Fail-fast
`clear()`	Toutes les collections	Supprime tous les éléments
`stream()`	Toutes les collections	Retourne un stream séquentiel
`removeIf(), replaceAll()`	List uniquement (la plupart des Set ne supportent pas replaceAll)	Opérations bulk
`toArray()`	Toutes les collections	Retourne Object[]

◄ 22. Introduction au Framework des Collections | ▲ Index | 24. Comparable, Comparator & Tri en Java ►

24. Comparable, Comparator & Tri en Java

Table des matières

Java fournit deux stratégies principales pour le tri et la comparaison : `Comparable` (ordre naturel) et `Comparator` (ordre personnalisé).

Comprendre leurs règles, leurs contraintes et leurs interactions avec les `generics` est essentiel.

- Pour les **types numériques**, le tri suit l'ordre numérique naturel, ce qui signifie que les valeurs plus petites précèdent les valeurs plus grandes.
- Le tri des **chaînes** suit l'ordre lexicographique (`code point` Unicode) : comparaison caractère par caractère ; les chiffres viennent avant les majuscules, les majuscules avant les minuscules.

Cet ordre est basé sur le `code point Unicode` de chaque caractère, et non sur une intuition alphabétique.

Un **Unicode code point** est une valeur numérique unique attribuée aux caractères dans le standard Unicode.

Plus précisément : un `Unicode code point` est un entier (écrit en hexadécimal sous la forme U+XXXX) qui représente un caractère, un symbole ou un caractère spécial spécifique indépendamment de la police, de la langue ou de la plateforme.

- Exemples :
    - U+0041 → A
    - U+0061 → a
    - U+0030 → 0
    - U+1F600 → 😀

Un code point n'est pas une séquence d'octets ; c'est un nombre abstrait.

La manière dont le code point est ensuite stocké en mémoire physique dépend de l'encodage (UTF-8, UTF-16, UTF-32).

Unicode définit les code points de U+0000 à U+10FFFF.

En bref : les Unicode code points définissent quel est le caractère ; les encodings définissent comment celui-ci est représenté en octets.

- Exemples d'ordre naturel

```
List<String> items = List.of("10", "2", "A", "Z", "a", "b");

List<String> sorted = new ArrayList<>(items);
Collections.sort(sorted);

System.out.println(sorted);
```

Sortie :

```
[10, 2, A, Z, a, b]
```

> **Note**
>
> L'ordre naturel est défini uniquement pour les types qui implémentent `Comparable`.

24.1 Comparable — Ordre Naturel

L'interface `Comparable<T>` définit l'ordre naturel d'un type.

Une classe l'implémente lorsqu'elle souhaite définir sa règle de tri par défaut.

24.1.1 Contrat de la Méthode Comparable

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Règles et valeur de retour :

- Retourne **négatif** → `this` < `other`
- Retourne **zéro** → `this` == `other`
- Retourne **positif** → `this` > `other`

> **Important**
>
> - L'ordre naturel doit être cohérent avec `equals()`, sauf si explicitement documenté autrement :
> - `compareTo()` est cohérent avec `equals()` si, et seulement si, `a.compareTo(b) == 0` et `a.equals(b) est true`.

> **Warning**
>
> compareTo peut lever une ClassCastException s'il reçoit un type non comparable — mais cela se produit généralement uniquement avec des types raw.

24.1.2 Exemple : Classe Implémentant Comparable

```
public class Person implements Comparable<Person> {

    private String name;
    private int age;

    public Person(String n, int a) {
        this.name = n;
        this.age = a;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }


}

var list = List.of(new Person("Bob", 40), new Person("Alice", 30));

list.stream().sorted().forEach(p -> System.out.println(p.getAge()));
```

La liste est triée par âge, car il s'agit de l'ordre numérique naturel.

24.1.3 Erreurs Courantes de Comparable

- Comparer tous les champs pertinents → résultats incohérents si ce n'est pas le cas
- Violer la transitivité → conduit à un comportement indéfini
- Lever des exceptions dans compareTo() casse le tri
- Ne pas implémenter la même logique que equals() → piège courant

---

24.2 Comparator — Ordre Personnalisé

L'interface `Comparator<T>` permet de définir plusieurs stratégies de tri sans modifier la classe elle-même.

24.2.1 Méthodes Principales de Comparator

```
int compare(T a, T b);
```

Méthodes de support supplémentaires :

24.2.1.1 Méthodes de Support Statiques de Comparator

Méthode	Statique / Instance	Type de Retour	Paramètres	Description
`Comparator.comparing(keyExtractor)`	statique	Comparator	Function<? super T, ? extends U>	Construit un comparator comparant les clés extraites en utilisant l'ordre naturel.
`Comparator.comparing(keyExtractor, keyComparator)`	statique	Comparator	Function<T,U>, Comparator	Construit un comparator comparant les clés extraites à l'aide d'un comparator personnalisé.
`Comparator.comparingInt(keyExtractor)`	statique	Comparator	ToIntFunction	Comparator optimisé pour les clés int (évite le boxing).
`Comparator.comparingLong(keyExtractor)`	statique	Comparator	ToLongFunction	Comparator optimisé pour les clés long.
`Comparator.comparingDouble(keyExtractor)`	statique	Comparator	ToDoubleFunction	Comparator optimisé pour les clés double.
`Comparator.naturalOrder()`	statique	Comparator	none	Comparator utilisant l'ordre naturel (Comparable).
`Comparator.reverseOrder()`	statique	Comparator	none	Ordre naturel inversé.
`Comparator.nullsFirst(comparator)`	statique	Comparator	Comparator	Enveloppe un comparator afin que les null soient comparés avant les non-null.
`Comparator.nullsLast(comparator)`	statique	Comparator	Comparator	Enveloppe un comparator afin que les null soient comparés après les non-null.

24.2.1.2 Méthodes d'Instance sur Comparator

Méthode	Statique / Instance	Type de Retour	Paramètres	Description
`thenComparing(otherComparator)`	instance	Comparator	Comparator	Ajoute un comparator secondaire lorsque le primaire compare comme égal.
`thenComparing(keyExtractor)`	instance	Comparator	Function<T,U>	Comparaison secondaire utilisant l'ordre naturel de la clé extraite.
`thenComparing(keyExtractor, keyComparator)`	instance	Comparator	Function<T,U>, Comparator	Comparaison secondaire avec un comparator personnalisé.
`thenComparingInt(keyExtractor)`	instance	Comparator	ToIntFunction	Comparaison numérique secondaire (optimisée).
`thenComparingLong(keyExtractor)`	instance	Comparator	ToLongFunction	Comparaison numérique secondaire.
`thenComparingDouble(keyExtractor)`	instance	Comparator	ToDoubleFunction	Comparaison numérique secondaire.
`reversed()`	instance	Comparator	none	Retourne un comparator inversé pour la même logique de comparaison.

24.2.2 Exemple de Comparator

```
var people = List.of(new Person("Bob",40), new Person("Ann",30));

Comparator<Person> byName = Comparator.comparing(Person::getName);

Comparator<Person> byAgeDesc = Comparator.comparingInt(Person::getAge).reversed();

var sorted = people.stream().sorted(byName.thenComparing(byAgeDesc)).toList();
```

24.3 Comparable vs Comparator

Caractéristique	Comparable	Comparator
Package	java.lang	java.util
Méthode	compareTo(T)	compare(T,T)
Type de Tri	Naturel (par défaut)	Personnalisé (stratégies multiples)
Modifie la Classe Source	OUI	NON
Utile Pour	Ordre par défaut	Ordre externe ou alternatif
Autorise Plusieurs Ordres	NON	OUI
Utilisé par Collections.sort	OUI	OUI
Utilisé par Arrays.sort	OUI	OUI

24.4 Tri des Tableaux et des Collections

24.4.1 Arrays sort()

```
int[] nums = {3,1,2};
Arrays.sort(nums); // ordre naturel

Person[] arr = {...};
Arrays.sort(arr); // Person doit implémenter Comparable
Arrays.sort(arr, byName); // en utilisant Comparator
```

24.4.2 Collections sort()

```
Collections.sort(list); // ordre naturel
Collections.sort(list, byName); // comparator
```

> **Note**
>
> Collections.sort(list) délègue à list.sort(comparator) depuis Java 8.

24.5 Tri Multi-Niveaux (thenComparing)

```
var cmp = Comparator
    .comparing(Person::getLastName)
        .thenComparing(Person::getFirstName)
            .thenComparingInt(Person::getAge);
```

24.6 Comparer les Primitifs Efficacement

```
Comparator.comparingInt(Person::getAge)
Comparator.comparingLong(...)
Comparator.comparingDouble(...)
```

> **Note**
>
> Ceux-ci évitent le boxing et sont préférés dans le code sensible aux performances.

24.7 Pièges Courants

- Trier une liste d'Object sans Comparable → ClassCastException à l'exécution
- compareTo incohérent avec equals → comportement imprévisible
- Comparator qui viole la transitivité → le tri devient indéfini
- Éléments null → sauf si le Comparator les gère, le tri lève une NPE
- Comparator comparant des champs de types mixtes → ClassCastException
- Utiliser la soustraction pour comparer des int peut provoquer un overflow → toujours utiliser `Integer.compare()`
- Trier une liste avec des éléments null et l'ordre naturel → NPE
- compareTo ne doit jamais retourner des valeurs négatives/zéro/positives incohérentes sur les mêmes deux objets (aucune aléatoire)

24.8 Exemple Complet

```
record Book(String title, double price, int year) {}

var books = List.of(
new Book("Java 17", 40.0, 2021),
new Book("Algorithms", 55.0, 2019),
new Book("Java 21", 42.0, 2023)
);

Comparator<Book> cmp =
Comparator
    .comparingDouble(Book::price)
        .thenComparing(Book::year)
            .reversed();

books.stream().sorted(cmp)
    .forEach(System.out::println);
```

> **Note**
>
> `reversed()` s'applique à l'ensemble du comparator composé, et non uniquement à la première clé de comparaison.

24.9 Résumé

- Utiliser `Comparable` pour l'ordre naturel (1 ordre par défaut).
- Utiliser `Comparator` pour des stratégies de tri flexibles ou multiples.
- Les comparators peuvent être composés (reversed, thenComparing).
- Le tri requiert une logique de comparaison cohérente.
- Arrays.sort et Collections.sort utilisent à la fois Comparable et Comparator.

◀ 23. Opérations Partagées des Collections & Égalité | ▲ Index | 25. L'API List ▶

25. L'API List

Table des matières

Dans le `Collections Framework`, une **List** représente une collection ordonnée, basée sur des indices, autorisant les doublons.

L'interface List étend `Collection` et est implémentée par :

```
List
├── ArrayList (Tableau redimensionnable — accès aléatoire rapide, insertions/suppressions
plus lentes au milieu)
├── LinkedList (Liste doublement chaînée — insertions/suppressions rapides, accès aléatoire
plus lent)
└── Vector (Liste synchronisée legacy — rarement utilisée aujourd'hui)
```

> **Note**
>
> Vector est legacy et synchronisé — à éviter sauf si explicitement requis.

25.1 Caractéristiques des List

- Ordonnées — les éléments préservent l'ordre d'insertion.
- Indexées — accessibles via `get(int)` et `set(int,E)`.
- Autorisent les doublons — `List` n'impose pas l'unicité.
- Peuvent contenir `null` — sauf si l'on utilise des implémentations spéciales.

25.2 Créer des List (Constructeurs)

25.2.1 Constructeurs de ArrayList

```
List<String> a1 = new ArrayList<>();
List<String> a2 = new ArrayList<>(50); // capacité initiale
List<String> a3 = new ArrayList<>(List.of("A", "B"));
```

> **Note**
>
> La capacité initiale n'est pas une taille. Elle décide seulement combien d'éléments le tableau interne peut contenir avant redimensionnement.

25.2.2 Constructeurs de LinkedList

```
List<String> l1 = new LinkedList<>();
List<String> l2 = new LinkedList<>(List.of("A", "B"));
```

> **Note**
>
> `LinkedList` implémente aussi `Deque`.

25.3 Méthodes Factory

25.3.1 `List.of()` (immuable)

```
List<String> list1 = List.of("A", "B", "C");
list1.add("X"); // ❌ UnsupportedOperationException
list1.set(0, "Z"); // ❌ UnsupportedOperationException
```

> **Note**
>
> Toutes les listes `List.of()` : - rejettent les `null` - sont immuables - lèvent `UOE` lors d'une modification structurelle

25.3.2 `List.copyOf()` (copie immuable)

```
List<String> src = new ArrayList<>();
src.add("Hello");

List<String> copy = List.copyOf(src); // instantané immuable
```

25.3.3 Arrays.asList() (liste à taille fixe)

```
String[] arr = {"A", "B"};
List<String> list = Arrays.asList(arr);

list.set(0, "Z"); // OK
list.add("X"); // ❌ UOE — la taille est fixe
```

> **Note**
>
> La liste est adossée au tableau : modifier l'un affecte l'autre.

25.4 Opérations Principales de List

25.4.1 Ajouter des Éléments

```
list.add("A");
list.add(1, "B"); // insère à l'index
list.addAll(otherList);
list.addAll(2, otherList);
```

25.4.2 Accéder aux Éléments

```
String x = list.get(0);
list.set(1, "NewValue");
```

> **Note**
>
> `get()` lève `IndexOutOfBoundsException` pour des index invalides.

Si vous essayez de `mettre à jour` un élément dans une List vide, même à l'index 0, vous obtenez une `IndexOutOfBoundsException`

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(5);
System.out.println(list.toString());
list.clear();
list.set(0, 2);
```

Sortie

```
[3, 5]
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for leng
```

> **Warning**
>
> Appeler get/set avec un index invalide lève IndexOutOfBoundsException

25.4.3 Supprimer des Éléments

```
list.remove(0); // remove(int index) — supprime par index ; remove(Object) — supprime le premi
list.remove("A"); // supprime la première occurrence
list.removeIf(s -> s.startsWith("X"));
list.clear();
```

25.4.4 Comportements et Caractéristiques Importants

Opération	Comportement	Exception(s)
`add(E)`	ajoute toujours à la fin	—
`add(int,E)`	décale les éléments vers la droite	IndexOutOfBoundsException
`get(int)`	temps constant pour ArrayList, linéaire pour LinkedList	IndexOutOfBoundsException
`set(int,E)`	remplace l'élément	IndexOutOfBoundsException
`remove(int)`	décale les éléments vers la gauche	IndexOutOfBoundsException
`remove(Object)`	supprime le premier élément égal	—

25.5 `contains()`, `equals()` et `hashCode()`

25.5.1 `contains()`

La Méthode `contains()` utilise `.equals()` sur les éléments.

25.5.2 Égalité des List

`List.equals()` effectue une comparaison élément par élément dans l'ordre.

```
List<String> a = List.of("A", "B");
List<String> b = List.of("A", "B");

System.out.println(a.equals(b)); // true
```

> **Note**
>
> - L'ordre compte.
> - Le type de liste ne compte PAS.

25.5.3 `hashCode()`

Calculé sur la base du contenu.

25.6 Itérer à Travers une List

25.6.1 Boucle For Classique

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

25.6.2 Boucle For Améliorée

```
for (String s : list) {
    System.out.println(s);
}
```

25.6.3 Iterator & ListIterator

```
Iterator<String> it = list.iterator();
while (it.hasNext()) { System.out.println(it.next()); }

ListIterator<String> lit = list.listIterator();
while (lit.hasNext()) {
    if (lit.next().equals("A")) lit.set("Z");
}
```

> **Warning**
>
> Tous les itérateurs standard de List sont fail-fast : une modification structurelle en dehors de l'itérateur cause ConcurrentModificationException.

> **Note**
>
> Seul `ListIterator` supporte la traversée bidirectionnelle et la modification.

25.7 La Méthode `subList()`

`subList()` crée une vue d'une portion de la liste, pas une copie. Modifier l'une des deux peut modifier l'autre.

25.7.1 Syntaxe

```
List<E> subList(int fromIndex, int toIndex);
```

25.7.2 Règles

Règle	Explication
fromIndex inclusif	l'élément à fromIndex est inclus
toIndex exclusif	l'élément à toIndex n'est PAS inclus
La vue est adossée à la liste originale	modifier l'une modifie l'autre
Modification structurelle du parent invalide la subList	→ ConcurrentModificationException

25.7.3 Exemples

```
List<String> list = new ArrayList<>(List.of("A", "B", "C", "D"));
List<String> view = list.subList(1, 3);
// view = ["B", "C"]

view.set(0, "X");
// list = ["A", "X", "C", "D"]
// view = ["X", "C"]
```

25.7.4 Modifier la liste parent invalide la vue

```
List<String> list = new ArrayList<>(List.of("A","B","C","D"));
List<String> view = list.subList(1, 3);

list.add("E"); // modification structurelle de la liste parent

view.get(0); // ❌ ConcurrentModificationException
```

25.7.5 Modifier la subList modifie le parent

```
view.remove(1);
// supprime "C" à la fois de la view et de la liste parent
```

25.7.6 Vider la subList vide une partie de la liste parent

```
view.clear();
// supprime les index 1 et 2 du parent
```

25.7.7 Pièges Courants

- Supposer que subList est indépendante : c'est une vue, pas une copie
- Supposer que subList permet le redimensionnement : fonctionne uniquement sur des listes parent modifiables
- Oublier que les modifications du parent invalident la vue entraînant ConcurrentModificationException
- Attentes d'index incorrectes : l'index de fin est exclusif

25.8 Tableau Résumé des Opérations Importantes

Opération	ArrayList	LinkedList	List Immuables
`add(E)`	rapide	rapide	❌ non supporté
`add(index,E)`	lent (shift)	rapide	❌
`get(index)`	rapide	lent	rapide
`remove(index)`	lent	lent (sauf si suppression du premier/dernier)	❌
`remove(Object)`	plus lent	plus lent	❌
`set(index,E)`	rapide	lent	❌
`iterator()`	rapide	rapide	rapide
`listIterator()`	rapide	rapide	rapide
`contains(Object)`	O(n)	O(n)	O(n)

◀ 24. Comparable, Comparator & Tri en Java | ▲ Index | 26. Set API ▶

26. Set API

Table des matières

Un **Set** en Java représente une collection qui **ne contient pas d'éléments dupliqués**.

Il modélise le concept mathématique d'`ensemble` : non ordonné (sauf si une implémentation ordonnée est utilisée) et composé de valeurs uniques.

Toutes les implémentations de Set reposent sur des **sémantiques d'égalité** (via `equals()` ou la logique de `Comparator`).

26.1 Hiérarchie des Set (Java Collections Framework)

```
Set<E>
 ├── SequencedSet<E> (Java 21+)
 │     └── LinkedHashSet<E>    (ordonné)
 ├── HashSet<E>                (non ordonné)
 └── SortedSet<E>
       └── NavigableSet<E>
             └── TreeSet<E>    (ordonné)
```

Toutes les implémentations de `Set` requièrent :

- l'unicité des éléments
- une égalité et un hashing prévisibles (selon l'implémentation)

> **Note**
>
> `LinkedHashSet` est désormais formellement un `SequencedSet` depuis Java 21.

26.2 Caractéristiques de Chaque Implémentation de Set

26.2.1 HashSet

- Set généraliste le plus rapide
- Non ordonné (aucune garantie sur l'ordre d'itération)
- Utilise `hashCode()` et `equals()`
- Autorise un seul élément `null`

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A");   // doublon ignoré
System.out.println(set); // ordre non garanti
```

26.2.2 LinkedHashSet

- Maintient l'**ordre d'insertion**
- Légèrement plus lent que HashSet
- Utile lorsqu'un ordre d'itération prévisible est requis

```
Set<String> set = new LinkedHashSet<>();
set.add("A");
set.add("C");
set.add("B");
System.out.println(set);  // [A, C, B]
```

26.2.3 TreeSet

Un Set **trié** dont l'ordre est déterminé par :

- L'ordre naturel (`Comparable`)
- Un `Comparator` fourni

TreeSet :

- N'autorise pas d'éléments `null` (NullPointerException à l'exécution)
- Garantit une itération triée
- Prend en charge des vues par plage : `headSet()` , `tailSet()` , `subSet()`

```
TreeSet<Integer> tree = new TreeSet<>();
tree.add(10);
tree.add(1);
tree.add(5);

System.out.println(tree); // [1, 5, 10]
```

> **Note**
>
> `TreeSet` exige que tous les éléments soient mutuellement comparables — mélanger des types non comparables produit une `ClassCastException` . Les opérations (add, remove, contains) sont en O(log n).

26.3 Règles d'Égalité dans les Set

Les règles diffèrent selon l'implémentation.

26.3.1 HashSet & LinkedHashSet

L'`unicité` est déterminée par deux méthodes :

- `hashCode()`
- `equals()`

Deux objets sont considérés comme le même élément si :

- Leurs hash codes correspondent
- Leur méthode `equals()` retourne `true`

> **Warning**
>
> Si vous modifiez un objet après l'avoir ajouté à un HashSet ou LinkedHashSet, son hashCode peut changer et le set peut perdre la référence à cet élément.

26.3.2 TreeSet

L'unicité est basée sur `compareTo()` ou sur le `Comparator` fourni.

Si `compare(a, b) == 0`, alors les objets sont considérés comme des doublons, même si `equals()` retourne false.

```
Comparator<String> comp = (a, b) -> a.length() - b.length();
Set<String> set = new TreeSet<>(comp);

set.add("Hi");
set.add("Yo"); // même longueur → traité comme doublon

System.out.println(set);  // ["Hi"]
```

26.4 Créer des Instances de Set

26.4.1 En Utilisant les Constructeurs

```
Set<String> s1 = new HashSet<>();
Set<String> s2 = new LinkedHashSet<>();
Set<String> s3 = new TreeSet<>();
```

26.4.2 Constructeurs de Copie

```
List<String> list = List.of("A", "B", "C");

Set<String> copy = new HashSet<>(list); // ordre perdu
System.out.println(copy);

Set<String> ordered = new LinkedHashSet<>(list); // conserve l'ordre de la liste
System.out.println(ordered);
```

26.4.3 Méthodes Factory

```
Set<String> s1 = Set.of("A", "B", "C");   // immuable
Set<String> empty = Set.of();             // set immuable vide
```

> **Note**
>
> Les set créés via les factory sont **immuables** : ajouter ou supprimer des éléments lève `UnsupportedOperationException`. `Set.of(...)` rejette les doublons à la création → IllegalArgumentException et rejette null → NullPointerException

---

26.5 Opérations Principales sur les Set

26.5.1 Ajouter des Éléments

```
set.add("A");          // retourne true si ajouté
set.add("A");          // retourne false si doublon
```

26.5.2 Vérifier l'Appartenance

```
set.contains("A");
```

26.5.3 Supprimer des Éléments

```
set.remove("A");
set.clear();
```

26.5.4 Opérations Bulk

```
set.addAll(otherSet);
set.removeAll(otherSet);
set.retainAll(otherSet); // intersection
```

26.6 Pièges Courants

- Utiliser TreeSet avec des objets non comparables → `ClassCastException`
- TreeSet n'utilise pas `equals()` : seul comparator/compareTo détermine l'unicité
- Utiliser des objets mutables comme clés de Set → casse les règles de hashing
- Les Set créés avec Set.of() sont immuables — la modification échoue
- HashSet ne garantit pas l'ordre d'itération
- TreeSet traite les objets avec compare()==0 comme des doublons même s'ils ne sont pas égaux

---

26.7 Tableau Récapitulatif

Implémentation	Conserve l'Ordre ?	Autorise Null ?	Trié ?	Logique Sous-jacente
HashSet	Non	Oui (1 null)	Non	hashCode + equals
LinkedHashSet	Oui (ordre d'insertion)	Oui (1 null)	Non	table de hachage + liste chaînée
TreeSet	Oui (trié)	Non	Oui (naturel/comparator)	compareTo / Comparator

---

◀ 25. L'API List | ▲ Index | 27. API Queue & Deque ▶

27. API Queue & Deque

Table des matières

Les interfaces `Queue` et `Deque` de Java modélisent des collections ordonnées conçues pour traiter des éléments dans une séquence particulière.

- Une **Queue** modélise typiquement une structure **FIFO** (First-In, First-Out).
- Une **Deque** (`double-ended queue`) permet l'insertion et la suppression aux deux extrémités, autorisant des comportements **FIFO** et **LIFO** dans une seule API.

27.1 Queue — Vue d'ensemble

L'interface `Queue` étend `Collection` et est couramment utilisée dans la programmation asynchrone, la distribution du travail, les algorithmes et le buffering.

Il existe deux familles de méthodes : celles qui **lèvent des exceptions** et celles qui **retournent des valeurs spéciales** (généralement `null`).

27.1.1 Méthodes Principales de Queue

Opération	Lève une Exception	Retourne une Valeur Spéciale	Description
Insertion	`add(e)`	`offer(e)`	Ajoute un élément ; `offer` est préféré pour les queues à capacité limitée
Suppression	`E remove()`	`E poll()`	Supprime et retourne la tête. `remove()` lève NoSuchElementException si la queue est vide, `poll()` retourne null
Lecture	`E element()`	`E peek()`	Retourne la tête sans la supprimer. `element()` lève NoSuchElementException si la queue est vide, `peek()` retourne null

27.1.2 Implémentations de Queue

Classes courantes implémentant `Queue` :

- `LinkedList` — non bornée, implémente aussi `Deque` et `List`.
- `ArrayDeque` — queue rapide basée sur un tableau redimensionnable ; ne peut pas stocker `null`.
- `PriorityQueue` — ordonne les éléments par ordre naturel ou comparator ; n'est pas FIFO.
- `ConcurrentLinkedQueue` — thread-safe, lock-free.

> **Note**
>
> `PriorityQueue` ne garantit pas que l'ordre d'itération corresponde à l'ordre de priorité.

> **Warning**
>
> La plupart des implémentations de Queue rejettent `null` car `null` est utilisé comme valeur de retour pour "vide".

27.2 Deque — Vue d'ensemble

`Deque` (double-ended queue) prend en charge l'insertion, la suppression et l'inspection à la fois en tête et en queue.

Elle est plus polyvalente qu'une Queue :

- FIFO (comportement de queue)
- LIFO (comportement de stack)
- Algorithmes bidirectionnels

27.2.1 Méthodes Principales de Deque

Opération	Avant	Arrière
Insertion	addFirst(e), offerFirst(e)	addLast(e), offerLast(e)
Suppression	removeFirst(), pollFirst()	removeLast(), pollLast()
Inspection	getFirst(), peekFirst()	getLast(), peekLast()

27.2.2 Implémentations de Deque

- `ArrayDeque` — implémentation recommandée pour un usage général (rapide, sans limite de capacité).
- `LinkedList` — complète mais plus lente en raison de la structure à nœuds.
- `ConcurrentLinkedDeque` — deque concurrente non bloquante.

> **Note**
>
> `Stack` est legacy ; utiliser `Deque` pour le comportement de stack (push/pop). Les opérations de queue de ArrayDeque et LinkedList (add/remove/peek) sont en O(1) amorti.

27.3 Utiliser une Queue

```
Queue<String> q = new LinkedList<>();

q.offer("A");
q.offer("B");
q.offer("C");

System.out.println(q.peek());   // A
System.out.println(q.poll());   // A
System.out.println(q.poll());   // B
System.out.println(q.poll());   // C
System.out.println(q.poll());   // null (queue vide)
```

27.4 Utiliser une Deque (comme Queue et comme Stack)

27.4.1 Exemple FIFO (Comportement Queue)

```
Deque<String> dq = new ArrayDeque<>();

dq.offerLast("A"); // enqueue
dq.offerLast("B");
dq.offerLast("C");

System.out.println(dq.pollFirst()); // A
System.out.println(dq.pollFirst()); // B
System.out.println(dq.pollFirst()); // C
```

27.4.2 Exemple LIFO (Comportement Stack)

```
Deque<String> stack = new ArrayDeque<>();

stack.push("A");
stack.push("B");
stack.push("C");

System.out.println(stack.pop()); // C
System.out.println(stack.pop()); // B
System.out.println(stack.pop()); // A
```

27.5 PriorityQueue — Queue Spéciale

`PriorityQueue` ordonne les éléments par **ordre naturel** ou via un `Comparator` fourni.

Caractéristiques importantes :

- Non FIFO — la tête est l'élément "le plus petit".
- L'ordre est garanti uniquement lors de la suppression, pas lors de l'itération.

- Les éléments `null` ne sont pas autorisés.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.offer(50);
pq.offer(10);
pq.offer(30);

System.out.println(pq.poll()); // 10
System.out.println(pq.poll()); // 30
System.out.println(pq.poll()); // 50
```

27.6 Blocking Queue (Bases)

Dans les environnements concurrents, le package `java.util.concurrent` fournit des types de queues bloquantes.

- `ArrayBlockingQueue` — tableau sous-jacent de taille fixe.
- `LinkedBlockingQueue` — optionnellement bornée.
- `PriorityBlockingQueue` — priority queue thread-safe.
- `DelayQueue` — éléments libérés après des délais.

> **Note**
>
> - BlockingQueue n'autorise jamais `null` .
> - put(e) — bloque jusqu'à ce qu'un espace soit disponible
> - take() — bloque jusqu'à ce qu'un élément soit disponible
> - BlockingQueue prend aussi en charge des opérations temporisées : offer(e, timeout), poll(timeout)

27.7 Pièges Courants

- Les méthodes de `Queue` et `Deque` existent en variantes "exception" et "valeur spéciale" — mémoriser lesquelles sont lesquelles.
- `ArrayDeque` ne peut pas stocker `null` — `null` est utilisé en interne.
- L'ordre d'itération de `PriorityQueue` n'est PAS trié.
- L'utilisation de `Stack` est déconseillée ; utiliser `Deque` à la place.
- Deque permet à la fois FIFO et LIFO et possède l'API **la plus complète**.

27.8 Tableau Récapitulatif

Interface	Comportement Typique	Null Autorisé ?	Implémentations Courantes	Notes
Queue	FIFO	Dépend	LinkedList, ArrayDeque, PriorityQueue	PriorityQueue non FIFO
Deque	FIFO + LIFO	Non (ArrayDeque)	ArrayDeque, LinkedList	Opérations complètes aux deux extrémités
PriorityQueue	Ordonnée par priorité	Non	PriorityQueue	Supprime d'abord l'élément le plus petit
BlockingQueue	FIFO thread-safe	Non	ArrayBlockingQueue, LinkedBlockingQueue	différences entre add/offer et put
ConcurrentLinkedQueue	FIFO lock-free	Non	ConcurrentLinkedQueue	Très rapide pour le multi-threading

---

◀ 26. Set API | ▲ Index | 28. Map API ▶

28. Map API

Table des matières

L'interface `Map` représente une collection de **paires clé–valeur**, où chaque clé est associée à au plus une valeur.

Contrairement aux autres types de collections, `Map` **n'étend pas** `Collection` et possède donc sa propre hiérarchie et ses propres règles.

28.1 Caractéristiques Fondamentales de Map

- Chaque clé est unique ; **les clés dupliquées écrasent la valeur précédente**
- Les valeurs peuvent être dupliquées
- Les Map ne prennent pas en charge l'accès positionnel (basé sur des indices)
- L'itération s'effectue via `keySet()`, `values()` ou `entrySet()`

> **Note**
>
> Une `Map` n'est pas une `Collection`, mais ses vues (keySet, values, entrySet) sont des collections.

28.2 Principales Implémentations de Map

Implémentation	Ordonnancement	Clés Null	Valeurs Null	Thread-Safe	Notes
`HashMap`	Aucun ordre	1	Plusieurs	Non	Rapide, la plus courante
`LinkedHashMap`	Ordre d'insertion	1	Plusieurs	Non	Itération prévisible
`TreeMap`	Triée par clé	Non	Plusieurs	Non	Les clés doivent être comparables
`Hashtable`	Aucun ordre	Non	Non	Oui	Legacy
`ConcurrentHashMap`	Aucun ordre	Non	Non	Oui	Adaptée à la concurrence

> **Note**
>
> L'ordre de `TreeMap` est déterminé soit par `Comparable`, soit par un `Comparator` fourni lors de la construction.

28.3 Créer des Map

Les `Map` peuvent être créées à l'aide de constructeurs ou de méthodes factory.

```
Map<String, Integer> map1 = new HashMap<>();
Map<String, Integer> map2 = new LinkedHashMap<>();
Map<String, Integer> map3 = new TreeMap<>();

Map<String, Integer> map4 = Map.of("A", 1, "B", 2);
Map<String, Integer> map5 = Map.ofEntries(
    Map.entry("X", 10),
    Map.entry("Y", 20)
);
```

> **Note**
>
> Les Map créées avec `Map.of(...)` et `Map.ofEntries(...)` sont **immuables**. Toute tentative de modification lève `UnsupportedOperationException`.

28.4 Opérations de Base sur les Map

Méthode	Description	Valeur de Retour
`put(k, v)`	Ajoute ou remplace une association	Valeur précédente ou null
`putIfAbsent(k,v)`	Ajoute seulement si la clé est absente	Valeur existante ou null
`get(k)`	Retourne la valeur ou null	Valeur spécifique ou null
`getOrDefault(k, default)`	Retourne la valeur ou la valeur par défaut	Valeur spécifique ou défaut
`remove(k)`	Supprime l'association	Valeur supprimée ou null
`containsKey(k)`	Vérifie la présence de la clé	boolean
`containsValue(v)`	Vérifie la présence de la valeur	boolean
`size()`	Nombre d'entrées	int
`isEmpty()`	Test de vacuité	boolean
`clear()`	Supprime toutes les entrées	void
`V merge(k, v, BiFunction(V, V, V))`	merge(k, v, remappingFunction)	si clé absente → définit la valeur ; si clé présente → function(oldValue, newValue) ; si la fonction retourne null → association supprimée

```java
Map<String, String> map = new HashMap<>();
map.put("A", "Apple");
map.put("B", "Banana");

map.put("A", "Avocado"); // écrase la valeur

String v = map.get("B"); // Banana
```

28.5 Itérer sur une Map

Les Map sont parcourues via des vues :

- `keySet()` → Set de clés
- `values()` → Collection de valeurs
- `entrySet()` → Set de Map.Entry

```java
for (String key : map.keySet()) {
    System.out.println(key);
}

for (String value : map.values()) {
    System.out.println(value);
}

for (Map.Entry<String, String> e : map.entrySet()) {
    System.out.println(e.getKey() + " = " + e.getValue());
}
```

> **Note**
>
> Modifier la map pendant l'itération sur ces vues peut lever `ConcurrentModificationException` (sauf pour les map concurrentes).

28.6 Déterminer l'Égalité dans les Map

L'égalité des map est définie comme suit :

- Deux map sont égales si elles contiennent les mêmes associations clé–valeur
- La comparaison des clés utilise `equals()`
- La comparaison des valeurs utilise `equals()`

```java
Map<String, Integer> m1 = Map.of("A", 1, "B", 2);
Map<String, Integer> m2 = Map.of("B", 2, "A", 1);

System.out.println(m1.equals(m2)); // true
```

> **Note**
>
> L'ordre d'itération n'affecte pas l'égalité des map.

28.7 Comportement Spécial de TreeMap

TreeMap maintient les entrées triées selon les clés.

```java
Map<Integer, String> tm = new TreeMap<>();
tm.put(3, "C");
tm.put(1, "A");
tm.put(2, "B");

System.out.println(tm); // {1=A, 2=B, 3=C}
```

> **Warning**
>
> Toutes les clés d'une `TreeMap` doivent être mutuellement comparables. Mélanger des types incompatibles provoque une `ClassCastException` à l'exécution.

28.8 Gestion des Null

Implémentation	Clé Null	Valeur Null
HashMap	Oui (1)	Oui
LinkedHashMap	Oui (1)	Oui
TreeMap	Non	Oui
Hashtable	Non	Non
ConcurrentHashMap	Non	Non

> **Note**
>
> `TreeMap` accepte les valeurs `null` uniquement lorsqu'elles ne participent pas à la comparaison des clés. En pratique, cela est rare, car les clés null sont interdites et les comparators peuvent rejeter les null.
>
> `HashMap` et `LinkedHashMap` autorisent une seule clé null — en insérer une autre remplace l'existante.

28.9 Pièges Courants

- Supposer que Map est une Collection
- Oublier que les clés dupliquées écrasent les valeurs
- Utiliser des clés null dans TreeMap ou ConcurrentHashMap
- Confondre l'ordre d'itération avec l'égalité
- Tenter de modifier des map immuables créées via Map.of

28.10 Résumé

- Les Map stockent des clés uniques associées à des valeurs
- L'ordonnancement dépend de l'implémentation
- L'égalité est basée sur les paires clé–valeur
- TreeMap exige des clés comparables
- Les map immuables lèvent des exceptions en cas de modification

---

◀ 27. API Queue & Deque | ▲ Index | 29. Collections Séquencées & Map Séquencées ▶

29. Collections Séquencées & Map Séquencées

Table des matières

Java 21 introduit les `Collections Séquencées` et les `Map Séquencées` afin d'unifier et de formaliser l'accès aux éléments en fonction de leur ordre d'apparition.

Cet ajout résout des incohérences de longue date entre listes, sets, queues, deques et map, en fournissant une API commune pour travailler avec le premier et le dernier élément, ainsi qu'avec des vues inversées.

29.1 Motivation et Contexte

Avant Java 21, les collections ordonnées (telles que List, LinkedHashSet, Deque ou LinkedHashMap) exposaient les opérations liées à l'ordre via des méthodes différentes ou, dans certains cas, pas du tout.
Les développeurs devaient s'appuyer sur des API spécifiques à l'implémentation ou sur des contournements indirects.

Les interfaces séquencées introduisent un contrat cohérent pour toutes les collections et map ordonnées, rendant les opérations basées sur l'ordre explicites, sûres et uniformes.

29.2 Interface SequencedCollection

`SequencedCollection<E>` est une nouvelle interface qui étend `Collection<E>` et représente des collections avec un ordre d'apparition bien défini.

Elle est implémentée par `List`, `Deque` et `LinkedHashSet` (`TreeSet` est ordonné mais n'implémente pas directement SequencedCollection).

29.2.1 Méthodes Principales de SequencedCollection

L'interface définit des méthodes pour accéder et manipuler les éléments aux deux extrémités de la collection.

Méthode	Description
`E getFirst()`	Retourne le premier élément
`E getLast()`	Retourne le dernier élément
`void addFirst(E e)`	Insère un élément au début
`void addLast(E e)`	Insère un élément à la fin
`E removeFirst()`	Supprime et retourne le premier élément
`E removeLast()`	Supprime et retourne le dernier élément
`SequencedCollection<E> reversed()`	Retourne une vue inversée

29.2.2 Implémentations de SequencedCollection

Les types standards suivants implémentent SequencedCollection :

Type	Notes
List	Ordonnée par index
Deque	File à double extrémité
LinkedHashSet	Maintient l'ordre d'insertion

29.2.3 Vues Inversées

L'appel à `reversed()` ne crée pas de copie.

Il retourne une vue live de la même collection avec l'ordre inversé.

```
List<Integer> list = new ArrayList<>(List.of(1, 2, 3));
SequencedCollection<Integer> rev = list.reversed();

rev.removeFirst(); // supprime 3
System.out.println(list); // [1, 2]
```

> **Note**
>
> Les vues inversées partagent la même collection sous-jacente. Les modifications structurelles dans l'une ou l'autre vue affectent l'autre : modifier soit la collection originale soit la vue inversée a un effet sur les deux.

29.3 Interface SequencedMap

`SequencedMap<K,V>` étend `Map<K,V>` et représente des map avec un ordre d'apparition des entrées bien défini.

Elle standardise des opérations qui n'existaient auparavant que dans des implémentations spécifiques comme `LinkedHashMap`.

29.3.1 Méthodes Principales de SequencedMap

Méthode	Description
`Entry<K,V> firstEntry()`	Première entrée de la map
`Entry<K,V> lastEntry()`	Dernière entrée de la map
`Entry<K,V> pollFirstEntry()`	Supprime et retourne la première entrée, ou null si vide
`Entry<K,V> pollLastEntry()`	Supprime et retourne la dernière entrée, ou null si vide
`SequencedMap<K,V> reversed()`	Vue inversée de la map

29.3.2 Implémentations de SequencedMap

Actuellement, la principale implémentation standard est :

Type	Ordonnancement
`LinkedHashMap`	Ordre d'insertion (ou ordre d'accès si configuré)

> **Note**
>
> LinkedHashMap peut réordonner les entrées lors de la lecture si elle est construite avec `accessOrder=true`.
>
> Dans ce cas, « première » et « dernière » reflètent l'ordre d'accès le plus récent.

29.3.3 Map Inversées

Comme pour les collections, `reversed()` sur une map séquencée retourne une vue, et non une copie.

```
SequencedMap<String, Integer> map =
new LinkedHashMap<>(Map.of("A", 1, "B", 2, "C", 3));

SequencedMap<String, Integer> rev = map.reversed();

rev.pollFirstEntry(); // supprime C=3
System.out.println(map); // {A=1, B=2}
```

> **Note**
>
> Comme pour SequencedCollection, `reversed()` retourne une vue live — les mutations s'appliquent aux deux map.

29.4 Relation avec les API Existantes

Les interfaces séquencées ne remplacent pas les types de collections existants.

Elles se placent au-dessus d'eux dans la hiérarchie et unifient les comportements communs.

Toutes les collections ordonnées existantes bénéficient automatiquement de ces API sans casser la rétrocompatibilité.

29.4.1 Quels Types Built-in Sont Séquencés

Le tableau suivant résume si les types standards de collections sont ordonnés et s'ils implémentent les nouvelles interfaces Sequenced.

Type	Ordonné ?	SequencedCollection ?	SequencedMap ?
`List`	✓ Oui	✓ Oui	✗ Non
`Deque`	✓ Oui	✓ Oui	✗ Non
`LinkedHashSet`	✓ Oui	✓ Oui	✗ Non
`TreeSet`	✓ Oui (trié)	✗ Non*	✗ Non
`HashSet`	✗ Non	✗ Non	✗ Non
`LinkedHashMap`	✓ Oui	✗ Non	✓ Oui
`HashMap`	✗ Non	✗ Non	✗ Non
`TreeMap`	✓ Oui (trié)	✗ Non	✗ Non

> **Note**
>
> `TreeSet` est ordonné, mais implémente `SortedSet` / `NavigableSet`, pas `SequencedCollection`.

29.5 Pièges Courants

- Les interfaces séquencées définissent des vues, pas des copies
- `reversed()` reflète les modifications de manière bidirectionnelle
- Toutes les implémentations de Set ou de Map ne sont pas séquencées
- HashSet et HashMap n'implémentent pas les interfaces séquencées
- L'ordre n'est garanti que lorsqu'il est explicitement défini
- Supprimer des éléments via un iterator sur la vue inversée impacte immédiatement l'ordre original

---

29.6 Résumé

- Les interfaces séquencées formalisent l'ordre d'apparition
- Elles fournissent un accès first/last et l'inversion
- Elles fonctionnent via des vues live, pas des copies
- Elles unifient les API entre listes, deque, set et map

◀ 28. Map API | ▲ Index | 30. Thread Java – Fondamentaux et Modèle d'Exécution ▶

Module 07

Concurrency and Threads

30. Thread Java – Fondamentaux et Modèle d'Exécution

Indice

Ce chapitre introduit les **thread** à partir des principes de base et explique comment ils sont modélisés et utilisés en Java 21.

Ce texte établit également les fondations conceptuelles nécessaires pour comprendre `concurrency`, `synchronization` et la `Java Concurrency API` traitée dans le prochain chapitre.

30.1 Thread, Processus et le Système d'Exploitation

Pour comprendre les thread, nous devons partir du modèle d'exécution du système d'exploitation.

Les systèmes d'exploitation modernes exécutent des programmes en utilisant des **processus** et des **thread**.

- **Processus**: Une instance de programme en exécution gérée par le système d'exploitation. Un processus possède son propre espace de mémoire virtuelle, des ressources système (fichiers, sockets) et au moins un thread.
- **Thread**: Une unité d'exécution légère à l'intérieur d'un processus. Les thread partagent la mémoire et les ressources du processus mais s'exécutent de manière indépendante.
- **Task**: Une unité logique de travail à exécuter. Un task peut être exécuté par un thread mais n'est pas lui-même un thread.
- **Core CPU**: Une unité d'exécution physique ou logique capable d'exécuter un thread à la fois. Plusieurs core permettent une véritable exécution parallèle.

Un seul processus peut contenir de nombreux thread, tous opérant dans le même environnement partagé. Cet environnement partagé est à la fois source des potentialités de la Concurrency et de ses risques.

30.2 Modèle de Mémoire: Stack et Heap

Les thread interagissent avec la mémoire de deux manières fondamentalement différentes.

- **Stack du Thread**: Zone de mémoire privée pour chaque thread. Elle stocke les frames des appels de méthode, les variables locales et l'état d'exécution. Chaque thread a exactement un stack.
- **Heap**: Zone de mémoire partagée utilisée pour les objets et les instances de classe. Tous les thread dans le même processus peuvent accéder au heap.

Puisque les `stack sont isolés` et le `heap est partagé`, les problèmes de concurrence surviennent lorsque plusieurs thread accèdent aux mêmes objets dans le heap sans coordination adéquate.

30.3 Contexte et Context Switching

Le système d'exploitation planifie l'exécution des thread sur les core de la CPU.

Puisque le nombre de thread exécutables dépasse souvent le nombre de core disponibles, le système d'exploitation effectue le **context switching**.

- **Contexte**: L'état complet d'exécution d'un thread, y compris registres, compteur de programme et pointeur de stack.
- **Context Switch**: L'acte de suspendre un thread et d'en reprendre un autre en sauvegardant et en restaurant leurs contextes respectifs.

Le `context switching` permet la concurrence mais a un coût: des cycles CPU sont consommés sans exécuter de logique applicative.

Les programmeurs Java doivent concevoir des systèmes qui équilibrent concurrence et efficacité.

30.4 Concurrency vs Parallelisme

Ces deux termes sont souvent confondus mais décrivent des concepts différents.

- **Concurrence**: Plusieurs thread sont en exécution dans le même intervalle de temps, éventuellement entrelacés sur un seul core CPU.
- **Parallelisme**: Plusieurs thread s'exécutent simultanément sur des core CPU différents.

Java supporte la concurrence indépendamment du parallélisme matériel.

Même sur un système single-core, les thread Java peuvent être concurrents via le time slicing.

30.5 Thread en Java: Modèle Conceptuel

En Java, un **thread** représente un chemin indépendant d'exécution à l'intérieur d'un seul processus JVM. Tous les thread Java opèrent dans le même heap et dans le même contexte de `class loading`, à moins qu'ils ne soient explicitement isolés via des mécanismes avancés.

- **Thread Java**: Un objet de type `java.lang.Thread` qui mappe à une unité d'exécution sous-jacente.
- **Runnable**: Une interface fonctionnelle qui représente un `task` dont la méthode `run()` contient la logique exécutable.

Un thread exécute du code en invoquant sa propre méthode `run()`, directement ou indirectement via le scheduler des thread de la JVM: voir Démarrer vs Exécuter un Thread

---

30.6 Catégories de Thread en Java 21

Java 21 définit différents types de thread, qui diffèrent par cycle de vie, scheduling et usage prévu.

- **Platform Thread**: Un thread Java traditionnel mappé un-à-un à un thread du système d'exploitation.
- **Virtual Thread**: Un thread léger géré par la JVM et schedulé sur des thread carrier. Introduit pour permettre une concurrence massive avec un overhead minimal.
- **Carrier Thread**: Un Platform Thread utilisé en interne par la JVM pour exécuter des thread virtuels.
- **Daemon Thread**: Un thread en arrière-plan qui n'empêche pas la terminaison de la JVM. Quand seuls des thread daemon restent en exécution, la JVM se termine.
- **Thread Utilisateur**: Tout thread non-daemon. La JVM attend que tous les thread utilisateur terminent avant de se terminer.
- **Thread Système**: Des thread créés en interne par la JVM pour le garbage collection, la compilation JIT et d'autres services runtime.

> **Note**
>
> - Les `virtual threads` sont des threads utilisateur légers ; ils ne sont **pas** daemon par défaut ;
> - Un VirtualThread (créé directement via `Thread.startVirtualThread()` ou `Thread.ofVirtual().start(...)`) accepte un Runnable comme tâche. Il n accepte pas directement un Callable : Si vous devez exécuter un Callable avec des virtual threads et récupérer un résultat, vous devez utiliser un ExecutorService ;
> - Les virtual threads sont implémentés par la classe `java.lang.VirtualThread`. Cette classe étend `BaseVirtualThread`, qui étend elle-même Thread. Par conséquent, un virtual thread est techniquement une sous-classe de Thread. Cependant, il n est pas exact de décrire un virtual thread comme une instance directe de la classe Thread, puisqu il est en réalité une instance d une sous-classe spécialisée conçue spécifiquement pour le comportement des virtual threads.

30.7 Créer des Thread en Java

Les thread peuvent être créés de différentes manières, toutes conceptuellement centrées sur la fourniture de logique exécutable.

- En étendant `Thread` et en redéfinissant `run()`.
- En passant un `Runnable` au constructeur de `Thread`.
- En utilisant des factories de thread et des executor (traités dans la section Concurrency API).

```java
Runnable runnable = ...

  // Crée un thread de plateforme via constructeur
  Thread thread = new Thread(runnable);
  thread.start();

  // Démarre un thread daemon pour exécuter un task
  Thread thread = Thread.ofPlatform().daemon().start(runnable);

  // Crée un thread non démarré nommé "duke", sa méthode start()
  // doit être invoquée pour planifier son exécution.
  Thread thread = Thread.ofPlatform().name("duke").unstarted(runnable);

  // Une ThreadFactory qui crée des thread daemon nommés "worker-0", "worker-1", ...
  ThreadFactory factory = Thread.ofPlatform().daemon().name("worker-", 0).factory();

  // Démarre un thread virtuel pour exécuter un task
  Thread thread = Thread.ofVirtual().start(runnable);

  // Une ThreadFactory qui crée des thread virtuels
  ThreadFactory factory = Thread.ofVirtual().factory();
```

> **Warning**
>
> - La seule création d'un thread ne démarre pas son exécution.
> - L'exécution commence seulement lorsque le scheduler de la JVM est impliqué.

30.8 Cycle de Vie et Exécution d'un Thread

Un thread Java traverse des états bien définis au cours de son cycle de vie.

- **New**: Objet thread créé mais pas encore démarré.
- **Runnable**: Éligible à l'exécution par le scheduler.
- **Running**: En exécution active sur un core CPU.
- **Blocked / Waiting**: Temporairement incapable de continuer à cause de synchronisation ou de coordination.
- **Terminated**: Exécution terminée ou interrompue.

La JVM et le système d'exploitation coopèrent pour déplacer les thread entre ces états.

Les thread dans l'état `BLOCKED`, `WAITING` ou `TIMED_WAITING` **n'utilisent pas de ressources CPU**

30.9 Démarrer vs Exécuter un Thread: Synchrone ou Asynchrone

Il existe une distinction conceptuelle critique entre invoquer `run()` et invoquer `start()`.

- Appeler directement `run()` exécute la méthode de manière synchrone dans le thread courant, comme un appel de méthode normal.
- Appeler `start()` demande à la JVM de créer un nouveau stack d'appel et d'exécuter `run()` de manière asynchrone dans un thread séparé.

Par conséquent, du code comme `new Thread(r).run();` NE crée PAS de concurrence. L'exécution reste synchrone et bloque le thread appelant jusqu'à l'achèvement.

> **Note**
>
> `Exécution asynchrone` signifie que l'appelant continue immédiatement tandis que le nouveau thread progresse de manière indépendante, soumis au scheduling.
>
> `Exécution synchrone` signifie que l'appelant attend que l'opération soit terminée.

> **Important**
>
> La concurrence commence **seulement** lorsque `start()` est invoqué.

30.10 Priorité des Thread et Scheduling

Les thread Java ont une priorité associée qui influence le scheduling.

- `Priorité du Thread` : Une valeur entière indiquant son importance relative, allant de minimum à maximum.
- `Scheduling` : La JVM délègue les décisions de scheduling au système d'exploitation, qui peut ou non respecter strictement les priorités.

La priorité du thread influence la probabilité de scheduling mais ne garantit jamais l'ordre d'exécution. Le code Java portable ne doit jamais dépendre des priorités pour la correction.

Il est possible de définir la **priorité** sur les `platform threads` ; pour les `thread virtuels` la **priorité** est toujours fixée à **5** (`Thread.NORM_PRIORITY`) et tenter de la modifier n'a aucun effet.

30.11 Différément et Yield des Thread

Les thread peuvent influencer volontairement le comportement de scheduling.

Appeler `Thread.yield()` signale la disponibilité à suspendre l'exécution.

- `Yielding` : Un thread suggère qu'il est disposé à suspendre l'exécution pour permettre à d'autres thread exécutables de progresser.
- `Sleeping` : Un thread suspend l'exécution pour une durée fixe, entrant dans un état d'attente temporisée.

Ces mécanismes ne garantissent pas l'exécution immédiate d'autres thread; ils fournissent seulement des suggestions de scheduling.

30.12 Interruption des Thread et Annulation Coopérative

Les thread Java ne peuvent pas être arrêtés de force depuis l'extérieur.

À la place, Java fournit un mécanisme coopératif appelé **interruption du thread**, qui permet à un thread de demander qu'un autre thread interrompe ce qu'il est en train de faire.

Le thread cible décide comment et quand répondre.

30.12.1 Ce que Signifie Interrompre un Thread

Interrompre un thread **ne** le termine **pas**. Appeler `interrupt()` définit un **flag d'interruption** interne sur le thread cible. Il est de la responsabilité du thread en exécution d'observer ce flag et de réagir de manière appropriée.

- `Demande d'interruption` : Un signal envoyé à un thread indiquant qu'il devrait s'arrêter ou changer son activité courante.
- `Flag d'interruption` : Un statut booléen associé à chaque thread, défini lorsque `interrupt()` est invoqué.
- `Annulation Coopérative` : Un design pattern dans lequel les thread vérifient périodiquement d'éventuelles interruptions et se terminent proprement.

30.12.2 Interrompre des Opérations Bloquantes

Certaines méthodes bloquantes en Java répondent immédiatement à l'interruption en lançant `InterruptedException` et en mettant à zéro le flag d'interruption. Ces méthodes incluent `sleep()`, `wait()` et `join()`.

Lorsqu'un thread est bloqué dans l'une de ces méthodes et qu'un autre thread l'interrompt, le thread bloqué est réveillé et une exception est lancée. Cela fournit un point de sortie sûr des opérations bloquantes.

30.12.3 Vérifier le Statut d'Interruption

Les thread qui ne sont pas bloqués doivent vérifier explicitement s'ils ont été interrompus. Java fournit deux façons de le faire.

- `Thread.currentThread().isInterrupted()` : Retourne le statut d'interruption sans le mettre à zéro.
- `Thread.interrupted()` : Retourne le statut d'interruption et le met à zéro. Ceci est subtil: l'appel suivant retournera false.

Ne pas vérifier le statut d'interruption peut amener les thread à ignorer des demandes d'annulation et à continuer à s'exécuter indéfiniment.

30.12.4 Exemple: Interrompre un Thread en Sleep

L'exemple suivant démontre l'annulation coopérative via interruption.

Un thread worker dort pendant qu'il exécute du travail. Le thread main l'interrompt, provoquant un shutdown propre.

```
class Main {

    static class Task implements Runnable {
        public void run() {
            try {
                while (true) {
                    System.out.println("Working...");
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("Task interrupted, shutting down");
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(new Task());
        worker.start();
        System.out.println("main before sleep...");
        Thread.sleep(3000);
        System.out.println("main after sleep...");
        worker.interrupt();
        System.out.println("main reached END");
    }
}
```

Output:

```
main before sleep...
Working...
Working...
Working...
main after sleep...
main reached END
Task interrupted, shutting down
```

> **Note**
>
> L'ordre de l'output peut varier légèrement à cause du scheduling.

30.12.5 Observations Clés

- Appeler `interrupt()` n'arrête pas directement le thread.
- L'interruption est détectée et `sleep()` lance une `InterruptedException`.
- Le thread worker se termine de lui-même de manière contrôlée.
- Une gestion correcte de l'interruption permet aux thread de libérer des ressources et de maintenir la cohérence du programme.

> **Note**
>
> Ignorer `InterruptedException` sans terminer ou restaurer le statut d'interruption est considéré comme une mauvaise pratique et peut mener à des thread non réactifs.

30.13 Thread et le Thread Principal

Chaque application Java commence avec un **thread principal**. Ce thread exécute la méthode `main(String[])`.

- Le thread principal est un thread utilisateur.
- La JVM reste active tant qu'au moins un thread utilisateur est en exécution.
- Si le thread principal se termine mais qu'il existe d'autres thread utilisateur, la JVM continue l'exécution en attendant que les thread utilisateur se terminent.
- Les thread daemon ne maintiennent pas la JVM en vie.

Comprendre le rôle du thread principal est essentiel pour raisonner sur la terminaison du programme et le traitement en arrière-plan.

30.14 Concurrency des Thread et État Partagé

La `Concurrence` naît lorsque plusieurs thread accèdent à un état mutable partagé.

- `État partagé` : Toute donnée située dans le heap accessible par plus d'un thread.
- `Race Condition` : Une erreur de correction causée par un accès non synchronisé à un état partagé.
- `Problème de visibilité` : Un thread opère sur des données obsolètes à cause de l'absence de synchronisation mémoire correcte.

Java résout ces problèmes avec synchronization, volatile, lock, atomiques et des frameworks de haut niveau (Executors, futures).

La synchronization, les variables volatile et les utilities de concurrence de haut niveau seront étudiées dans les sections suivantes.

30.15 Sommaire

- Les `Thread` sont le bloc de construction fondamental de l'exécution concurrente en Java.
- Ils existent à l'intérieur des processus, partagent la mémoire et sont schedulés par la JVM en coopération avec le système d'exploitation.
- Une gestion correcte des thread évite des fuites, des deadlocks et du gaspillage de CPU.

31. Java Concurrency APIs

Indice

Ce chapitre introduit la **Java Concurrency API**, qui fournit des abstractions de haut niveau pour gérer la concurrence de manière sûre, efficace et scalable.

À la différence de la manipulation de bas niveau des thread présentée dans le chapitre précédent, la Concurrency API se concentre sur **task**, **executor** et **mécanismes de coordination**, permettant aux programmeurs de raisonner sur ce qui doit être fait plutôt que sur la manière dont les thread sont schedulés.

31.1 Objectifs et Portée de la Concurrency API

La `Java Concurrency API`, principalement située dans le package `java.util.concurrent`, a été introduite pour affronter des problèmes fondamentaux inhérents à la gestion manuelle des thread.

- Séparer la soumission des task de la gestion des thread.
- Réduire la `synchronization` de bas niveau sujette à erreurs.
- Améliorer scalabilité et performance sur des systèmes multi-core.
- Fournir des mécanismes structurés pour `coordination`, `cancellation` et `shutdown`.

L'API n'élimine pas les problèmes de concurrence mais fournit des outils pour les gérer de manière sûre et prévisible.

Au lieu de créer et de contrôler explicitement les thread, les programmeurs exécutent des task et laissent le framework gérer **thread allocation, riuso, et synchronization**.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Task executed"));
executor.shutdown();
```

31.2 Problèmes Fondamentaux du Threading

Avant de comprendre la `Concurrency API`, il est essentiel de comprendre les problématiques de concurrence qu'elle veut atténuer.

Ces problèmes proviennent de `shared mutable state`, `scheduling unpredictability` et `improper coordination`.

31.2.1 Race Conditions

Une **race condition** se produit lorsque plusieurs thread accèdent à `shared mutable state` (un état mutable et partagé) et la correction du programme dépend du timing ou de l'intercalage de leur exécution.

- Causée par un accès non synchronisé à des données partagées.
- Conduit à un état du programme inconsistent ou incorrect.

```
class Counter {
    int count = 0;
    void increment() {
       count++;
    }
}
```

Si plusieurs thread invoquent `increment()` de manière concurrente, certains increments peuvent être perdus parce que l'opération n'est pas atomique.

31.2.2 Deadlock

Un **deadlock** se produit lorsque deux ou plusieurs thread sont bloqués de manière permanente, chacun en attente d'une ressource détenue par un autre thread.

- Typiquement causé par des dépendances circulaires entre lock.
- Aucun thread impliqué ne peut faire de progrès.

```
synchronized (lockA) {
    synchronized (lockB) {
    }
}
```

Si un autre thread acquiert d'abord `lockB` et ensuite attend `lockA`, un deadlock peut se produire.

> **Note**
>
> Les deadlock dans le monde réel impliquent typiquement des lock multiples et des inversions d'ordre.

31.2.3 Starvation

La **starvation** se produit lorsqu'un thread se voit refuser indéfiniment l'accès aux ressources, même si ces ressources sont disponibles.

- Souvent causée par `unfair locking` ou des policy de scheduling.
- Le thread reste `runnable` mais n'est jamais exécuté.

```
ReentrantLock lock = new ReentrantLock(false); // unfair lock
```

Certains thread peuvent acquérir de manière répétée le lock tandis que d'autres attendent indéfiniment.

31.2.4 Livelock

Dans un **livelock**, les thread ne sont pas bloqués mais réagissent continuellement l'un à l'autre d'une manière qui en empêche le progrès.

- Les thread restent actifs mais inefficaces.

- Souvent causé par une logique de retry ou d'avoidance agressive.

```
while (!tryLock()) {
    Thread.sleep(10);
}
```

Les deux thread peuvent répéter continuellement le retry, empêchant le forward progress.

31.3 Des Thread aux Task

La Concurrency API déplace le modèle de programmation de la gestion directe des **thread** vers la soumission de **task**.

Un **task** représente une unité logique de travail indépendante du thread qui l'exécute.

- **Runnable**: Représente un task qui ne retourne pas un résultat.
- **Callable**: Représente un task qui retourne un résultat et peut lancer des checked exceptions.

```
Runnable task = () -> System.out.println("Runnable task");
Callable<Integer> callable = () -> 42;
```

Cette abstraction permet aux task d'être réutilisés, schedulés de manière flexible et exécutés via des stratégies d'exécution différentes.

31.4 Executor Framework

L'**Executor Framework** est le cœur de la Concurrency API.

Il gère la création des thread, le riuso et l'exécution des task à travers une interface simple.

- **Executor**: Interface de base pour exécuter des task.
- **ExecutorService**: Étend Executor avec contrôle du lifecycle et gestion des résultats.
- **ScheduledExecutorService**: Supporte l'exécution de task delayed et périodiques.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(() -> System.out.println("Task 1"));
executor.execute(() -> System.out.println("Task 2"));
executor.shutdown();
```

31.4.1 Submitting Task et Futures

Les task soumis via `execute()` retournent `void`: c'est une méthode "fire-and-forget" qui ne retourne aucune information sur le résultat du task.

Les task soumis en utilisant `submit()` retournent un **Future**, qui représente le résultat d'une computation asynchrone.

Les deux méthodes sont utilisées pour soumettre du travail pour une exécution asynchrone.

```
Future<Integer> future = executor.submit(() -> 10 + 20);
Integer result = future.get();
```

Method	Description
void **execute(Runnable task)**	Exécute un task de manière asynchrone sans valeur de retour et sans `Future`.
Future<?> **submit(Runnable task)**	Exécute un task de manière asynchrone; aucun résultat n'est produit (`Future.get()` retourne `null`).
Future **submit(Callable task)**	Exécute un task de manière asynchrone et retourne un résultat de type `T`.
List<Future> **invokeAll(Collection<? extends Callable> tasks)**	Exécute tous les task et retourne un `Future` pour chacun, après que tous complètent.
T **invokeAny(Collection<? extends Callable> tasks)**	Exécute les task et retourne le résultat d'un qui complète avec succès; les autres sont annulés.

Method	Description
boolean **isDone()**	Retourne `true` si le task est terminé (normalement, exceptionnellement, ou via cancellation).
boolean **isCancelled()**	Retourne `true` si le task a été annulé avant la fin normale.
boolean **cancel(boolean mayInterruptIfRunning)**	Tente d'annuler l'exécution. Si `true`, interrompt le thread en exécution si possible.
T **get()**	Blocque jusqu'à la fin et retourne le résultat, ou lance une exception si échoué ou annulé.
T **get(long timeout, TimeUnit unit)**	Blocque jusqu'au timeout donné et retourne le résultat, ou lance `TimeoutException` si non terminé.

> **Warning**
>
> `execute()` rejettera les exceptions silencieusement à moins qu'elles ne soient gérées à l'intérieur du task.

31.4.2 Callable vs Runnable

Les deux interfaces représentent des task, mais avec des capacités différentes.

- `Runnable` : Aucune valeur de retour, ne peut pas lancer de checked exceptions.
- `Callable` : Retourne une valeur et supporte des checked exceptions.

```
Callable<String> c = () -> "done";
Runnable r = () -> System.out.println("done");
```

Pour une computation asynchrone orientée résultat, `Callable` est généralement préféré.

31.5 Thread Pools et Scheduling

Les executor gèrent des **thread pools**, qui réutilisent un nombre fixe ou dynamique de thread pour exécuter des task de manière efficace.

- **Fixed thread pool**: Limite la concurrence à un nombre fixe de thread.
- **Cached thread pool**: Croît et se réduit dynamiquement selon la demande: crée de nouveaux thread quand nécessaire mais réutilise des thread disponibles.
- **Single-thread executor**: Garantit l'exécution séquentielle des task.
- **Scheduled executor**: Supporte des task delayed et périodiques.

Méthode Factory	Type de Retour	Description
`Executors.newFixedThreadPool(int nThreads)`	ExecutorService	Crée un thread pool avec un nombre fixe de threads.
`Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`	ExecutorService	Identique à newFixedThreadPool mais avec un ThreadFactory personnalisé.
`Executors.newSingleThreadExecutor()`	ExecutorService	Crée un thread pool à un seul thread qui exécute les task de manière séquentielle.
`Executors.newSingleThreadExecutor(ThreadFactory threadFactory)`	ExecutorService	Executor à un seul thread avec un ThreadFactory personnalisé.
`Executors.newCachedThreadPool()`	ExecutorService	Crée un thread pool qui crée de nouveaux threads si nécessaire et réutilise ceux inactifs.
`Executors.newCachedThreadPool(ThreadFactory threadFactory)`	ExecutorService	Thread pool cached avec un ThreadFactory personnalisé.
`Executors.newSingleThreadScheduledExecutor()`	ScheduledExecutorService	Crée un scheduled executor à un seul thread.
`Executors.newSingleThreadScheduledExecutor(ThreadFactory threadFactory)`	ScheduledExecutorService	Scheduled executor à un seul thread avec ThreadFactory personnalisé.
`Executors.newScheduledThreadPool(int corePoolSize)`	ScheduledExecutorService	Crée un scheduled thread pool avec la taille core spécifiée.
`Executors.newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)`	ScheduledExecutorService	Scheduled thread pool avec ThreadFactory personnalisé.
`Executors.newWorkStealingPool()`	ExecutorService	Crée un work-stealing pool en utilisant le nombre de processeurs disponibles comme niveau de parallélisme.
`Executors.newWorkStealingPool(int parallelism)`	ExecutorService	Crée un work-stealing pool avec le niveau de parallélisme spécifié.
`Executors.newThreadPerTaskExecutor(ThreadFactory threadFactory)`	ExecutorService	Crée un executor qui démarre un nouveau thread pour chaque task.

Méthode Factory	Type de Retour	Description
`Executors.newVirtualThreadPerTaskExecutor()`	ExecutorService	Crée un executor qui démarre un nouveau virtual thread pour chaque task.

`Task scheduling` : les task soumis à un executor sont placés dans une file et récupérés par les threads du pool ; l'ordre d'exécution dépend de l'implémentation de l'executor, de la politique de file et de la disponibilité des threads. Dans un scheduled executor, les task sont ordonnés selon leur temps d'activation ; les task périodiques sont réinsérés dans la file après chaque exécution.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

scheduler.schedule(
    () -> System.out.println("Delayed"),
    2, TimeUnit.SECONDS);
```

Method	Description
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)	Planifie une action one-shot qui devient exécutable après le delay spécifié.
ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit)	Planifie un task one-shot qui retourne une valeur après le delay spécifié.
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Planifie une exécution périodique à fixed rate : chaque exécution est planifiée par rapport au temps initial ; si une exécution est retardée, les suivantes peuvent tenter de « rattraper ».
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Planifie une exécution périodique avec fixed delay : chaque exécution est planifiée par rapport à la fin de la précédente ; aucun comportement de rattrapage.

> **Important**
>
> Ne jamais créer des thread manuellement dans une boucle: utilise plutôt des pools ou des virtual threads.

31.6 Lifecycle et Terminaison de l'Executor

Les executor doivent être fermés explicitement pour libérer des ressources et permettre la terminaison de la JVM.

- **shutdown()**: Démarre un shutdown ordonné: complète les task en attente mais n'accepte pas de task supplémentaires.
- **close()**: (Java 19+) appelle shutdown() et attend que les task finissent, se comportant comme support try-with-resources pour ExecutorService.
- **shutdownNow()**: Tente un shutdown immédiat et interrompt les task en exécution.
- **awaitTermination()**: Attend la complétion ou un timeout.

```
executor.shutdown();
executor.awaitTermination(5, TimeUnit.SECONDS);
```

31.7 Stratégies de Thread Safety

La Concurrency API fournit de multiples stratégies complémentaires pour obtenir thread safety.

31.7.1 Synchronisation

La synchronisation impose `mutual exclusion` et `memory visibility` en utilisant un intrinsic lock (monitor) associé à un objet ou à une classe.

```
synchronized void increment() {
    count++;
}
```

Quand un thread entre dans une méthode synchronized:

- Il acquiert l'intrinsic lock de l'objet target (`this` pour les méthodes d'instance).
- Un seul thread à la fois peut détenir le même lock, empêchant une exécution concurrente.
- Quand la méthode se termine, le lock est libéré automatiquement.

La synchronization établit une **happens-before relationship** dans le Java Memory Model:

- Toutes les écritures faites dans la région synchronized sont flushées dans la mémoire principale quand le lock est libéré.
- Un thread qui acquiert le même lock ensuite est garanti de voir ces update.

La keyword synchronized peut être appliquée à:

- **Méthodes d'instance** (lock sur `this`)
- **Méthodes statiques** (lock sur l'objet `Class`)
- **Blocs** (lock sur un objet spécifique, permettant un contrôle plus fin)

> **Important**
>
> La synchronisation est simple mais peut réduire la scalabilité sous contention.

31.7.2 Variables Atomiques

Les `atomic classes` fournissent des opérations lock-free, thread-safe implémentées en utilisant des primitives CPU de bas niveau comme Compare-And-Swap (CAS).

```
AtomicInteger count = new AtomicInteger();
count.incrementAndGet();
```

31.7.2.1 Atomic classes

Atomic Class	Description
AtomicBoolean	Met à jour et lit de manière atomique une valeur `boolean`.
AtomicInteger	Met à jour et lit de manière atomique une valeur `int`.
AtomicLong	Met à jour et lit de manière atomique une valeur `long`.
AtomicReference	Met à jour et lit de manière atomique un reference à objet.
AtomicIntegerArray	Fournit des opérations atomiques sur les éléments d'un array `int`.
AtomicLongArray	Fournit des opérations atomiques sur les éléments d'un array `long`.
AtomicReferenceArray	Fournit des opérations atomiques sur les éléments d'un array de reference.
AtomicStampedReference	Met à jour de manière atomique un reference avec un integer stamp pour éviter des problèmes ABA.
AtomicMarkableReference	Met à jour de manière atomique un reference avec un boolean mark.

31.7.2.2 Méthodes Atomiques

Method	Description
get()	Retourne la valeur courante avec une sémantique volatile-read.
set(value)	Définit la valeur avec une sémantique volatile-write.
lazySet(value)	Définit éventuellement la valeur avec des garanties d'ordering plus faibles.
compareAndSet(expect, update)	Définit de manière atomique la valeur si la valeur courante est égale à la valeur attendue.
getAndSet(value)	Définit de manière atomique la valeur et retourne la valeur précédente.
incrementAndGet()	Incrémente de manière atomique la valeur et retourne le résultat mis à jour.
getAndIncrement()	Incrémente de manière atomique la valeur et retourne le résultat précédent.
decrementAndGet()	Décrémente de manière atomique la valeur et retourne le résultat mis à jour.
getAndDecrement()	Décrémente de manière atomique la valeur et retourne le résultat précédent.
addAndGet(delta)	Ajoute de manière atomique le delta donné et retourne le résultat mis à jour.
getAndAdd(delta)	Ajoute de manière atomique le delta donné et retourne le résultat précédent.

Variables Atomiques

:

- Exécutent des opérations uniques **atomiquement**
- Fournissent des **memory visibility guarantees** similaires à `volatile`
- Évitent le thread blocking, les rendant hautement scalables sous contention

Cependant, les atomic variables garantissent l'atomicité seulement pour des **opérations individuelles**.

Composer plusieurs opérations requiert quand même une synchronization externe.

Les Variables atomiques sont typiquement utilisées pour

:

- Counter et sequence generator
- Flag et state indicator
- Update à haut throughput et basse latence

31.7.3 Lock Framework

Le package `java.util.concurrent.locks` fournit des mécanismes de locking explicites qui offrent plus de flexibilité et de contrôle que synchronized.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Caractéristiques clés du Lock framework:

- Les lock doivent être acquis et libérés explicitement

- L'acquisition du lock peut être interruptible ou time-bounded
- Les lock peuvent être configurés avec une fairness policy (paramètre) quand l'ordering est requis (quand tu dois contrôler l'ordre dans lequel les thread tournent)
- Plusieurs objets Condition peuvent être associés à un seul lock

31.7.3.1 Lock implementations

Lock Implementation	Description
Lock	Interface core qui définit des opérations de lock explicites.
ReentrantLock	Lock reentrant de `mutual exclusion` avec fairness policy optionnelle.
ReadWriteLock	Interface qui définit des lock séparés de read et write.
ReentrantReadWriteLock	Fournit des lock séparés reentrant de read et write pour améliorer la scalabilité en lecture.
StampedLock	Lock qui supporte des modes optimistic, read et write locking (non-reentrant).

> **Warning**
>
> À la différence d'autres lock, StampedLock **n'est pas reentrant** — le réacquérir depuis le même thread cause un deadlock.

31.7.3.2 Common Lock methods

Method	Description
lock()	Acquiert le lock, bloquant indéfiniment jusqu'à disponibilité.
unlock()	Libère le lock; doit être appelé par le thread propriétaire.
tryLock()	Tente d'acquérir le lock immédiatement sans bloquer: retourne un boolean indiquant si le lock a été acquis avec succès
tryLock(long, TimeUnit)	Tente d'acquérir le lock dans le timeout donné.
lockInterruptibly()	Acquiert le lock à moins que le thread ne soit interrompu.
newCondition()	Crée une instance `Condition` pour une coordination fine-grained entre thread.

À la différence de synchronized, les lock ne sont pas libérés automatiquement, rendant essentiel l'usage correct de try/finally pour éviter des deadlock.

31.7.4 Coordination Utilities

Les coordination utilities permettent aux thread de coordonner des phases d'exécution sans protéger des données partagées via `mutual exclusion`.

D'autres coordination primitives incluent: - `CountDownLatch` - `Semaphore` - `Phaser`

```java
import java.util.concurrent.CyclicBarrier;

public class BarrierExample {

    private static final int THREAD_COUNT = 3;

    public static void main(String[] args) {

        CyclicBarrier barrier = new CyclicBarrier(
            THREAD_COUNT,
            () -> System.out.println("All threads reached the barrier. Proceeding...")
        );

        Runnable task = () -> {
            String name = Thread.currentThread().getName();
            try {
                System.out.println(name + " performing initial work");
                Thread.sleep((long) (Math.random() * 2000));

                // Wait for other threads
                System.out.println(name + " waiting at barrier");
                barrier.await();

                // Executed only after all threads reach the barrier
                System.out.println(name + " performing next phase");

            } catch (Exception e) {
                e.printStackTrace();
            }
        };

        for (int i = 1; i <= THREAD_COUNT; i++) {
            new Thread(task, "Worker-" + i).start();
        }
    }
}
```

Sample Output:

```
Worker-1 performing initial work
Worker-2 performing initial work
Worker-3 performing initial work
Worker-3 waiting at barrier
Worker-1 waiting at barrier
Worker-2 waiting at barrier
All threads reached the barrier. Proceeding...
Worker-3 performing next phase
Worker-1 performing next phase
Worker-2 performing next phase
```

Un `CyclicBarrier` :

- Bloque les thread jusqu'à ce qu'un nombre prédéfini de thread atteigne la barrier
- Libère simultanément tous les thread en attente une fois que la barrier est tripped
- Peut être réutilisé pour plusieurs cycles de coordination

Ces utilities se concentrent sur execution ordering et synchronization, pas sur data protection.

31.8 Concurrent Collections

Les concurrent collections sont des **thread-safe data structures** conçues pour supporter des **niveaux élevés de concurrence** sans exiger une synchronization externe.

À la différence des synchronized wrappers (ex. `Collections.synchronizedMap`), les concurrent collections: - Utilisent **fine-grained locking** ou des **lock-free techniques** - Permettent à plusieurs thread d'accéder et de modifier la collection simultanément - Scalent mieux sous contention

Des exemples communs incluent:

- **ConcurrentHashMap**
- Une concurrent map à hautes performance qui permet des lectures et des update concurrents en partitionnant l'état interne et en minimisant le lock contention.

- **CopyOnWriteArrayList**
- Une thread-safe list optimisée pour des scénarios avec **beaucoup de lectures et peu d'écritures**. Les opérations de write créent un nouvel array interne, permettant aux lectures de procéder sans locking.
- **BlockingQueue**
- Une queue conçue pour des **producer-consumer patterns**, où les thread peuvent se bloquer en attendant des éléments ou une capacité disponible.

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();
queue.put("item");    // blocks if the queue is full
queue.take();         // blocks if the queue is empty
```

Les blocking queue gèrent la synchronization en interne, simplifiant la coordination entre thread producer et consumer.

> **Caution**
>
> Les CopyOnWrite collections sont memory-expensive; les éviter dans des workload write-heavy.

31.9 Parallel Streams

Les `parallel streams` fournissent du **declarative data parallelism**, permettant que les opérations du stream soient exécutées de manière concurrente sur plusieurs thread avec des changements minimaux de code.

Caractéristiques clés: - Activés via `parallelStream()` ou `stream().parallel()` - Exécutés en interne en utilisant le **common ForkJoinPool** - Divisent automatiquement les données en chunk traités en parallèle

Les parallel streams fonctionnent mieux quand: - Les opérations sont **CPU-bound** - Les fonctions sont **stateless et non-blocking** - La source de données est suffisamment grande pour amortir l'overhead de la parallélisation

```
list.parallelStream()
    .map(x -> x * x)
    .forEach(System.out::println);
```

Puisque l'ordre d'exécution n'est pas garanti, les parallel streams devraient éviter: - Shared mutable state - Blocking I/O - Order-dependent side effects

> **Note**
>
> Utilise `forEachOrdered()` si un output déterministique est requis.

31.10 Relation avec Virtual Threads

En Java 21, l' `Executor framework` s'intègre de manière seamless avec **virtual threads**, permettant massive concurrency avec usage minimal de ressources.

```
ExecutorService executor =
Executors.newVirtualThreadPerTaskExecutor();

executor.submit(() -> blockingIO());
executor.close();
```

Cela permet au code blocking de scaler efficacement sans redessiner les API.

31.11 Sommaire

- La `Java Concurrency API` fournit une alternative robuste, scalable et plus sûre à la gestion manuelle des thread.

- Abstraire l'exécution, coordonner les task et offrir des utilities thread-safe permet aux développeurs de construire des systèmes concurrents à la fois performants et maintenables.
- Choisis l'outil juste: synchronized → locks → atomics → executors → virtual threads.

◀ 30. Thread Java – Fondamentaux et Modèle d'Exécution | ▲ Index | 32. Fondamentaux des fichiers et des chemins ▶

Module 08

Java I/O and NIO

32. Fondamentaux des fichiers et des chemins

Table des matières

Cette section se concentre sur `Path`, `File`, `Files` et les classes associées, en expliquant pourquoi elles existent, quels problèmes elles résolvent et quelles sont les différences entre les API legacy `java.io` et `NIO v.2` (nouvelles API d'E/S), avec une attention particulière à la sémantique du système de fichiers, à la résolution des chemins et aux idées fausses courantes.

32.1 Modèle conceptuel : système de fichiers, fichiers, répertoires, liens et cibles-d'E/S

Avant de comprendre les API d'E/S Java, il est essentiel de comprendre avec quoi elles interagissent.

Java I/O n'opère pas dans le vide : il interagit avec des abstractions de système de fichiers fournies par le système d'exploitation.

Cette section définit ces concepts indépendamment de Java, puis explique comment Java I/O les mappe et quels problèmes sont résolus.

32.2 Système de fichiers – L'abstraction globale

Un `système de fichiers` est un mécanisme structuré fourni par un système d'exploitation pour organiser, stocker, récupérer et gérer des données sur des dispositifs de stockage persistant.

Au niveau conceptuel, un système de fichiers résout plusieurs problèmes fondamentaux :

- Stockage persistant au-delà de l'exécution du programme
- Organisation hiérarchique des données
- Nommer et localiser les données
- Contrôle d'accès et permissions
- Garanties de concurrence et de cohérence

En Java NIO, un système de fichiers est représenté par l'abstraction `FileSystem`, généralement obtenue via `FileSystems.getDefault()` pour le système de fichiers du système d'exploitation.

Aspect	Signification
Persistance	Les données survivent à la terminaison de la JVM
Portée	Géré par le SE, pas par la JVM
Multiplicité	Plusieurs systèmes de fichiers peuvent exister
Exemples	Disk FS, ZIP FS, in-memory FS

> **Note**
>
> Java n'implémente pas de systèmes de fichiers ; il s'adapte aux implémentations fournies par le SE ou par des providers personnalisés.

32.3 Chemin – Localiser une entrée dans un système de fichiers

Un `chemin` est un localisateur logique, pas une ressource.

Il décrit où quelque chose se trouverait dans un système de fichiers, pas ce que c'est ni si cela existe.

Un `chemin` résout le problème de l'`addressing` :

- Identifie un emplacement
- Est interprété dans un système de fichiers spécifique
- Peut ou non correspondre à une entrée existante

Propriété	Path
Conscient de l'existence	Non
Conscient du type	Non
Immuable	Oui
Ressource du SE	Non

> **Note**
>
> En Java, `Path` représente des entrées potentielles du système de fichiers, pas des entrées réelles.

32.4 Fichiers – Conteneurs persistants de données

Un `fichier` est une entrée du système de fichiers dont le rôle principal est de stocker des données.

Le système de fichiers traite les fichiers comme des séquences de bytes opaques.

Problèmes résolus par les fichiers :

- Stockage durable d'informations
- Accès séquentiel et aléatoire aux données
- Partage des données entre processus

Du point de vue du système de fichiers, un fichier a :

- Contenu (bytes)
- Métadonnées (taille, timestamps, permissions)
- Un emplacement (chemin)

Aspect	Description
Contenu	Orienté byte
Interprétation	Définie par l'application
Durée de vie	Indépendante des processus
Accès Java	Streams, channels, méthodes de Files

> **Note**
>
> `Texte` vs `binaire` n'est pas un concept de système de fichiers ; c'est une interprétation au niveau application.

32.5 Répertoires – Conteneurs structurels

Un `répertoire (ou dossier)` est une entrée du système de fichiers dont le but est d'organiser d'autres entrées.

Les `répertoires` résolvent le problème de l'évolutivité et de l'organisation :

- Regrouper des entrées liées
- Permettre un nommage hiérarchique
- Supporter une recherche efficace

Aspect	Répertoire
Stocke des données	Non (stocke des références)
Contient	Fichiers, répertoires, liens
Lecture/écriture	Structurelle, pas basée sur le contenu
Accès Java	Files.list, Files.walk

> **Note**
>
> Un répertoire n'est pas un fichier avec du contenu, même si les deux partagent des métadonnées communes.

32.6 Liens – Mécanismes d'indirection

Un `lien` est une entrée du système de fichiers qui référence une autre entrée.

Les liens résolvent le problème de l'indirection et de la réutilisation.

32.6.1 Liens physiques

Un `lien physique` est un nom supplémentaire pour les mêmes données sous-jacentes.

- Plusieurs chemins pointent vers les mêmes données de fichier
- La suppression n'a lieu que lorsque tous les liens sont supprimés

32.6.2 Liens symboliques (Soft)

Un `lien symbolique` est un fichier spécial qui contient un chemin vers une autre entrée :

- Peut pointer vers des cibles inexistantes
- Résolu au moment de l'accès

Type de lien	Référence	Peut être dangling	Gestion Java
Physique	Données	Non	Transparent
Symbolique	Chemin	Oui	Contrôle explicite

> **Note**
>
> Java NIO expose le comportement des liens explicitement via `LinkOption`.
>
> Dans de nombreux systèmes de fichiers courants, le code Java ne peut pas créer des liens physiques de manière pleinement portable, tandis que les liens symboliques sont supportés directement via `Files.createSymbolicLink(...)` (là où autorisé par le SE / permissions).

32.7 Autres types d'entrées du système de fichiers

Certaines entrées du système de fichiers ne sont pas des conteneurs de données mais des endpoints d'interaction.

Type	But
Fichier de périphérique	Interface vers le matériel
FIFO / Pipe	Communication inter-processus
Fichier socket	Communication réseau

> **Note**
>
> Java I/O peut interagir avec ces entrées, mais le comportement dépend de la plateforme.

32.8 Comment Java IO interagit avec ces concepts

Les API Java I/O opèrent à différents niveaux d'abstraction :

- `Path` / `File` (API legacy) → décrit une entrée du système de fichiers
- `File` (API legacy) / `Files` → interroge ou modifie l'état du système de fichiers
- `Streams` / `Channels` → déplacent des bytes ou des caractères

API Java	Rôle
`Path`	Addressing
`File` (API legacy)	Addressing / opérations sur le système de fichiers
`Files`	Opérations sur le système de fichiers
`InputStream` / `Reader`	Lecture de données
`OutputStream` / `Writer`	Écriture de données
`Channel` / `SeekableByteChannel`	Avancé / accès aléatoire

> **Note**
>
> Aucune API Java "n'est" un fichier ; les API médiatisent l'accès à des ressources gérées par le système de fichiers.

32.9 Pièges conceptuels fondamentaux

- Confondre les chemins avec les fichiers
- Supposer que les chemins impliquent l'existence
- Supposer que les répertoires stockent les données des fichiers
- Supposer que les liens sont toujours résolus automatiquement

> **Note**
>
> Séparer toujours emplacement, structure et flux de données lorsqu'on raisonne sur les E/S.

32.10 Pourquoi Path et Files existent (contexte-IO)

Le classique `java.io` mélangeait trois préoccupations différentes dans des API mal séparées :

- Représentation du chemin (où se trouve la ressource ?)
- Interaction avec le système de fichiers (existe-t-elle ? quel type ?)
- Accès aux données (lecture/écriture de bytes ou de caractères)

La conception NIO.2 (Java 7+) sépare délibérément ces préoccupations :

- `Path` → décrit un emplacement
- `Files` → effectue des opérations sur le système de fichiers
- `Streams / Channels` → déplacent des données

> **Note**
>
> Un `Path` n'ouvre jamais un fichier et ne touche jamais le disque à lui seul.

32.11 File est (API legacy) à la fois un path et une api-d'opérations-sur-fichiers

Oui — dans l'ancienne API d'E/S, `java.io.File` joue de manière confuse deux rôles en même temps, et cette conception est exactement l'une des raisons pour lesquelles `java.nio.file` a été introduit.

Réponse courte

- `File` représente un chemin du système de fichiers
- `File` expose aussi des opérations sur le système de fichiers
- Il ne représente **ni** un fichier ouvert, **ni** le contenu du fichier

> **Note**
>
> Ce mélange de responsabilités est considéré comme un défaut de conception rétrospectivement.

32.11.1 Ce qu'est vraiment File

Conceptuellement, `File` est plus proche de ce que nous appelons aujourd'hui un `Path`, mais avec des méthodes opérationnelles ajoutées.

Aspect	java.io.File
Représente un emplacement	Oui
Ouvre un fichier	Non
Lit / écrit des données	Non
Interroge le système de fichiers	Oui
Modifie le système de fichiers	Oui
Contient un handle SE	Non

> **Note**
>
> Un objet `File` peut exister même si le fichier n'existe pas.

32.11.2 Responsabilités de type-Path

`File` se comporte comme une abstraction de chemin parce qu'il :

- Encapsule un pathname du système de fichiers (absolu ou relatif)
- Peut être résolu par rapport au répertoire de travail
- Peut être converti en forme absolue ou canonique

Exemples :

```
File f = new File("data.txt"); // chemin relatif
File abs = f.getAbsoluteFile(); // chemin absolu
File canon = f.getCanonicalFile(); // normalisé + résolu
```

32.11.3 Responsabilités d'opérations sur le système de fichiers

En même temps, `File` expose des méthodes qui touchent le système de fichiers :

- exists()
- isFile(), isDirectory()
- length()
- delete()
- mkdir(), mkdirs()
- list(), listFiles()

> **Note**
>
> La plupart de ces méthodes renvoient `boolean` au lieu de lancer `IOException`, ce qui masque les causes des échecs.

32.11.4 Ce que File N'EST PAS

- Ce n'est pas un file descriptor ouvert
- Ce n'est pas un stream
- Ce n'est pas un channel
- Ce n'est pas un conteneur de données du fichier

Il faut tout de même utiliser des streams ou des reader/writer pour accéder au contenu.

32.11.5 L'ancien double rôle

Le double rôle de `File` a causé plusieurs problèmes :

- Préoccupations mélangées (chemin + opérations)
- Mauvaise gestion des erreurs (boolean au lieu d'exceptions)
- Support faible pour les liens et les systèmes de fichiers multiples
- Comportement dépendant de la plateforme

32.11.6 Comment NIO a corrigé cela

NIO.2 sépare explicitement les responsabilités :

Responsabilité	Ancienne API	API NIO
`Représentation Path`	`File`	`Path`
`Opérations sur le système de fichiers`	`File`	`Files`
`Accès aux données`	Streams	Streams / Channels

> **Note**
>
> Cette séparation est l'une des améliorations conceptuelles les plus importantes en Java I/O.

32.11.7 Résumé

- `File` représente un chemin ET effectue des opérations sur le système de fichiers
- Il ne lit ni n'écrit jamais le contenu du fichier
- Il n'ouvre jamais un fichier
- `Path` + `Files` est le remplacement moderne

32.12 Path est une description, pas une ressource

Un `Path` est une abstraction pure représentant une séquence d'éléments de nom dans un système de fichiers.

- Il n'implique PAS l'existence
- Il n'implique PAS l'accessibilité
- Il ne contient PAS de file descriptor

Ceci est fondamentalement différent des streams ou des channels.

Concept	Path	Stream / Channel
`Ouvre ressource`	Non	Oui
`Touche disque`	Non	Oui
`Contient handle SE`	Non	Oui
`Immuable`	Oui	Non

> **Note**
>
> Créer un Path ne peut pas lancer `IOException` car aucun E/S ne se produit.

32.13 Chemins absolus vs relatifs

Comprendre la résolution des chemins est essentiel.

32.13.1 Chemins absolus

Un chemin absolu identifie complètement un emplacement depuis la racine du système de fichiers.

- Racine dépendante de la plateforme
- Indépendant du répertoire de travail de la JVM

Plateforme	Exemple de chemin absolu
Unix	`/home/user/file.txt`
Windows	`C:\Users\User\file.txt`

> **Important**
>
> - Un chemin commençant par un slash `(/)` (type Unix) ou par une lettre de drive telle que `C:` (Windows) est **typiquement** considéré comme un chemin absolu.
> - Le symbole `.` est une référence au répertoire courant tandis que `..` est une référence à son répertoire parent. Sur Windows, un chemin comme `\dir\file.txt` (sans lettre de drive) est *rooted* sur le drive courant, pas pleinement qualifié avec drive + chemin.

Exemple :

```
/dirA/dirB/../dirC/./content.txt

is equivalent to:

/dirA/dirC/content.txt

// in this example the symbols were redundant and unnecessary
```

32.13.2 Chemins relatifs

Un chemin relatif est résolu par rapport au répertoire de travail courant de la JVM.

- Dépend de l'endroit où la JVM a été lancée
- Source courante de bugs

> **Note**
>
> Le répertoire de travail est typiquement disponible via `System.getProperty("user.dir")`.

Exemple :

```
dirB/dirC/content.txt
```

32.14 Connaissance du système de fichiers et séparateurs

NIO introduit l'abstraction de système de fichiers, qui était largement absente dans java.io.

32.14.1 FileSystem

Un `FileSystem` représente une implémentation concrète spécifique de système de fichiers.

- Le système de fichiers par défaut correspond au système de fichiers du SE
- D'autres systèmes de fichiers possibles (ZIP, mémoire, réseau)

> **Note**
>
> Les chemins sont toujours associés à exactement UN FileSystem.

32.14.2 Séparateurs de chemin

Les séparateurs diffèrent selon les plateformes, mais `Path` les abstrait.

Aspect	java.io.File	java.nio.file.Path
Séparateur	Basé sur des chaînes	Conscient du système de fichiers
Portabilité	Gestion manuelle	Automatique
Comparaison	Sujette aux erreurs	Plus sûre

> **Note**
>
> Hardcoder `"/"` ou `"\\"` est déconseillé ; `Path` le gère automatiquement.

32.15 Ce que Files fait réellement et ce qu'il ne fait pas

La classe `Files` effectue de vraies opérations d'E/S.

32.15.1 Files FAIT

- Ouvre des fichiers indirectement (via streams / channels renvoyés par ses méthodes)
- Crée et supprime des entrées du système de fichiers
- Lance des exceptions checked en cas d'échec
- Respecte les permissions du système de fichiers

32.15.2 Files NE FAIT PAS

- Maintenir des ressources ouvertes après le retour de la méthode (sauf les streams)
- Stocker le contenu des fichiers en interne
- Garantir l'atomicité sauf si spécifié
- Maintenir un handle persistant vers des fichiers ouverts (les streams/channels possèdent le handle à la place)

> **Note**
>
> Les méthodes qui renvoient des streams (par ex. `Files.lines()`) gardent le fichier ouvert jusqu'à ce que le stream soit fermé.

32.16 Philosophie de gestion des erreurs : Old-vs-NIO

Une grande différence conceptuelle réside dans le reporting des erreurs.

Aspect	`java.io.File`	`java.nio.file.Files`
Signalement d'erreur	boolean / `null`	`IOException`
Diagnostic	Faible	Riche
Conscience des race	Faible	Améliorée
Préférence	Déconseillé	Préféré

32.17 Idées fausses courantes

- "Path représente un fichier" → faux
- "normalize vérifie l'existence" → faux
- "Files.readAllLines stream les données" → faux
- "Les chemins relatifs sont portables" → faux
- "Créer un Path peut échouer à cause des permissions" → faux

> **Note**
>
> De nombreuses méthodes NIO qui semblent "sûres" sont purement syntaxiques (comme `normalize` ou `resolve`) : elles ne touchent **pas** le système de fichiers et ne peuvent pas détecter des fichiers manquants.

◀ 31. Java Concurrency APIs | ▲ Index | 33. APIs des fichiers et des chemins ▶

33. APIs des fichiers et des chemins

Table des matières

Cette section se concentre sur la création de localisateurs de système de fichiers en utilisant l'API legacy `java.io.File` et l'API moderne `java.nio.file.Path` : comment convertir entre eux et comprendre les surcharges, les valeurs par défaut et les pièges courants.

33.1 `File` legacy et `Path` NIO : création et conversion

33.1.1 Créer un `File` (Legacy)

Une instance `File` représente un pathname du système de fichiers (absolu ou relatif).

En créer une n'accède pas au système de fichiers et ne lance pas `IOException`.

Constructeurs principaux (les plus courants) :

- `new File(String pathname)`
- `new File(String parent, String child)`
- `new File(File parent, String child)`
- `new File(URI uri)` (typiquement `file:...`)

```
import java.io.File;
import java.net.URI;

File f1 = new File("data.txt"); // relatif
File f2 = new File("/tmp", "data.txt"); // parent + enfant
File f3 = new File(new File("/tmp"), "data.txt");

File f4 = new File(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `new File(...)` n'ouvre jamais le fichier.
> - Existence/permissions sont vérifiées seulement lorsque vous appelez des méthodes comme `exists()`, `length()`, ou lorsque vous ouvrez un stream/channel.

33.1.2 Créer un `Path` (NIO v.2)

Un `Path` est aussi juste un locator.

Comme `File`, créer un `Path` n'accède pas au système de fichiers.

Factories principales :

- `Path.of(String first, String... more)` (Java 11+)
- `Paths.get(String first, String... more)` (style plus ancien ; toujours valide)
- `Path.of(URI uri)` (ex. `file:///...`)

```
import java.net.URI;
import java.nio.file.Path;
import java.nio.file.Paths;

Path p1 = Path.of("data.txt"); // relatif
Path p2 = Path.of("/tmp", "data.txt"); // parent + enfant

Path p3 = Paths.get("data.txt"); // style factory legacy
Path p4 = Path.of(URI.create("file:///tmp/data.txt"));
```

> **Note**
>
> - `Path.of(...)` et `Paths.get(...)` sont équivalents pour le système de fichiers par défaut.
> - Préférez `Path.of` dans le code moderne.

33.1.3 Absolu vs relatif : ce que signifie "relatif"

`File` et `Path` peuvent être créés comme chemins relatifs.

Les chemins relatifs sont résolus par rapport au répertoire de travail du processus (typiquement `System.getProperty("user.dir")`).

```java
import java.io.File;
import java.nio.file.Path;

File rf = new File("data.txt");
Path rp = Path.of("data.txt");

System.out.println(rf.isAbsolute()); // false
System.out.println(rp.isAbsolute()); // false

System.out.println(rf.getAbsolutePath());
System.out.println(rp.toAbsolutePath());
```

> **Note**
>
> Les chemins relatifs sont une source courante de bugs “works on my machine” parce que `user.dir` dépend de la manière/où la JVM a été lancée.

33.1.4 Joindre / construire des paths

- Le `File` legacy utilise des constructeurs (parent + enfant).
- NIO utilise `resolve` et des méthodes associées.

Tâche	Legacy (File)	NIO (Path)
Joindre parent + enfant	`new File(parent, child)`	`parent.resolve(child)`
Joindre plusieurs segments	Constructeurs répétés	`Path.of(a, b, c)` ou `resolve()` chaîné

```java
import java.io.File;
import java.nio.file.Path;

File f = new File("/tmp", "a.txt");

Path base = Path.of("/tmp");
Path p = base.resolve("a.txt"); // /tmp/a.txt
Path p2 = base.resolve("dir").resolve("a.txt"); // /tmp/dir/a.txt
```

33.1.4.1 `resolve()`

Combine des paths d’une manière filesystem-aware.

- Les paths relatifs sont ajoutés
- Un argument absolu remplace le path de base

> **Note**
>
> `Path.resolve(...)` a une règle : si l’argument est absolu, il retourne l’argument et ignore la base (vous ne pouvez pas combiner deux paths absolus en utilisant `resolve`).

33.1.4.2 `relativize()`

`Path.relativize` calcule un **path relatif** d’un path à un autre. Le path résultant, lorsqu’il est `resolved` contre le path source, donne le path cible.

Autrement dit :

- Il répond à la question : “Comment aller du path A au path B ?”
- Le résultat est toujours un path **relatif**
- Aucun accès au système de fichiers n’a lieu

Règles fondamentales

`relativize` a des préconditions strictes. Les violer lance une exception.

Règle	Explication
Les deux paths doivent être absolus	ou tous les deux relatifs
Les deux paths doivent appartenir au même système de fichiers	même provider
Les composants de root doivent correspondre	même root (sur Windows, même drive)
Le résultat n'est jamais absolu	toujours relatif

> **Note**
>
> Si un path est absolu et l'autre relatif, `IllegalArgumentException` est lancée.

Exemple relatif simple :

Les deux paths sont relatifs, donc la relativisation est autorisée.

```
Path p1 = Path.of("docs/manual");
Path p2 = Path.of("docs/images/logo.png");

Path relative = p1.relativize(p2);
System.out.println(relative);
```

```
../images/logo.png
```

Interprétation : depuis `docs/manual`, monter d'un niveau, puis entrer dans `images/logo.png`.

Exemple de paths absolus :

Les paths absolus fonctionnent exactement de la même manière.

```
Path base = Path.of("/home/user/projects");
Path target = Path.of("/home/user/docs/readme.txt");

Path relative = base.relativize(target);
System.out.println(relative);
```

```
../docs/readme.txt
```

Utiliser `resolve` pour vérifier le résultat

Une propriété clé de `relativize` est cette identité :

```
base.resolve(base.relativize(target)).equals(target)
```

```
Path base = Path.of("/a/b/c");
Path target = Path.of("/a/d/e");

Path r = base.relativize(target);
System.out.println(r); // ../../d/e
System.out.println(base.resolve(r)); // /a/d/e
```

Exemple : mélanger des paths absolus et relatifs (CAS ERREUR)

C'est l'une des erreurs les plus courantes.

```
Path abs = Path.of("/a/b");
Path rel = Path.of("c/d");

abs.relativize(rel); // lance une exception
```

```
Exception in thread "main" java.lang.IllegalArgumentException
```

> **Note**
>
> `relativize` NE tente PAS de convertir automatiquement des paths en absolus.

Exemple : roots différentes (piège spécifique Windows)

Sur Windows, des paths avec des lettres de drive différentes ne peuvent pas être relativisés.

```
Path p1 = Path.of("C:\\data\\a");
Path p2 = Path.of("D:\\data\\b");

p1.relativize(p2); // IllegalArgumentException
```

> **Note**
>
> Sur les systèmes Unix-like, la root est toujours `/`, donc ce problème ne se produit pas.

33.1.5 Convertir entre `File` et `Path`

La conversion est directe et sans perte pour les paths normaux du système de fichiers local.

Conversion	Comment
File → Path	`file.toPath()`
Path → File	`path.toFile()`

```
import java.io.File;
import java.nio.file.Path;

File f = new File("data.txt");
Path p = f.toPath();

File back = p.toFile();
```

> **Note**
>
> La conversion ne valide pas l'existence. Elle convertit seulement des représentations.

33.1.6 Conversion URI (quand nécessaire)

Les `URI` sont utiles lorsque les paths doivent être représentés dans une forme standard et absolue (par ex. interopérer avec des ressources réseau ou de la configuration).

Les deux APIs supportent la conversion URI.

Direction	Legacy (File)	NIO (Path)
Depuis URI	`new File(uri)`	`Path.of(uri)`
Vers URI	`file.toURI()`	`path.toUri()`

```java
import java.io.File;
import java.net.URI;
import java.nio.file.Path;

File f = new File("/tmp/data.txt");
URI u1 = f.toURI();

Path p = Path.of("/tmp/data.txt");
URI u2 = p.toUri();

Path pFromUri = Path.of(u2);
File fFromUri = new File(u1);
```

> **Note**
>
> `new File(URI)` requiert un URI `file:` et lance `IllegalArgumentException` si l'URI n'est pas hiérarchique ou n'est pas un file URI.

33.1.7 Canonique vs absolu vs normalisé (différences fondamentales)

Ces termes sont souvent mélangés. Ils ne sont pas identiques.

Concept	Legacy (File)	NIO (Path)	Touche le système de fichiers
Absolu	`getAbsoluteFile()`	`toAbsolutePath()`	Non
Normalisé	(pas de normalize pur, utiliser canonical)*	`normalize()`	`normalize()` : Non
Canonique / Réel	`getCanonicalFile()`	`toRealPath()`	Oui

> **Note**
>
> `File.getCanonicalFile()` et `Path.toRealPath()` peuvent résoudre des symlinks et exigent que le path existe, donc ils peuvent lancer `IOException`.
>
> File ne fournit pas de méthode pour une normalisation purement syntaxique : historiquement beaucoup de développeurs utilisaient getCanonicalFile(), mais ceci accède au système de fichiers et peut échouer.

```java
import java.io.File;
import java.io.IOException;
import java.nio.file.Path;

File f = new File("a/../data.txt");
System.out.println(f.getAbsolutePath()); // absolu, peut encore contenir ".."

try {
    System.out.println(f.getCanonicalPath()); // résout "..", peut toucher le système de fichi
} catch (IOException e) {
    System.out.println("Canonical failed: " + e.getMessage());
}

Path p = Path.of("a/../data.txt");
System.out.println(p.toAbsolutePath()); // absolu, peut encore contenir ".."
System.out.println(p.normalize()); // purement syntaxique

try {
    System.out.println(p.toRealPath()); // résout symlinks, exige l'existence
} catch (IOException e) {
    System.out.println("RealPath failed: " + e.getMessage());
}
```

33.1.7.1 `normalize()`

Supprime des éléments de nom **redondants** comme `.` et `..`.

- Purement syntaxique
- Ne vérifie pas si le path existe

> **Note**
>
> `normalize()` est purement syntaxique, ne vérifie pas l'existence, et peut produire des paths invalides s'il est mal utilisé.

33.1.8 Tableau de comparaison rapide (création + conversion)

Besoin	Legacy (File)	NIO (Path)	Préféré aujourd'hui
Créer depuis string	`new File("x")`	`Path.of("x")`	Path
Parent + enfant	`new File(p, c)`	`Path.of(p, c)` ou `resolve()`	Path
Convertir entre APIs	`toPath()`	`toFile()`	Path-centric
Normaliser	`getCanonicalFile()` (basé filesystem)	`normalize()` (syntactique seulement)	Path
Résoudre symlinks	Canonical	`toRealPath()`	Path

33.2 Gérer les fichiers et répertoires : créer, copier, déplacer, remplacer, comparer, supprimer (Legacy vs NIO)

Cette section couvre les opérations que vous effectuez sur des entrées du système de fichiers (fichiers/répertoires) : créer, copier, déplacer/renommer, remplacer, comparer et supprimer.

Elle contraste `java.io.File` legacy (et des helpers legacy associés) avec `java.nio.file` moderne (NIO.2).

33.2.1 Modèle mental : "Path/Locator" vs "Opérations"

Les deux APIs utilisent des objets qui représentent un path, mais les opérations diffèrent :

- Legacy : `File` est à la fois un wrapper de path et une API d'opérations (responsabilité mélangée)
- NIO : `Path` est le path ; `Files` effectue les opérations (séparation des préoccupations)

Responsabilité	Legacy	NIO
Représentation du path	`File`	`Path`
Opérations sur le système de fichiers	`File`	`Files`
Reporting riche des erreurs	Faible (booleans)	Fort (exceptions)

> **Note**
>
> Les méthodes legacy retournent souvent `boolean` (échec silencieux), tandis que NIO lance `IOException` avec cause.

33.2.2 Créer des fichiers et des répertoires

La création est là où l'ancienne API est la plus maladroite et l'API NIO la plus expressive.

Tâche	Approche legacy	Approche NIO	Notes
Créer fichier vide	ouvrir+fermer un stream	`Files.createFile`	NIO échoue si existe
Créer un répertoire	`mkdir`	`Files.createDirectory`	Le parent doit exister
Créer des répertoires récursivement	`mkdirs`	`Files.createDirectories`	Crée les parents

33.2.2.1 Créer un fichier

Legacy n'a pas de méthode "créer fichier vide", donc typiquement vous créez un fichier en ouvrant un output stream (side effect).

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

File f = new File("created-legacy.txt");
try (FileOutputStream out = new FileOutputStream(f)) {
    // le fichier est créé (ou tronqué) comme side effect
}
```

NIO fournit une méthode explicite de création.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path p = Path.of("created-nio.txt");
Files.createFile(p);
```

> **Note**
>
> `Files.createFile` lance `FileAlreadyExistsException` si l'entrée existe.

33.2.2.2 Créer des répertoires

```
import java.io.File;

File dir1 = new File("a/b");
boolean ok1 = dir1.mkdir(); // échoue si le parent "a" n'existe pas
boolean ok2 = dir1.mkdirs(); // crée les parents
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path d = Path.of("a/b");
Files.createDirectory(d); // le parent doit exister
Files.createDirectories(d); // crée les parents, ok si déjà existe
```

> **Note**
>
> Legacy `mkdir()/mkdirs()` retournent `false` en cas d'échec sans dire pourquoi. NIO lance `IOException`.

33.2.3 Copier des fichiers et des répertoires

La copie legacy est généralement une copie manuelle par stream (ou des libs externes). NIO a une opération unique et explicite.

Capacité	Legacy	NIO
Copier contenu de fichier	Streams manuels	`Files.copy`
Copier dans une cible existante	Manuel	Option `REPLACE_EXISTING`
Copier arbre de répertoires	Récursion manuelle	Récursion manuelle (mais meilleurs outils : `Files.walk` + `Files.copy`)

33.2.3.1 Copier un fichier (NIO)

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.io.IOException;

Path src = Path.of("src.txt");
Path dst = Path.of("dst.txt");

Files.copy(src, dst); // échoue si dst existe
Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.copy` lance `FileAlreadyExistsException` si la cible existe et que vous n'avez pas utilisé `REPLACE_EXISTING`.

33.2.3.2 Copie manuelle (Legacy, basée sur stream)

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

try (FileInputStream in = new FileInputStream("src.bin");
FileOutputStream out = new FileOutputStream("dst.bin")) {

    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) != -1) {
        out.write(buf, 0, n);
    }
}
```

> **Note**
>
> Rappelez-vous `read(byte[])` retourne le nombre de bytes lus ; vous devez écrire seulement ce compte, pas le buffer entier.

33.2.4 Déplacer / renommer et remplacer

Dans les deux APIs, rename/move est "au niveau metadata" quand possible, mais peut se comporter comme copy+delete entre systèmes de fichiers. NIO rend cela explicite via des options.

Opération	Legacy	NIO
Renommer/déplacer	`File.renameTo`	`Files.move`
Remplacer existant	Peu fiable	`REPLACE_EXISTING`
Déplacement atomique	Non supporté	`ATOMIC_MOVE` (si supporté)

33.2.4.1 Renommage legacy (piège commun)

```java
import java.io.File;

File from = new File("old.txt");
File to = new File("new.txt");

boolean ok = from.renameTo(to); // peut échouer silencieusement
System.out.println(ok);
```

> **Note**
>
> - `renameTo` est notoirement platform-dependent et retourne seulement `boolean`.
> - Il peut échouer parce que la cible existe, le fichier est ouvert, permissions, ou déplacement cross-filesystem.

33.2.4.2 NIO Move (préféré)

```java
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.io.IOException;

Path from = Path.of("old.txt");
Path to = Path.of("new.txt");

Files.move(from, to); // échoue si la cible existe
Files.move(from, to, StandardCopyOption.REPLACE_EXISTING);
```

> **Note**
>
> `Files.move` lance `FileAlreadyExistsException` quand la cible existe et que `REPLACE_EXISTING` n'est pas spécifié.

33.2.5 Comparer des paths et des fichiers

Comparer des locators peut signifier : égalité de string/path, égalité normalisée/canonique, ou "même fichier sur disque".

Les APIs diffèrent significativement ici.

Objectif de comparaison	Legacy	NIO
Même texte de path	`File.equals`	`Path.equals`
Normaliser le path	`getCanonicalFile`	`normalize`
Même fichier/ressource sur disque	faible (heuristique canonique)	`Files.isSameFile`

33.2.5.1 Égalité vs même fichier

Deux strings de path différentes peuvent référer au même fichier.

```java
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;

Path p1 = Path.of("a/../data.txt");
Path p2 = Path.of("data.txt");

System.out.println(p1.equals(p2)); // false (texte de path différent)
System.out.println(p1.normalize().equals(p2.normalize())); // peut encore être false si relati

try {
    System.out.println(Files.isSameFile(p1, p2)); // peut être true, peut lancer si non access
} catch (IOException e) {
    System.out.println("isSameFile failed: " + e.getMessage());
}
```

> **Note**
>
> `Files.isSameFile` peut accéder au système de fichiers et peut lancer `IOException` (problèmes de permissions, fichiers manquants, etc.).

33.2.6 Supprimer des fichiers et des répertoires

La suppression est simple conceptuellement mais a des edge cases importants : répertoires non vides, cibles manquantes, et différences de reporting d'erreurs.

Tâche	Legacy	NIO	Comportement si manquant
Supprimer fichier/dir	`File.delete`	`Files.delete`	Legacy false, NIO exception
Supprimer si existe	Pas direct (check+delete)	`Files.deleteIfExists`	retourne boolean
Supprimer dir non vide	Récursion manuelle	Récursion manuelle (walk)	Les deux exigent récursion

33.2.6.1 Delete legacy

```java
import java.io.File;

File f = new File("x.txt");
boolean ok = f.delete(); // false si non supprimé
System.out.println(ok);
```

> **Note**
>
> Legacy `delete()` échoue (retourne false) pour un répertoire non vide et souvent ne fournit aucune raison.

33.2.6.2 NIO Delete et Delete-If-Exists

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.nio.file.DirectoryNotEmptyException;
import java.io.IOException;

Path p = Path.of("x.txt");

try {
    Files.delete(p);
} catch (NoSuchFileException e) {
    System.out.println("Missing: " + e.getFile());
} catch (DirectoryNotEmptyException e) {
    System.out.println("Directory not empty: " + e.getFile());
} catch (IOException e) {
    System.out.println("Delete failed: " + e.getMessage());
}

boolean deleted = Files.deleteIfExists(p);
System.out.println(deleted);
```

> **Note**
>
> Certification tip: `Files.delete` lance `NoSuchFileException` si manquant, tandis que `deleteIfExists` retourne `false`.

33.2.7 Copier / supprimer récursivement des arbres de répertoires (pattern NIO)

NIO ne fournit pas une seule méthode “copyTree/deleteTree”, mais l’approche standard utilise `Files.walk` ou `Files.walkFileTree`.

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

Path root = Path.of("dirToDelete");

Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
        if (exc != null) throw exc;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});
```

> **Note**
>
> Supprimer un arbre de répertoires exige de supprimer d’abord les fichiers, puis les répertoires (post-order). C’est une question de raisonnement courante.

33.2.8 Checklist de résumé

- Préférer `Files.createFile/createDirectory/createDirectories` aux workarounds legacy
- `File.renameTo` est peu fiable ; préférer `Files.move` avec options
- `Files.copy/move` lancent `FileAlreadyExistsException` à moins que `REPLACE_EXISTING` soit utilisé
- `Files.delete` lance ; `Files.deleteIfExists` retourne boolean
- `Files.isSameFile` peut lancer `IOException` et peut toucher le système de fichiers

- La suppression de répertoires non vides exige une récursion (les deux APIs)

◄ 32. Fondamentaux des fichiers et des chemins | ▲ Index | 34. Streams I/O Java ►

34. Streams I/O Java

Table des matières

Ce chapitre fournit une explication détaillée des `flux I/O Java`.

Il couvre les flux classiques **java.io**, les compare à **java.nio / java.nio.file**, et explique les principes de conception, les API, les cas limites et les distinctions pertinentes.

34.1 Qu'est-ce qu'un flux I/O en Java ?

Un `flux I/O` représente un flux de données entre un programme Java et une source ou une destination externe.

Les données circulent de manière séquentielle, comme de l'eau dans un tuyau.

- Un flux n'est pas une structure de données ; il ne stocke pas les données de manière permanente
- Les flux sont unidirectionnels (entrée OU sortie)
- Les flux abstraient la source sous-jacente (fichier, réseau, mémoire, périphérique)
- Les flux fonctionnent de manière bloquante, synchrone (I/O classique)

En Java, les flux sont organisés autour de deux dimensions majeures :

- `Direction` : Entrée vs Sortie
- `Type de données` : Octets vs Caractères

34.2 Flux d'octets vs flux de caractères

Java distingue les flux selon l'unité de données qu'ils traitent.

34.2.1 Flux d'octets

- Fonctionnent avec des octets bruts 8 bits
- Utilisés pour les données binaires (images, audio, PDF, ZIP)
- Classes de base : `InputStream` et `OutputStream`

34.2.2 Flux de caractères

- Fonctionnent avec des caractères Unicode 16 bits
- Gèrent automatiquement l'encodage des caractères
- Classes de base : `Reader` et `Writer`

34.2.3 Tableau récapitulatif

Aspect	Flux d'octets	Flux de caractères
`Unité de données`	byte (8 bits)	char (16 bits)
`Gestion de l'encodage`	Aucune	Oui (conscient du charset)
`Classes de base`	InputStream / OutputStream	Reader / Writer
`Usage typique`	Fichiers binaires	Fichiers texte
`Focus`	I/O bas niveau	Traitement de texte

34.3 Flux de bas niveau vs flux de haut niveau

Les flux dans `java.io` suivent un pattern decorator. Les flux sont empilés pour ajouter des fonctionnalités.

34.3.1 Flux de bas niveau (Node Streams)

Les flux de bas niveau se connectent directement à une source ou un puits de données.

- Ils savent lire/écrire des octets ou des caractères
- Ils ne fournissent PAS de mise en tampon, de formatage ou de gestion d'objets

34.3.2 Flux de bas niveau courants

Classe de flux	Objectif
`FileInputStream`	Lire des octets depuis un fichier
`FileOutputStream`	Écrire des octets dans un fichier
`FileReader`	Lire des caractères depuis un fichier
`FileWriter`	Écrire des caractères dans un fichier

- Exemple : flux d'octets de bas niveau

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        System.out.println(b);
    }
}
```

> **Note**
>
> Les flux de bas niveau sont rarement utilisés seuls dans des applications réelles en raison de performances médiocres et de fonctionnalités limitées.

34.3.3 Flux de haut niveau (Filter / Processing Streams)

Les flux de haut niveau enveloppent d'autres flux pour ajouter des fonctionnalités.

- Mise en tampon
- Conversion de type de données
- Sérialisation d'objets
- Lecture/écriture de primitifs

34.3.4 Flux de haut niveau courants

Classe de flux	Ajoute des fonctionnalités
`BufferedInputStream`	Mise en tampon
`BufferedReader`	Lecture par lignes
`DataInputStream`	Types primitifs
`ObjectInputStream`	Sérialisation d'objets
`PrintWriter`	Sortie texte formatée

- Exemple : chaînage de flux

```
try (BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(
            new FileInputStream("text.txt")))) {

    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

34.3.5 Règles de chaînage des flux et erreurs courantes

L'exemple précédent illustre le chaînage des flux, un concept central de `java.io` basé sur le pattern decorator.

Chaque flux enveloppe un autre flux, ajoutant des fonctionnalités tout en préservant une hiérarchie de types stricte.

34.3.5.1 Règle fondamentale de chaînage

Un flux ne peut envelopper qu'un autre flux d'un niveau d'abstraction compatible.

- Les flux d'octets ne peuvent envelopper que des flux d'octets
- Les flux de caractères ne peuvent envelopper que des flux de caractères
- Les flux de haut niveau nécessitent un flux de bas niveau sous-jacent

> **Note**
>
> Vous ne pouvez pas mélanger arbitrairement `InputStream` avec `Reader` ou `OutputStream` avec `Writer`.

34.3.5.2 Incompatibilité flux d'octets vs flux de caractères

Une erreur très courante consiste à tenter d'envelopper un flux d'octets directement avec une classe basée sur les caractères (ou inversement).

34.3.5.3 Chaînage invalide (erreur de compilation)

```
BufferedReader reader =
    new BufferedReader(new FileInputStream("text.txt"));
```

> **Note**
>
> Cela échoue parce que `BufferedReader` attend un `Reader`, pas un `InputStream`.

34.3.5.4 Pont entre flux d'octets et flux de caractères

Pour convertir entre les flux basés sur les octets et ceux basés sur les caractères, Java fournit des classes passerelles qui effectuent un décodage/encodage explicite du charset.

- `InputStreamReader` convertit octets → caractères
- `OutputStreamWriter` convertit caractères → octets

34.3.5.5 Patron correct de conversion

```
BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(new FileInputStream("text.txt")));
```

> **Note**
>
> La passerelle gère le décodage des caractères en utilisant un charset (par défaut ou explicite).

34.3.5.6 Règles d'ordre dans les chaînes de flux

L'ordre d'enveloppement n'est pas arbitraire.

- Le flux de bas niveau doit être le plus interne
- Les passerelles (si nécessaires) viennent ensuite
- Les flux tamponnés ou de traitement viennent en dernier

34.3.5.7 Ordre logique correct

```
FileInputStream → InputStreamReader → BufferedReader
```

34.3.5.8 Règle de gestion des ressources

Fermer le flux le plus externe ferme automatiquement tous les flux enveloppés.

> **Note**
>
> C'est pourquoi try-with-resources devrait référencer uniquement le flux de plus haut niveau.

34.3.5.9 Pièges courants

- Essayer de tamponner un flux du mauvais type
- Oublier la passerelle entre flux d'octets et flux de caractères
- Supposer que `Reader` fonctionne avec des données binaires
- Utiliser le charset par défaut involontairement
- Fermer manuellement les flux internes (risque de double-close) : `close()` sur l'enveloppe externe suffit et est recommandé

34.4 Classes de base principales de `java.io` et méthodes clés

Le package `java.io` est construit autour d'un petit ensemble de **classes de base abstraites**. Comprendre ces classes et leurs contrats est essentiel, car toutes les classes I/O concrètes s'appuient sur elles.

34.4.1 InputStream

Classe de base abstraite pour l'entrée orientée octets. Tous les flux d'entrée lisent des octets bruts (valeurs 8 bits) depuis une source telle qu'un fichier, un socket réseau, ou un buffer mémoire.

34.4.1.1 Méthodes clés

Méthode	Description
int `read()`	Lit un octet (0–255) ; retourne -1 à la fin du flux
int `read(byte[])`	Lit des octets dans un buffer ; retourne le nombre d'octets lus ou -1
int `read(byte[], int, int)`	Lit jusqu'à length octets dans une tranche de buffer
int `available()`	Octets lisibles sans bloquer (indice, pas garantie)
void `close()`	Libère la ressource sous-jacente

> **Note**
>
> Les méthodes `read()` sont bloquantes par défaut.
>
> Elles suspendent le thread appelant jusqu'à ce que des données soient disponibles, que la fin de flux soit atteinte, ou qu'une erreur I/O se produise.

La méthode `read()` à octet unique est principalement un primitif de bas niveau.

En pratique, lire un octet à la fois est inefficace et devrait presque toujours être évité au profit de lectures tamponnées.

34.4.1.2 Exemple d'utilisation typique

```
try (InputStream in = new FileInputStream("data.bin")) {
    byte[] buffer = new byte[1024];
    int count;
    while ((count = in.read(buffer)) != -1) {
        // traiter buffer[0..count-1]
    }
}
```

34.4.2 OutputStream

Classe de base abstraite pour la sortie orientée octets.

Elle représente une destination où des octets bruts peuvent être écrits.

34.4.2.1 Méthodes clés

Méthode	Description
void `write(int b)`	Écrit les 8 bits de poids faible de l'entier
void `write(byte[])`	Écrit un tableau d'octets entier
void `write(byte[], int, int)`	Écrit une tranche d'un tableau d'octets
void `flush()`	Force l'écriture des données tamponnées
void `close()`	Effectue flush et libère la ressource

> **Note**
>
> Appeler `close()` appelle implicitement `flush()`.
>
> Ne pas faire flush ou close sur un OutputStream peut entraîner une perte de données.

34.4.2.2 Exemple d'utilisation typique

```
try (OutputStream out = new FileOutputStream("out.bin")) {
    out.write(new byte[] {1, 2, 3, 4});
    out.flush();
}
```

34.4.3 Reader et Writer

`Reader` et `Writer` sont les équivalents `orientés caractères` de InputStream et OutputStream.

Ils fonctionnent sur des caractères Unicode 16 bits au lieu d'octets bruts.

Classe	Direction	Basée sur caractères	Consciente de l'encodage
`Reader`	Entrée	Oui	Oui
`Writer`	Sortie	Oui	Oui

Readers et Writers impliquent toujours un `charset`, explicitement ou implicitement.

Cela en fait l'abstraction correcte pour le traitement de texte.

34.4.3.1 Gestion du charset

```
Reader reader = new InputStreamReader(
    new FileInputStream("file.txt"),
    StandardCharsets.UTF_8
);
```

> **Note**
>
> `InputStreamReader` et `OutputStreamWriter` sont des classes passerelles.
>
> Ils convertissent entre `flux d'octets` et `flux de caractères` en utilisant un `charset`.

34.5 Flux tamponnés et performance

Les `flux tamponnés` enveloppent un autre flux et ajoutent un buffer en mémoire.

Au lieu d'interagir avec le système d'exploitation à chaque read ou write, les données sont accumulées en mémoire et transférées en plus gros blocs.

- `BufferedInputStream` / `BufferedOutputStream` pour les flux d'octets
- `BufferedReader` / `BufferedWriter` pour les flux de caractères

> **Note**
>
> Les `flux tamponnés` sont des `decorators` : ils ne remplacent pas le flux sous-jacent, ils l'améliorent en ajoutant un comportement de mise en tampon.

34.5.1 Pourquoi la mise en tampon compte

Aspect	Non tamponné	Tamponné
`Appels système`	Fréquents	Réduits
`Performance`	Médiocre	Élevée
`Utilisation mémoire`	Minimale	Légèrement plus élevée

Les appels système sont des opérations coûteuses.

La mise en tampon les minimise en regroupant plusieurs lectures ou écritures logiques en moins d'opérations I/O physiques.

34.5.2 Comment fonctionne la lecture non tamponnée

Dans un flux non tamponné, chaque appel à read() peut entraîner un appel système natif.

C'est particulièrement inefficace lorsqu'on lit de grandes quantités de données.

```
try (InputStream in = new FileInputStream("data.bin")) {
    int b;
    while ((b = in.read()) != -1) {
        // chaque read() peut déclencher un appel système
    }
}
```

> **Note**
>
> Lire octet par octet sans mise en tampon est presque toujours un anti-pattern de performance.

34.5.3 Comment fonctionne BufferedInputStream

`BufferedInputStream` lit en interne un grand bloc d'octets dans un buffer.

Les appels `read()` suivants sont servis directement depuis la mémoire jusqu'à ce que le buffer soit vide.

```java
try (InputStream in =
    new BufferedInputStream(new FileInputStream("data.bin"))) {
        int b;
        while ((b = in.read()) != -1) {
            // la plupart des lectures sont servies depuis la mémoire, pas depuis l'OS
        }
}
```

> **Note**
>
> Le programme appelle toujours `read()` de manière répétée, mais le système d'exploitation n'est accédé que lorsque le buffer interne doit être rempli à nouveau.

34.5.4 Exemple de sortie tamponnée

La sortie tamponnée accumule les données en mémoire et les écrit en plus gros blocs.

L'opération `flush()` force l'écriture immédiate du buffer.

```java
try (OutputStream out =
    new BufferedOutputStream(new FileOutputStream("out.bin"))) {
        for (int i = 0; i < 1_000; i++) {
            out.write(i);
        }
        out.flush(); // force les données tamponnées sur disque
}
```

> **Note**
>
> `close()` appelle automatiquement flush().
>
> Appeler `flush()` explicitement est utile lorsque les données doivent être visibles immédiatement.

34.5.5 BufferedReader vs Reader

`BufferedReader` ajoute une `**lecture par lignes**` efficace au-dessus d'un Reader.

Sans mise en tampon, chaque caractère lu peut impliquer un appel système.

```java
try (BufferedReader reader =
    new BufferedReader(new FileReader("file.txt"))) {

        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
}
```

> **Note**
>
> La méthode `readLine()` n'est disponible que sur `BufferedReader` (pas sur `Reader`), car elle s'appuie sur la mise en tampon pour détecter efficacement les limites de ligne.

34.5.6 Exemple de BufferedWriter

```java
try (BufferedWriter writer =
    new BufferedWriter(new FileWriter("file.txt"))) {

        writer.write("Hello");
        writer.newLine();
        writer.write("World");
}
```

`BufferedWriter` minimise l'accès disque et fournit des méthodes pratiques telles que newLine().

> **Note**
>
> Enveloppez toujours les flux de fichiers avec une mise en tampon sauf s'il y a une forte raison de ne pas le faire
>
> Préférez BufferedReader / BufferedWriter pour le texte
>
> Préférez BufferedInputStream / BufferedOutputStream pour les données binaires

---

34.6 java.io vs java.nio (et java.nio.file)

Les applications Java modernes favorisent de plus en plus les API NIO et NIO.2, mais java.io reste fondamental et largement utilisé.

34.6.1 Différences conceptuelles

Aspect	java.io	java.nio / nio.2
`Modèle de programmation`	Basé sur les flux	Basé sur buffers / channels
`I/O bloquantes`	bloquantes par défaut	Capable de non-bloquant
`API fichiers`	File	Path + Files
`Scalabilité`	Limitée	Élevée
`Introduit`	Java 1.0	Java 4 / Java 7

> **Note**
>
> `java.nio` ne remplace pas `java.io`.
>
> De nombreuses classes NIO s'appuient en interne sur des flux ou coexistent avec eux.

34.6.2 java.nio (I/O de fichier moderne)

Le package `java.nio.file` (NIO.2) fournit une API fichiers de haut niveau, expressive et plus sûre. C'est l'approche préférée pour les opérations sur fichiers en Java 11+.

Exemple : lire un fichier (NIO)

```
Path path = Path.of("file.txt");
List<String> lines = Files.readAllLines(path);
```

Code java.io équivalent

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```

34.7 Quand utiliser quelle API

Scénario	API recommandée
`Lecture/écriture simple de fichier`	java.nio.file.Files
`Streaming binaire`	InputStream / OutputStream
`Traitement de texte en caractères`	Reader / Writer
`Serveurs haute performance`	java.nio.channels
`API legacy`	java.io

34.8 Pièges courants et conseils

- La fin de fichier est indiquée par -1, pas par une exception
- Fermer un flux wrapper ferme automatiquement le flux enveloppé
- `BufferedReader.readLine()` supprime les séparateurs de ligne
- `InputStreamReader` implique toujours un charset
- Les méthodes utilitaires Files lancent des IOException checked
- `available()` ne doit pas être utilisé pour détecter EOF

> **Note**
>
> La plupart des bugs I/O proviennent d'hypothèses incorrectes sur le blocking, la mise en tampon ou l'encodage des caractères.

◀ 33. APIs des fichiers et des chemins | ▲ Index | 35. API Java d'E/S (Legacy et NIO) ▶

35. API Java d'E/S (Legacy et NIO)

Table des matières

35.1 Legacy java.io — Conception, comportement et subtilités

L'API legacy `java.io` est l'abstraction E/S originale introduite dans Java 1.0.

Elle est orientée flux, bloquante, et mappée étroitement sur les concepts E/S du système d'exploitation.

Même si des API plus récentes existent, `java.io` reste fondamentale: beaucoup d'API de niveau supérieur s'appuient dessus, et elle est encore très utilisée.

35.1.1 L'abstraction de flux

Un `stream` représente un flux continu de données entre une source et une destination.

Dans `java.io`, les flux sont **unidirectionnels**: ils sont soit **d'entrée** soit **de sortie**.

Flux	Direction	Unité de données	Catégorie
`InputStream`	Entrée	Octets (8-bit)	Flux d'octets
`OutputStream`	Sortie	Octets (8-bit)	Flux d'octets
`Reader`	Entrée	Caractères	Flux de caractères
`Writer`	Sortie	Caractères	Flux de caractères

Les `stream` masquent l'origine concrète des données (fichier, réseau, mémoire) et exposent une interface uniforme de lecture/écriture.

35.1.2 Chaînage des flux et pattern Décorateur

La plupart des flux java.io sont conçus pour être combinés.

Chaque wrapper ajoute un comportement sans changer la source de données sous-jacente.

```
InputStream in =
    new BufferedInputStream(
        new FileInputStream("data.bin"));
```

Dans cet exemple:

- `FileInputStream` effectue l'accès réel au fichier
- `BufferedInputStream` ajoute un buffer en mémoire

> **Note**
>
> Cette conception est connue comme **Decorator Pattern**.
>
> Elle permet de stratifier des fonctionnalités de manière dynamique.

35.1.3 E/S bloquantes: ce que cela signifie

Tous les flux legacy `java.io` sont **bloquants**.

Cela signifie qu'un thread qui effectue des E/S peut être suspendu par le système d'exploitation.

Par exemple, quand tu appelles `read()` :

- si des données sont disponibles, elles sont retournées tout de suite
- s'il n'y a pas de données, le thread attend
- si on atteint la fin du flux, -1 est retourné

> **Note**
>
> Le comportement bloquant simplifie la programmation, mais limite la scalabilité.

35.1.4 Gestion des ressources: `close()`, `flush()` et pourquoi ils existent

Les flux encapsulent souvent des ressources natives du système d'exploitation comme `file descriptor` ou des handles de socket.

Ces ressources sont limitées et doivent être libérées explicitement.

Méthode	Objectif
`flush()`	Écrit les données bufferisées vers la destination
`close()`	Effectue flush et libère la ressource

```
try (OutputStream out = new FileOutputStream("file.bin")) {
    out.write(42);
} // close() appelé automatiquement
```

> **Note**
>
> Ne pas fermer les flux peut causer une perte de données ou un épuisement des ressources.

35.1.5 `finalize()` : pourquoi il existe et pourquoi il échoue

Les premières versions de Java ont tenté d'automatiser le nettoyage des ressources en utilisant la finalisation.

La méthode `finalize()` était appelée par le garbage collector avant de récupérer la mémoire.

Cependant, les temps du GC sont imprévisibles.

Aspect	finalize()
Temps d'exécution	Non spécifié
Fiabilité	Faible
État actuel	Déprécié

> **Note**
>
> `finalize()` ne doit jamais être utilisé pour le nettoyage E/S; il est déprécié et non sûr.

35.1.6 `available()` : objectif et abus

`available()` estime combien d'octets peuvent être lus sans bloquer.

Il n'indique pas la quantité totale de données restantes.

Cas d'usage typiques:

- éviter des blocages en UI ou parsing de protocoles
- dimensionner des buffers temporaires

```
if (in.available() > 0) {
    in.read(buffer);
}
```

> **Note**
>
> `available()` ne doit pas être utilisé pour détecter EOF. Seul `read()`, qui retourne -1, signale la fin du flux.

35.1.7 `mark()` et `reset()` : backtracking contrôlé

Certains flux d'entrée permettent de marquer une position et d'y revenir ensuite.

```
BufferedInputStream in = new BufferedInputStream(...);
in.mark(1024);
// read ahead
in.reset();
```

Flux	markSupported()
`FileInputStream`	Non
`BufferedInputStream`	Oui
`ByteArrayInputStream`	Oui

35.1.8 Reader, Writer et encodage des caractères

`Reader` et `Writer` opèrent sur des `caractères`, pas sur des octets.

Cela requiert un `encodage des caractères` (charset).

Si tu ne spécifies pas un charset, celui par défaut de la plateforme est utilisé.

```
new FileReader("file.txt"); // encodage par défaut de la plateforme
```

> **Note**
>
> S'appuyer sur le charset par défaut mène à des bugs de non-portabilité.
>
> Spécifie toujours un charset explicitement.

35.1.9 File vs FileDescriptor

`File` représente un `chemin` dans le filesystem.

Il ne représente pas une ressource ouverte.

`FileDescriptor` représente un handle natif du SE vers un fichier ou un flux ouvert.

Classe	Représente	Possède handle OS?
`File`	Chemin filesystem	Non
`FileDescriptor`	Handle fichier natif OS	Oui

> **Note**
>
> Plusieurs flux peuvent partager le même FileDescriptor.
>
> En en fermant un, on ferme la ressource sous-jacente pour tous.

35.2 `java.nio` — Buffer, Channel et E/S non bloquantes

L'API `java.nio` (New I/O) a été introduite pour résoudre les limites de `java.io`.

Elle offre un modèle E/S de plus bas niveau et plus explicite, qui mappe bien sur les systèmes d'exploitation modernes.

À la base, `java.nio` tourne autour de trois concepts:

- `Buffer` — conteneurs de mémoire explicites
- `Channel` — connexions de données bidirectionnelles
- `Selector` — multiplexage de lE/S non bloquante

35.2.1 Des flux aux buffers: un changement conceptuel

Les flux legacy masquent la gestion de la mémoire au programmeur.

Au contraire, `NIO` rend la mémoire explicite via les buffers.

Aspect	java.io	java.nio
Modèle de données	Basé sur flux (push)	Basé sur buffer (pull depuis les buffers)
Mémoire	Cachée dans les flux	Explicite via buffer
Contrôle	Simple, peu granulaire	Plus granulaire et configurable

Avec NIO, l'application contrôle quand les données sont lues en mémoire et comment elles sont consommées.

35.2.2 Buffer: objectif et structure

Un `buffer` est un conteneur typé de taille fixe.

Toutes les opérations E/S NIO lisent depuis ou écrivent sur des buffers.

Le buffer le plus commun est `ByteBuffer`.

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

Propriété	Signification
`capacity`	Taille totale du buffer
`position`	Index courant de lecture/écriture
`limit`	Limite des données lisibles ou inscriptibles

35.2.3 Cycle de vie du buffer: Write → Flip → Read

Les `buffer` ont un cycle d'usage rigoureux.

Le comprendre mal est une source commune de bugs.

Séquence typique:

- écris les données dans le buffer
- `flip()` pour passer en mode lecture
- lis les données du buffer
- `clear()` ou `compact()` pour le réutiliser

```java
ByteBuffer buffer = ByteBuffer.allocate(16);

buffer.put((byte) 1);
buffer.put((byte) 2);

buffer.flip(); // passe en mode lecture

while (buffer.hasRemaining()) {
    byte b = buffer.get();
}

buffer.clear(); // prêt à écrire de nouveau
```

> **Note**
>
> `flip()` n'efface pas les données: il règle position et limit.

35.2.4 `clear()` vs `compact()`

Après la lecture, un buffer peut être réutilisé de deux manières.

Méthode	Comportement
`clear()`	Jette les données non lues
`compact()`	Préserve les données non lues

`compact()` est utile dans les protocoles streaming où dans le buffer peuvent rester des messages partiels.

35.2.5 Heap buffers vs Direct buffers

Les buffers peuvent être alloués dans deux régions de mémoire différentes.

```java
ByteBuffer heap = ByteBuffer.allocate(1024);
ByteBuffer direct = ByteBuffer.allocateDirect(1024);
```

Type	Position mémoire	Caractéristiques
`Heap`	Heap JVM	GC, économique à allouer
`Direct`	Mémoire native	Meilleur throughput E/S, plus coûteux à allouer

> **Note**
>
> Les direct buffer réduisent les copies entre JVM et OS, mais doivent être utilisés avec attention pour éviter une pression mémoire.

35.2.6 Channel: ce que c'est

Un `channel` représente une connexion vers une entité E/S comme fichier, socket ou device.

À la différence des flux, **les channel sont bidirectionnels**.

Channel	Type	Objectif
`FileChannel`	Fichier	E/S sur fichiers
`SocketChannel`	TCP	Networking stream (TCP)
`DatagramChannel`	UDP	Networking datagram (UDP)

```
try (FileChannel channel =
    FileChannel.open(Path.of("file.txt"))) {

    ByteBuffer buffer = ByteBuffer.allocate(128);
    channel.read(buffer);
}
```

35.2.7 Channel bloquants vs non bloquants

Les channel peuvent opérer en mode bloquant ou non bloquant.

```
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
```

En mode **non bloquant**:

- `read()` peut retourner tout de suite avec 0 octets
- `write()` peut écrire seulement une partie des données

> **Note**
>
> L'E/S non bloquante déplace la complexité du SE vers l'application.

35.2.8 Scatter/Gather E/S

NIO supporte lecture/écriture depuis/vers plusieurs buffers avec une seule opération.

```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);

ByteBuffer[] buffers = { header, body };
channel.read(buffers);
```

Utile pour des protocoles structurés (header + payload).

35.2.9 Selector: multiplexage de IE/S non bloquante

Les `Selector` permettent à un seul thread de monitorer plusieurs channel.

Ils sont la base des serveurs scalables.

Composant	Rôle
`Selector`	Monitore plusieurs channel
`SelectionKey`	Représente enregistrement et état du channel
`Interest set`	Opérations observées (read, write, etc.)

35.2.10 Quand utiliser `java.nio`

`NIO` est adapté quand:

- il faut une haute concurrence
- il te faut un contrôle fin sur la mémoire
- tu implémentes des protocoles ou des serveurs

Pour des opérations simples sur fichiers, souvent `java.nio.file.Files` suffit.

35.3 `java.nio.file` (NIO.2) — Opérations sur fichiers et répertoires (Legacy vs Moderne)

Cette section se concentre sur les opérations pratiques sur fichiers et répertoires.

Nous comparons les approches legacy (java.io.File + flux java.io) avec celles modernes NIO.2 (Path + Files).

L'objectif n'est pas seulement de connaître les noms des méthodes, mais de comprendre:

- ce que fait vraiment chaque méthode
- ce qu'elle retourne et comment elle signale les erreurs
- quels pièges existent (race condition, liens, permissions, portabilité)
- quand une méthode de Files est une amélioration sûre par rapport à l'ancienne approche

35.3.1 Vérifications d'existence et d'accessibilité

Une opération très commune est de vérifier si un fichier existe et s'il est accessible (lecture, écriture, exécution).

À la fois l'API legacy (java.io.File) et NIO.2 (java.nio.file.Files) fournissent des méthodes pour ces vérifications.

Il est toutefois important de comprendre que ces vérifications sont volontairement imprécises dans les deux API.

Ce sont des indices best-effort, pas des garanties fiables.

35.3.1.1 API legacy (File)

```
File f = new File("data.txt");

boolean exists = f.exists();
boolean canRead = f.canRead();
boolean canWrite = f.canWrite();
boolean canExec = f.canExecute();
```

Ces méthodes retournent boolean et n'expliquent pas pourquoi une opération a échoué.

Par exemple, exists() peut retourner false quand:

- le fichier n'existe vraiment pas
- le fichier existe mais l'accès est refusé
- un lien symbolique est cassé
- une erreur E/S se produit

L'API ne permet pas de distinguer les cas.

35.3.1.2 API moderne (Files)

```
Path p = Path.of("data.txt");

boolean exists = Files.exists(p);
boolean readable = Files.isReadable(p);
boolean writable = Files.isWritable(p);
boolean executable = Files.isExecutable(p);
```

Ces méthodes aussi retournent boolean et masquent la raison de l'éventuel insuccès.

NIO.2 ajoute une méthode explicite pour exprimer l'incertitude:

```
boolean notExists = Files.notExists(p);
```

> **Note**
>
> `exists()` et `notExists()` peuvent être tous deux `false` quand l'état n'est pas déterminable (par exemple à cause de permissions).

Cela ne rend pas la vérification plus précise: cela rend seulement l'incertitude explicite.

35.3.1.2.1 Conscience des liens symboliques (amélioration réelle)

Une vraie amélioration de NIO.2 est le contrôle sur comment gérer les liens symboliques:

```
Files.exists(p, LinkOption.NOFOLLOW_LINKS);
```

La classe File legacy ne distingue pas de manière fiable:

- fichier manquant
- lien symbolique cassé
- lien vers target inaccessible

NIO.2 permet des check link-aware et une inspection explicite des liens.

35.3.1.2.2 Pattern d'usage correct (critique)

Aucune des deux API ne donne de diagnostics fiables via boolean "de check".

Le code NIO.2 correct ne "contrôle pas avant".

À la place il tente l'opération et gère l'exception:

```
try {
    Files.delete(p);
} catch (NoSuchFileException e) {
    // le fichier n'existe vraiment pas
} catch (AccessDeniedException e) {
    // problème de permissions
} catch (IOException e) {
    // autre erreur E/S
}
```

> **Note**
>
> Le vrai avantage de NIO.2 est le diagnostic via exceptions pendant les actions, pas des check d'existence plus "précis".

35.3.1.2.3 Tableau récapitulatif

Objectif	Legacy (File)	Moderne (Files)	Détail clé
Vérifier existence	`exists()`	`exists()` / `notExists()`	notExists() peut être false si l'état n'est pas déterminable
Vérifier read/write	`canRead()` / `canWrite()`	`isReadable()` / `isWritable()`	Files peut utiliser LinkOption.NOFOLLOW_LINKS quand supporté
Détails erreur	Non disponibles	Disponibles via exceptions sur les actions	Les check boolean n'expliquent pas le motif de l'échec

35.3.2 Création de fichiers et de répertoires

La création est une grande faiblesse du File legacy.

Dans le legacy on utilise souvent createNewFile() et mkdir/mkdirs(), qui retournent boolean et donnent peu d'infos diagnostiques.

35.3.2.1 API legacy (File)

```
File f = new File("a.txt");
boolean created = f.createNewFile(); // peut lancer IOException

File dir = new File("dir");
boolean ok1 = dir.mkdir();
boolean ok2 = new File("a/b/c").mkdirs();
```

`mkdir()` crée un seul niveau; `mkdirs()` crée aussi les parents.

Les deux retournent false en cas d'échec mais sans dire pourquoi.

35.3.2.2 API moderne (Files)

```
Path file = Path.of("a.txt");
Files.createFile(file);

Path dir1 = Path.of("dir");
Files.createDirectory(dir1);

Path dirDeep = Path.of("a/b/c");
Files.createDirectories(dirDeep);
```

> **Note**
>
> `Files.createFile` lance `FileAlreadyExistsException` si le fichier existe.
>
> Souvent il est préférable aux check boolean parce qu'il est race-safe.

Objectif	Legacy (File)	Moderne (Files)	Détail clé
Créer fichier	`createNewFile()`	`createFile()`	NIO lance FileAlreadyExistsException s'il existe
Créer répertoire	`mkdir()`	`createDirectory()`	NIO lance des exceptions détaillées
Créer parents	`mkdirs()`	`createDirectories()`	Atomicité non garantie pour répertoires profonds

35.3.3 Suppression de fichiers et de répertoires

La sémantique de delete diffère beaucoup entre legacy et NIO.2.

Le legacy `delete()` retourne boolean; NIO.2 offre des méthodes qui lancent des exceptions significatives.

35.3.3.1 API legacy (File)

```
File f = new File("a.txt");
boolean deleted = f.delete();
```

S'il échoue (permissions, fichier manquant, répertoire non vide), `delete()` retourne souvent false sans détails.

35.3.3.2 API moderne (Files)

```
Files.delete(Path.of("a.txt"));
```

Pour "supprime si présent", utilise `deleteIfExists()`.

```
Files.deleteIfExists(Path.of("a.txt"));
```

Objectif	Legacy (File)	Moderne (Files)	Détail clé
Supprimer	`delete()`	`delete()`	`Files.delete()` lance exception avec la cause de l'échec
Supprimer si existe	`exists() + delete()`	`deleteIfExists()`	Évite race TOCTOU (check-then-act)

35.3.4 Copie de fichiers et de répertoires

Dans le legacy, copier requiert typiquement lecture/écriture manuelle via flux.

NIO.2 fournit des opérations de copie de haut niveau avec options.

35.3.4.1 Technique legacy (flux manuels)

```
try (InputStream in = new FileInputStream("src.bin"); OutputStream out = new FileOutputStream(

    byte[] buf = new byte[8192];
    int n;
    while ((n = in.read(buf)) != -1) {
        out.write(buf, 0, n);
    }
}
```

C'est verbeux et c'est facile de se tromper (absence de buffering, fermeture, etc.).

35.3.4.2 API moderne (Files.copy)

```
Files.copy(Path.of("src.bin"), Path.of("dst.bin"));
```

Le comportement est contrôlable avec options.

```
Files.copy(
    Path.of("src.bin"),
    Path.of("dst.bin"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES
);
```

> **Note**
>
> `Files.copy` lance FileAlreadyExistsException par défaut.
>
> Utilise `REPLACE_EXISTING` quand l'overwrite est intentionnel.

Objectif	Approche legacy	Moderne (Files)	Détail clé
Copier fichier	Boucle flux manuelle	`Files.copy(Path, Path, …)`	Options: `REPLACE_EXISTING`, `COPY_ATTRIBUTES`
Copier flux	InputStream/OutputStream	`Files.copy(InputStream, Path, …)`	Utile pour upload/download et piping
Copier répertoire	Récursion manuelle	`walkFileTree + Files.copy`	Aucun one-liner pour copy complète d'arbre

35.3.5 Déplacement et renommage

Le renommage legacy utilise souvent `File.renameTo()`, notoirement peu fiable et dépendant de la plateforme.

NIO.2 fournit `Files.move()` avec sémantique précise et options.

35.3.5.1 API legacy

```
boolean ok = new File("a.txt").renameTo(new File("b.txt"));
```

`renameTo()` retourne false sans explication, et peut échouer entre filesystem.

35.3.5.2 API moderne

```
Files.move(Path.of("a.txt"), Path.of("b.txt"));
```

Les options rendent le comportement explicite.

```
Files.move(
    Path.of("a.txt"),
    Path.of("b.txt"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.ATOMIC_MOVE
);
```

> **Note**
>
> ATOMIC_MOVE est garanti seulement si le déplacement arrive dans le même filesystem. Sinon une exception est lancée.

Objectif	Legacy (File)	Moderne (Files)	Détail clé
Renommage / move	`renameTo()`	`move()`	Exceptions + options explicites
Move atomique	Non supporté	`move(…, ATOMIC_MOVE)`	Garanti seulement même filesystem
Replace existing	Non explicite	`REPLACE_EXISTING`	Intention d'overwrite explicite

35.3.6 Lecture et écriture de texte et d'octets (améliorations de Files)

Une grande amélioration de NIO.2 est la classe utilitaire `Files`, avec des méthodes de haut niveau pour lecture/écriture communes.

Elle réduit le boilerplate et améliore la justesse.

35.3.6.1 Lecture/écriture texte legacy

```
try (BufferedReader r = new BufferedReader(new FileReader("file.txt"))) {
    String line = r.readLine();
}
```

```
try (BufferedWriter w = new BufferedWriter(new FileWriter("file.txt"))) {
    w.write("hello");
}
```

Ces classes legacy utilisent souvent le charset par défaut si on n'utilise pas un bridge explicite.

35.3.6.2 Lecture/écriture texte moderne

```
List<String> lines = Files.readAllLines(Path.of("file.txt"), StandardCharsets.UTF_8);
Files.write(Path.of("file.txt"), lines, StandardCharsets.UTF_8);

Files.lines(Path.of("file.txt")).forEach(System.out::println);

String string = Files.readString(Path.of("file.txt"));
Files.writeString(Path.of("file.txt"), string);
```

35.3.6.3 Lecture/écriture binaire moderne

```
byte[] data = Files.readAllBytes(Path.of("data.bin"));
Files.write(Path.of("out.bin"), data);
```

> **Important**
>
> `readAllBytes` et `readAllLines` chargent tout en mémoire.
>
> Utilise `Files.lines()` (lazy) ou, pour de gros fichiers, préfère des API streaming comme newBufferedReader/newInputStream.

Tâche	Méthode legacy	Méthode NIO.2 Files	Détail clé
Lire tous les octets	Boucle InputStream manuelle	`readAllBytes()`	Charge tout en mémoire
Lire toutes les lignes	Boucle BufferedReader	`readAllLines()`	Charge tout en mémoire
Lire lignes lazy	Boucle BufferedReader	`lines()`	Lazy, stream à fermer
Écrire octets	OutputStream	`write(Path, byte[])`	Concis
Écrire lignes	Boucle BufferedWriter	`write(Path, Iterable, …)`	Charset spécifiable
Append texte	FileWriter(true)	`write(…, APPEND)`	Options explicites

35.3.7 newInputStream/newOutputStream et newBufferedReader/newBufferedWriter

Ces `factory method` créent des flux/reader à partir d'un Path.

Ils sont le bridge recommandé entre streaming classique et gestion Path NIO.2.

```
try (InputStream in = Files.newInputStream(Path.of("a.bin"))) { }
try (OutputStream out = Files.newOutputStream(Path.of("b.bin"))) { }
```

```
try (BufferedReader r = Files.newBufferedReader(Path.of("t.txt"), StandardCharsets.UTF_8)) { }
try (BufferedWriter w = Files.newBufferedWriter(Path.of("t.txt"), StandardCharsets.UTF_8)) { }
```

35.3.8 Listing de répertoires et traversée d'arbres

Dans le legacy, le listing de répertoires se base sur `File.list()` et `File.listFiles()`.

Ces méthodes retournent des array et offrent peu de diagnostics.

35.3.8.1 Listing legacy

```
File dir = new File(".");
File[] children = dir.listFiles();
```

NIO.2 offre plus d'approches selon le besoin.

35.3.8.2 Listing moderne (DirectoryStream)

```
try (DirectoryStream<Path> ds = Files.newDirectoryStream(Path.of("."))) {
    for (Path p : ds) {
        System.out.println(p);
    }
}
```

35.3.8.3 Walking moderne (Files.walk)

```
Files.walk(Path.of("."))
    .filter(Files::isRegularFile)
    .forEach(System.out::println);
```

> **Note**
>
> `Files.walk` retourne un Stream qui doit être fermé. Utilise `try-with-resources`.

```
try (Stream<Path> s = Files.walk(Path.of("."))) {
    s.forEach(System.out::println);
}
```

35.3.8.4 Traversal avec FileVisitor

Pour un contrôle complet (skip subtree, gestion erreurs, follow link), utilise `walkFileTree + FileVisitor`.

```
Files.walkFileTree(Path.of("."), new SimpleFileVisitor<>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

Objectif	Legacy	Moderne	Détail clé
Listing dir	`list()` / `listFiles()`	`newDirectoryStream()`	Lazy, doit être fermé
Walk tree (simple)	Récursion manuelle	`walk()` (Stream)	Stream doit être fermé
Walk tree (contrôle)	Récursion manuelle	`walkFileTree()`	Contrôle fin et gestion erreurs

35.3.9 Recherche et filtre

La recherche est typiquement `traversal + filtre`.

NIO.2 offre building block: glob pattern, stream, visitor.

```
try (DirectoryStream<Path> ds =
    Files.newDirectoryStream(Path.of("."), "*.txt")) {
    for (Path p : ds) {
        System.out.println(p);
    }
}
```

```
try (Stream<Path> s = Files.find(Path.of("."), 10,
    (p, a) -> a.isRegularFile() && p.toString().endsWith(".log"))) {
    s.forEach(System.out::println);
}
```

35.3.10 Attributs: lecture, écriture et view

Le File legacy expose peu d'attributs (size, lastModified).

NIO.2 supporte des metadata riches via attribute view.

35.3.10.1 Attributs legacy

```
long size = new File("a.txt").length();
long lm = new File("a.txt").lastModified();
```

35.3.10.2 Attributs modernes

```
BasicFileAttributes a =
    Files.readAttributes(Path.of("a.txt"), BasicFileAttributes.class);

long size = a.size();
FileTime modified = a.lastModifiedTime();
```

Accès via noms string-based:

```
Object v = Files.getAttribute(Path.of("a.txt"), "basic:size");
Files.setAttribute(Path.of("a.txt"), "basic:lastModifiedTime", FileTime.fromMillis(0));
```

> **Note**
>
> Les attribute view dépendent du filesystem.
>
> Les attributs non supportés génèrent des exceptions.

35.3.11 Liens symboliques et follow des liens

NIO.2 peut détecter et lire des liens symboliques de manière explicite.

```
Path link = Path.of("mylink");
boolean isLink = Files.isSymbolicLink(link);

if (isLink) {
    Path target = Files.readSymbolicLink(link);
}
```

Beaucoup de méthodes suivent les liens par défaut.

Pour l'éviter, passe `LinkOption.NOFOLLOW_LINKS` quand supporté.

35.3.12 Synthèse: pourquoi Files est une amélioration

La classe utilitaire `Files` améliore la programmation filesystem parce que:

- réduit le boilerplate (copy/move/read/write)
- fournit des options explicites (overwrite, atomic move, follow links)
- offre des metadata plus riches (attributes/views)
- supporte traversal et recherche scalables

Les API legacy restent surtout pour compatibilité ou quand requises par des librairies legacy.

35.4 Sérialisation — Object stream, compatibilité et pièges

La sérialisation est le processus de convertir un graphe d'objets en un flux d'octets pour le mémoriser ou le transmettre, et le reconstruire ensuite.

En Java, la sérialisation classique est implémentée par `java.io.ObjectOutputStream` et `java.io.ObjectInputStream`.

Ce sujet est important parce qu'il combine:

- flux E/S et graphes d'objets
- versioning et backward compatibility
- considérations de sécurité et pattern d'usage sûrs
- règles du langage (`transient` , static, `serialVersionUID`)

35.4.1 Ce que fait la sérialisation (et ce qu'elle ne fait pas)

Quand un objet est sérialisé, Java écrit des informations suffisantes pour le reconstruire:

- nom de la classe
- serialVersionUID
- valeurs des champs d'instance sérialisables
- références entre objets (identité)

La sérialisation n'inclut pas automatiquement:

- champs static (état de classe)
- champs transient (exclus explicitement)
- objets référencés non sérialisables (à moins de gestion spéciale)

35.4.2 Les deux principales marker interface

La sérialisation Java est activée en implémentant une de ces interfaces.

Interface	Signification	Niveau de contrôle
`Serializable`	Marker opt-in, mécanisme par défaut	Moyen (hook possibles)
`Externalizable`	Requiert implémentation manuelle read/write	Haut (contrôle total sur le format)

> **Note**
>
> `Serializable` n'a pas de méthodes: c'est une marker interface.
>
> `Externalizable` étend Serializable et ajoute readExternal/writeExternal.

35.4.3 Exemple de base: écrire et lire un objet

Pattern minimal utilisé en pratique.

```
import java.io.*;

class Person implements Serializable {

    private String name;
    private int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

}

public class Demo {

    public static void main(String[] args) throws Exception {

        Person p = new Person("Alice", 30);

        try (ObjectOutputStream out =
                 new ObjectOutputStream(new FileOutputStream("p.bin"))) {
            out.writeObject(p);
        }

        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream("p.bin"))) {
            Person copy = (Person) in.readObject();
        }
    }

}
```

> **Note**
>
> `readObject()` retourne Object: un cast est nécessaire. `readObject()` peut lancer ClassNotFoundException.

35.4.4 Graphes d'objets, références et identité

La sérialisation préserve l'identité des objets à l'intérieur du même flux.

Si la même référence apparaît plusieurs fois, Java l'écrit une seule fois puis écrit une back-reference.

```
Person p = new Person("Bob", 40);
Object[] arr = { p, p }; // même référence deux fois

out.writeObject(arr);
Object[] restored = (Object[]) in.readObject();

// restored[0] et restored[1] pointent vers le même objet
```

> **Note**
>
> Cela prévient la récursion infinie dans des graphes cycliques.

35.4.5 `serialVersionUID` : la clé de versioning

`serialVersionUID` est un identifiant `long` utilisé pour vérifier la compatibilité entre flux sérialisé et définition de la classe.

Si l'UID diffère, la désérialisation échoue typiquement avec InvalidClassException.

Si tu ne déclares pas `serialVersionUID`, la JVM en calcule un depuis la structure de la classe: de petites modifications peuvent le compromettre.

```
class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    private String name;
    private int age;
}
```

Type de modification	Impact compatibilité (par défaut)
Ajouter un champ	Souvent compatible (champ nouveau avec défaut)
Supprimer un champ	Souvent compatible (champ manquant ignoré)
Changer type de champ	Souvent incompatible
Changer nom/paquet	Incompatible
Changer serialVersionUID	Incompatible

> **Note**
>
> Déclarer un serialVersionUID stable est la manière standard de contrôler la compatibilité.

35.4.6 Champs `transient` et `static`

Les champs `transient` sont exclus de la sérialisation.

À la désérialisation, les champs transient prennent des valeurs par défaut (0, false, null) sauf restauration manuelle.

Les champs `static` appartiennent à la classe, pas à l'instance, donc ils ne sont pas sérialisés.

```java
class Session implements Serializable {

    private static final long serialVersionUID = 1L;

    static int counter = 0;      // non sérialisé
    transient String token;      // non sérialisé
    String user;                 // sérialisé
}
```

> **Note**
>
> Si un transient est nécessaire après la désérialisation, il doit être recalculé ou restauré manuellement.

35.4.7 Champs non sérialisables et NotSerializableException

Si un objet contient un champ dont le type n'est pas sérialisable, la sérialisation échoue avec NotSerializableException.

```java
class Holder implements Serializable {

    private static final long serialVersionUID = 1L;

    private Thread t; // Thread n'est pas sérialisable
}
```

Solutions typiques:

- marquer le champ transient
- le remplacer par une représentation sérialisable
- utiliser des hook de sérialisation custom

35.4.8 Constructeurs et sérialisation

Le comportement des constructeurs en désérialisation est une source fréquente de confusion.

Java restaure l'état principalement depuis le flux d'octets, sans exécuter les constructeurs.

35.4.8.1 Règle: les constructeurs des classes Serializable NE sont pas appelés

Pendant la désérialisation d'une classe Serializable, ses constructeurs, ou tout bloc statique ou bloc d'initialisation d'instance, NE sont pas exécutés.

L'instance est créée sans appeler ces constructeurs (ou tout bloc statique ou bloc d'initialisation d'instance), et les champs sont injectés depuis le flux.

> **Note**
>
> Pour cela les constructeurs des classes Serializable ne doivent pas contenir une logique d'initialisation essentielle: elle ne serait pas exécutée en désérialisation.

35.4.8.2 Règle d'héritage: la première superclass non-Serializable est appelée

Si une classe Serializable a une superclasse non Serializable, la désérialisation doit initialiser cette partie.

Donc Java appelle **le constructeur no-arg de la première superclasse non-Serializable**.

Implications:

- la superclasse non Serializable doit avoir un no-arg accessible
- les sous-classes Serializable sautent les constructeurs, les superclasses non Serializable non

35.4.8.3 Tableau: quels constructeurs sont exécutés

Type de classe	Constructeur appelé en désérialisation
Classe Serializable	Non
Sous-classe Serializable	Non
Première superclasse non Serializable	Oui (no-arg)
Classe Externalizable	Oui (public no-arg requis)

35.4.8.4 Exemple: quels constructeurs sont appelés

```
import java.io.*;

class A {
    A() {
        System.out.println("A constructor");
    }
}

class B extends A implements Serializable {
    private static final long serialVersionUID = 1L;
    B() {
        System.out.println("B constructor");
    }
}

class C extends B {
    private static final long serialVersionUID = 1L;
    C() {
        System.out.println("C constructor");
    }
}

public class Demo {
    public static void main(String[] args) throws Exception {

        C obj = new C();

        try (ObjectOutputStream out =
                 new ObjectOutputStream(new FileOutputStream("c.bin"))) {
            out.writeObject(obj);
        }

        try (ObjectInputStream in =
                 new ObjectInputStream(new FileInputStream("c.bin"))) {
            Object restored = in.readObject();
        }
    }
}
```

Output attendu et explication
Pendant la construction normale (new C()):

```
A constructor
B constructor
C constructor
```

Pendant la désérialisation (readObject):

```
A constructor
```

Explication:

- C est Serializable → C() n'est pas appelé
- B est Serializable → B() n'est pas appelé
- A n'est pas Serializable → A() est appelé (no-arg)
- Les champs de B et C sont restaurés depuis le flux

> **Note**
>
> Si la première superclasse non-Serializable n'a pas un no-arg accessible, la désérialisation échoue.

35.4.9 Hook de sérialisation custom: `writeObject` et `readObject`

Les hook custom servent quand la sérialisation par défaut ne suffit pas (état transient, champs dérivés, chiffrement, validation, compatibilité).

Ils sont avancés mais importants pour une désérialisation correcte.

35.4.9.1 Pourquoi la sérialisation custom existe

Par défaut, Java sérialise automatiquement tous les champs d'instance non static et non transient.

C'est commode, mais cela ne couvre pas des besoins fréquents.

Motifs typiques:

- un champ ne doit pas être sauvegardé directement (données sensibles)
- un champ est dérivé/cache et doit être recalculé
- validation en lecture est nécessaire (refuser un état invalide)
- logique de backward/forward compatibility est nécessaire
- un objet référencé n'est pas Serializable et doit être géré

35.4.9.2 Ce que sont vraiment `writeObject` et `readObject`

Pour personnaliser sérialisation/désérialisation, une classe peut définir deux méthodes privées spéciales appelées `writeObject` et `readObject`.

Ce ne sont pas des override de méthodes d'interfaces ou de superclass: elles ne font pas partie du flux normal du programme.

Tu ne les appelles jamais toi.

Le framework de sérialisation (ObjectOutputStream/ObjectInputStream) les identifie via reflection, **seulement** si nom et signature sont exacts, et les invoque automatiquement.

S'ils n'existent pas (ou la signature est mauvaise), la sérialisation par défaut est utilisée.

> **Note**
>
> Si la signature est erronée (visibilité, paramètres, return type, exceptions), le framework ne la reconnaît pas et revient silencieusement au défaut.

35.4.9.3 Signatures requises (exactes)

```
private void writeObject(ObjectOutputStream out) throws IOException

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException
```

Contraintes:

- elles doivent être private
- elles doivent retourner void
- les types des paramètres doivent correspondre exactement
- les exceptions doivent être compatibles

35.4.9.4 Ce qui se passe en sérialisation: step-by-step

Quand tu sérialises:

```
out.writeObject(obj);
```

Mécanisme:

- vérifie Serializable
- cherche un private writeObject(ObjectOutputStream)
- si absent → sérialisation par défaut
- si présent → ton writeObject est appelé

Point clé: à l'intérieur de writeObject, Java n'écrit pas automatiquement les champs "normaux" si tu ne le demandes pas. Pour cela existe:

```
out.defaultWriteObject();
```

`defaultWriteObject()` signifie: "sérialise les champs sérialisables normaux avec le mécanisme standard".

Ensuite tu peux écrire des données extra comme tu veux.

35.4.9.5 Pattern typique et règle de l'ordre write/read

Pattern typique: utiliser default puis étendre.

L'ordre de lecture doit coïncider avec l'ordre d'écriture.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject(); // écrit les champs normaux
    out.writeInt(42);         // écrit des données extra
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();   // lit les champs normaux
    int x = in.readInt();     // lit les données extra dans le même ordre
}
```

> **Note**
>
> Si tu écris des valeurs extra (int/string/etc.), tu dois les lire dans la même séquence, sinon la désérialisation échoue ou corrompt l'état.

35.4.10 Exemple d'usage: restaurer un champ dérivé transient

Cas typique: recalculer une valeur cached transient après désérialisation.

```
class User implements Serializable {

    private static final long serialVersionUID = 1L;

    private String firstName;
    private String lastName;

    private transient String fullName;

    User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = firstName + " " + lastName;
    }

    private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException {

        in.defaultReadObject();                 // restaure firstName et lastName
        fullName = firstName + " " + lastName; // recalcule le transient
    }
}
```

35.4.11 Externalizable: contrôle total (et responsabilité totale)

Externalizable requiert de définir manuellement comment écrire et lire l'objet.

Il requiert aussi un constructeur public no-arg, parce que la désérialisation instancie d'abord l'objet.

```java
import java.io.*;

class Point implements Externalizable {
    int x;
    int y;

    public Point() { } // requis

    public Point(int x, int y) { this.x = x; this.y = y; }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(x);
        out.writeInt(y);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException {
        x = in.readInt();
        y = in.readInt();
    }
}
```

> **Note**
>
> Avec Externalizable tu contrôles le format. Si tu le changes, tu dois gérer toi la backward compatibility.

35.4.12 Considérations de sécurité sur `readObject()`

La désérialisation de données non fiables est dangereuse parce qu'elle peut exécuter du code indirectement via:

- hook readObject
- logique d'initialisation
- gadget chain dans des librairies

Lignes directrices:

- ne désérialise jamais des octets non fiables sans un motif fort
- préférer des formats sûrs (JSON, protobuf) pour des inputs externes
- si obligé, utiliser object filter et validation rigoureuse

35.4.13 Pièges communs et conseils pratiques

- Serializable est seulement marker: il ne requiert pas de méthodes
- `readObject` retourne Object et peut lancer ClassNotFoundException
- les champs `static` ne sont jamais sérialisés
- les champs `transient` reviennent à défaut sauf restauration
- sans `serialVersionUID` la compatibilité peut se casser "par surprise"
- Externalizable requiert public no-arg constructor
- NotSerializableException quand un champ référencé n'est pas sérialisable

35.4.14 Quand utiliser (ou éviter) la sérialisation Java

Utilise la sérialisation classique surtout pour:

- persistance locale de courte durée avec versions contrôlées
- caching en mémoire quand les deux extrémités sont fiables
- systèmes legacy qui l'utilisent déjà

Évite-la pour:

- protocoles de réseau publics
- stockage à long terme avec schéma évolutif
- inputs non fiables

36. Interagir avec l'Utilisateur (Flux E/S Standard)

Table des matières

Les programmes Java doivent souvent interagir avec l'utilisateur: afficher des informations, lire une entrée et formater la sortie.

Cette interaction est implémentée en utilisant les flux E/S standard, qui sont des flux Java normaux connectés au système d'exploitation.

Ce chapitre explique comment Java interagit avec la console et l'entrée/sortie standard, en partant des concepts les plus basiques et en passant aux API de niveau plus élevé.

36.1 Les Flux E/S Standard

Chaque programme Java commence avec trois flux prédéfinis fournis par la JVM.

Ils sont connectés à l'environnement du processus (en général un terminal ou une console).

Flux	Champ	Type	But
Sortie standard	`System.out`	PrintStream	Sortie normale
Erreur standard	`System.err`	PrintStream	Sortie d'erreur
Entrée standard	`System.in`	InputStream	Entrée de l'utilisateur

> **Note**
>
> Ces flux sont créés par la JVM, pas par le programme.
>
> Ils existent pendant toute la durée du processus.

36.2 `PrintStream` : Qu'est-ce et Pourquoi il Existe

`PrintStream` est un flux de sortie orienté octets conçu pour une sortie lisible par l'utilisateur.

Il enveloppe un autre OutputStream et ajoute des méthodes d'impression pratiques.

`System.out` et `System.err` sont tous deux des instances de `PrintStream`.

36.2.1 Caractéristiques Clés de PrintStream

- Flux orienté octets avec des helpers pour l'impression de texte
- Fournit des méthodes `print()` et `println()`
- Convertit automatiquement les valeurs en texte
- Ne lance pas `IOException` sur erreurs d'écriture
- Supporte optionnellement l'auto-flush sur newline / `println()`

> **Note**
>
> Contrairement à la plupart des flux, PrintStream supprime les IOExceptions.
>
> Les erreurs doivent être vérifiées en utilisant checkError().

36.2.2 Utilisation de Base de PrintStream

```
System.out.println("Hello");
System.out.print("Value: ");
System.out.println(42);
```

`println()` ajoute automatiquement le séparateur de ligne spécifique à la plateforme.

36.2.3 Formater l'Output avec PrintStream

PrintStream supporte une sortie formatée en utilisant `printf()` et `format()`, qui sont basés sur la même syntaxe que String.format().

```
System.out.printf("Name: %s, Age: %d%n", "Alice", 30);
```

Spécificateur	Signification
`%s`	Chaîne
`%d`	Entier
`%f`	Virgule flottante
`%n`	Nouvelle ligne indépendante de la plateforme

> **Note**
>
> `printf()` n'ajoute pas automatiquement une nouvelle ligne à moins qu'on spécifie `%n` .

36.3 Lire l'Input comme Flux E/S

L'entrée standard (System.in) est un InputStream connecté à l'entrée de l'utilisateur.

Elle fournit des octets bruts et doit être adaptée pour un usage pratique.

36.3.1 Lecture de Bas Niveau depuis System.in

Au niveau le plus bas, tu peux lire des octets bruts depuis System.in.

Ceci est rarement pratique pour des programmes interactifs.

```
int b = System.in.read();
```

> **Note**
>
> `System.in.read()` bloque jusqu'à ce que l'entrée soit disponible.

36.3.2 Utilisation de InputStreamReader et BufferedReader

Pour lire une entrée textuelle, `System.in` est typiquement enveloppé dans un Reader et bufferisé.

```
BufferedReader reader =
new BufferedReader(new InputStreamReader(System.in));

String line = reader.readLine();
```

Ceci convertit `octets → caractères` et permet une entrée basée sur des lignes.

36.4 La Classe Scanner (Pratique mais Subtile)

`Scanner` est un utilitaire de haut niveau pour le parsing d'entrée textuelle.

Il est souvent utilisé pour l'interaction avec la console, surtout dans de petits programmes.

```
Scanner sc = new Scanner(System.in);
int value = sc.nextInt();
String text = sc.nextLine();
```

> **Note**
>
> `Scanner` effectue tokenisation et parsing, pas une simple lecture.
>
> Cela la rend pratique mais plus lente et parfois surprenante.

36.4.1 Problèmes Communs de Scanner

- Mélanger `nextInt()` (et autres `nextXxx()`) avec `nextLine()` peut sembler "sauter" l'entrée car le newline final du token numérique est encore dans le buffer.
- Les erreurs de parsing lancent InputMismatchException
- Scanner est relativement lente pour des entrées de grande taille

---

36.5 Fermeture des Flux Système

Les `flux système` sont spéciaux et doivent être gérés avec attention.

Flux	Fermer explicitement?
`System.out`	Non
`System.err`	Non
`System.in`	En général non

Fermer `System.out` ou `System.err` ferme le flux sous-jacent du système d'exploitation et affecte l'ensemble de la JVM: fermer ces flux affecte l'ensemble du processus JVM, pas seulement la classe ou la méthode courante.

> **Note**
>
> Dans presque toutes les applications, tu ne devrais PAS fermer `System.out` ou `System.err`.

36.6 Acquérir l'Input avec `Console`

La classe `Console` fournit une manière de niveau plus élevé et plus sûre d'interagir avec l'utilisateur.

Elle est conçue spécifiquement pour des programmes de console interactifs.

```
Console console = System.console();
if (console == null) {
    throw new IllegalStateException("No console available");
}
```

> **Note**
>
> `System.console()` peut retourner `null` quand aucune console n'est disponible (par ex. IDE, entrée redirigée).

La présence d'une console dépend de la plateforme sous-jacente et de la manière dont la JVM est lancée.

Si la JVM est démarrée depuis une ligne de commande interactive et que les flux d'entrée/sortie standard ne sont pas redirigés, une console est généralement disponible.
Dans ce cas, la console est habituellement connectée au clavier et à l'affichage à partir desquels le programme a été lancé.

Si la JVM est démarrée dans un contexte non interactif — par exemple depuis un IDE, un planificateur de tâches en arrière-plan, un gestionnaire de services, ou avec des flux standard redirigés — une console ne sera généralement pas disponible.

Lorsqu'une console existe, elle est représentée par une instance unique de la classe Console, qui peut être obtenue en invoquant la méthode `System.console()`. Si aucun périphérique console n'est disponible, cette méthode retourne `null`.

36.6.1 Lire l'Input depuis Console

```
String name = console.readLine("Name: ");
```

`readLine()` affiche un prompt et lit une ligne complète d'entrée.

36.6.2 Lire des Mots de Passe de Manière Sécurisée

Console permet de lire des mots de passe sans afficher les caractères.

```
char[] password = console.readPassword("Password: ");
```

> **Note**
>
> Les mots de passe sont retournés en `char[]` afin qu'ils puissent être effacés de la mémoire.

36.7 Formater l'Output de la Console

Console supporte aussi une sortie formatée, similaire à PrintStream.

```
console.printf("Welcome %s%n", name);
```

Ceci utilise les mêmes spécificateurs de format que `printf()`.

36.8 Comparaison entre Console, Scanner et BufferedReader

API	Cas d'usage	Points forts	Limitations
`BufferedReader`	Entrée textuelle simple	Rapide, prévisible, charset explicite	Parsing manuel
`Scanner`	Entrée basée sur tokens / parsing	Pratique, expressive	Plus lente, comportement des tokens subtil
`Console`	Apps console interactives	Mots de passe, prompts, E/S formatées	Peut ne pas être disponible (`null`)

36.9 Redirection et Flux Standard

Les flux standard peuvent être redirigés par le système d'exploitation. Le code Java ne doit pas changer.

```
java App < input.txt > output.txt
```

Du point de vue du programme, `System.in` et `System.out` se comportent encore comme des flux normaux.

> **Note**
>
> La redirection est gérée par le système d'exploitation ou par le shell. Le code Java ne doit pas changer pour la supporter.

36.10 Pièges Communs et Bonnes Pratiques

- PrintStream supprime les IOExceptions
- `System.console()` peut retourner null
- Ne pas fermer `System.out` ou `System.err`
- Scanner mélange parsing et lecture
- Console est préférable pour les mots de passe
- Si tu utilises `Scanner` sur `System.in`, ne ferme pas le Scanner si d'autres parties du programme doivent encore lire depuis `System.in` (fermer le Scanner ferme `System.in`).

36.11 Synthèse Finale

- `System.out` et `System.err` sont des PrintStream pour la sortie
- `System.in` est un flux d'octets qui doit être adapté pour le texte
- `BufferedReader` et `Scanner` sont des stratégies communes d'entrée
- `Console` fournit une entrée et une sortie interactives sûres
- Les flux standard s'intègrent naturellement avec la redirection du système d'exploitation

---

Java Platform Module System (JPMS)

37. Java Platform Module System (JPMS)

Table des matières

Le `Java Platform Module System` (**JPMS**) a été introduit en Java 9.

C'est un mécanisme au niveau du langage et au niveau du runtime pour structurer les applications Java en unités fortement encapsulées appelées `modules`.

JPMS influence la manière dont le code est :

- organisé
- compilé
- lié
- empaqueté
- chargé au runtime

Comprendre JPMS est essentiel pour le Java moderne, en particulier pour les grandes applications, les bibliothèques, les images de runtime et les outils de déploiement.

37.1 Pourquoi les modules ont été introduits

Avant Java 9, les applications Java étaient construites en utilisant uniquement :

- des `packages`
- des fichiers `JAR`
- le `classpath`

Ce modèle présentait des limitations sérieuses à mesure que les applications grandissaient.

37.1.1 Problèmes avec le classpath

Le classpath est une liste plate de JAR dans laquelle :

- toutes les classes publiques sont accessibles à tous
- il n'existe pas de déclaration fiable des dépendances
- les versions en conflit sont courantes
- l'encapsulation est faible ou inexistante

- des classes dupliquées s'écrasent silencieusement en fonction de l'ordre du classpath

Cela a conduit à des problèmes bien connus tels que :

- "JAR hell"
- des bugs liés à l'ordre du classpath
- l'utilisation accidentelle d'API internes
- des erreurs d'exécution qui n'étaient pas détectées à la compilation

37.1.2 Exemple d'un problème de classpath

Supposons que deux bibliothèques dépendent de versions différentes du même JAR tiers.

Une seule version peut être placée sur le classpath.

La version choisie dépend uniquement de l'ordre du classpath, et non de l'appropriation réelle.

> **Note**
>
> Ce problème ne peut pas être résolu de manière fiable avec le seul outil du classpath.

37.2 Qu'est-ce qu'un module ?

Un `module` est une unité de code nommée et auto-descriptive.

De manière synthétique, un module est un ensemble d'un ou plusieurs packages liés, accompagné d'un fichier descripteur de module qui définit explicitement ses dépendances et les fonctionnalités qu'il met à disposition.

Un module fournit ainsi à ses utilisateurs un ensemble de fonctionnalités clairement défini et contrôlé.

Chaque module nommé possède un nom unique qui l'identifie auprès du compilateur et du système de modules.

Il déclare explicitement :

- de quoi il dépend
- ce qu'il expose aux autres modules
- ce qu'il garde caché

Un module est plus fort qu'un package et plus structuré qu'un JAR.

37.2.1 Propriétés fondamentales des modules

Propriété	Description
Encapsulation forte	Les packages sont cachés par défaut
Dépendances explicites	Les dépendances doivent être déclarées
Configuration fiable	Les dépendances manquantes provoquent des erreurs précoces
Identité nommée	Chaque module possède un nom unique

37.2.2 Module vs package vs JAR

Concept	But	Encapsulation
Package	Regroupement d'espace de noms	Faible (public reste visible)
JAR	Empaquetage / déploiement	Aucune (toutes les classes visibles lorsqu'elles sont sur le classpath)
Module	Encapsulation + unité de dépendance	Forte (packages non exportés cachés)

37.3 Le descripteur `module-info.java`

Chaque `module nommé` est défini par un fichier descripteur de module appelé :

```
module-info.java
```

Ce fichier décrit le module au compilateur et au runtime.

37.3.1 Descripteur de module minimal

Un descripteur de module minimal déclare uniquement le nom du module. Le nom du fichier doit être exactement `module-info.java`, et il doit se trouver à la racine de l'arbre des sources du module.

```
module com.example.hello {
}
```

> **Note**
>
> **Un module sans directives n'exporte rien et ne dépend de rien**.

37.4 Structure des répertoires d'un module

Un projet modulaire suit une structure standard de répertoires.

Le descripteur du module se trouve à la racine de l'arbre des sources du module.

```
src/
└─ com.example.hello/
    ├─ module-info.java
    └─ com/
        └─ example/
            └─ hello/
                └─ Main.java
```

Points clés :

- Le **nom du répertoire correspond au nom du module**
- `module-info.java` se trouve en haut de la racine des sources du module
- les packages suivent les règles standard de nommage Java

> **Note**
>
> Dans les projets IDE et build-tool, la structure des fichiers peut différer (par ex. Maven utilise `src/main/java`). Ce qui reste toujours vrai : `module-info.java` se trouve à la racine de l'arbre des sources du module et les chemins des packages suivent le nommage standard Java.

37.5 Un premier programme modulaire

Créons une application modulaire minimale.

37.5.1 Classe principale

```
package com.example.hello;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, modular world!");
    }
}
```

37.5.2 Descripteur du module

```
module com.example.hello {
    exports com.example.hello;
}
```

La `directive exports` rend le `package` accessible aux autres modules.

Sans elle, le package est encapsulé et inaccessible.

37.6 Explication de l'encapsulation forte

Dans `JPMS`, les packages NE sont PAS accessibles par défaut.

Même les classes public sont cachées à moins d'être exportées explicitement.

Dans les modules, `public` signifie "public vers les autres modules *seulement si* le package conteneur est exporté."

Situation	Accessible depuis un autre module ?
Classe public dans un package non exporté	Non
Classe public dans un package exporté	Oui
Membre protected dans un package exporté	Oui, mais seulement via héritage (pas accès général)
Classe/membre package-private (n'importe quel package)	Non
Membre private	Non

> **Note**
>
> C'est une différence fondamentale par rapport au modèle basé sur le classpath.

37.7 Synthèse des idées clés

- `JPMS` introduit les modules comme unités fortes d'encapsulation
- Les dépendances sont explicites et vérifiées
- `module-info.java` est le descripteur central

- Les packages sont cachés sauf s'ils sont exportés
- La visibilité basée sur le classpath ne s'applique plus dans les modules
- La visibilité public n'est plus suffisante : les exports du module contrôlent l'accessibilité

◀ 36. Interagir avec l'Utilisateur (Flux E/S Standard) | ▲ Index | 38. Compiler, Empaqueter et Exécuter des Modules ▶

38. Compiler, Empaqueter et Exécuter des Modules

Table des matières

Une fois qu'un `module` est défini avec un fichier `module-info.java`, il doit être compilé, empaqueté et exécuté à l'aide d'outils conscients des modules.

Cette section explique comment la `toolchain Java` change lorsque des modules sont impliqués.

38.1 Le Module Path vs le Classpath

`JPMS` introduit un nouveau concept : le **module path**.

Il existe aux côtés du **classpath** traditionnel, mais les deux se comportent de manière très différente.

Aspect	Classpath	Module path
Structure	Liste plate de JAR	Modules avec identité
Encapsulation	Aucune	Forte
Vérification des dépendances	Aucune	Stricte
Split packages	Autorisés	Interdits (modules nommés)
Ordre de résolution	Dépendant de l'ordre	Déterministe

> **Note**
>
> - Un JAR placé sur le `module path` est traité comme un `module` :
>   - S il contient un `module-info.class`, il devient un `named module`.
>   - S il ne contient pas de descripteur de module, il devient un `automatic module`.
> - Un JAR placé sur le classpath n est pas traité comme un module.
>   - À la place, il devient partie du unnamed module, avec toutes les autres entrées du classpath.
> - Un JAR modulaire (c est-à-dire un JAR contenant module-info.class) peut toujours être utilisé comme un JAR régulier.
>   - S il est placé sur le classpath au lieu du module path, il est traité comme partie du unnamed module, permettant aux applications non modulaires de l utiliser sans adopter le module system.
> - Split packages :
>   - Sont autorisés sur le classpath (plusieurs JAR peuvent contenir des classes dans le même package).
>   - Sont interdits pour les named modules ou automatic modules sur le module path.

38.2 Options de Ligne de Commande Relatives aux Modules

Lorsqu'on travaille avec le Java Module System, `java` et `javac` fournissent des options spécifiques pour compiler et exécuter des applications modulaires.

Certaines options sont communes, tandis que d'autres sont spécifiques à un outil.

38.2.1 Options Disponibles dans `java` et `javac`

Ces options peuvent être utilisées aussi bien lors de la compilation que lors de l'exécution :

- **`--module`** ou **`-m`**
  Utilisée pour compiler ou exécuter uniquement le module spécifié.
- **`--module-path`** ou **`-p`**
  Spécifie les chemins dans lesquels `java` ou `javac` rechercheront les définitions de modules.

Le système de modules Java fournit trois options spéciales en ligne de commande, utilisables avec `javac` et `java`, qui permettent de modifier temporairement les règles d'accès entre modules sans changer les fichiers `module-info.java`. Ces options s'appliquent uniquement à l'exécution courante de la commande et ne modifient pas de manière permanente les descripteurs de modules.

Les trois options sont :

- `--add-reads`
- `--add-exports`
- `--add-opens`

Elles sont généralement utilisées pour les tests, la rétrocompatibilité, les phases de migration, ou lorsque l'on travaille avec des modules tiers que l'on ne peut pas modifier.

Supposons par exemple que `moduleA` doive accéder aux types publics de `moduleB`, mais que :

- `moduleA` ne déclare pas `requires moduleB;`
- `moduleB` n'exporte pas le package requis vers `moduleA`

Au lieu de modifier les fichiers `module-info.java`, il est possible d'accorder temporairement l'accès nécessaire avec :

```
javac --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...

java  --add-reads moduleA=moduleB \
      --add-exports moduleB/com.modB.package1=moduleA \
      ...
```

Signification des options :

- `--add-reads moduleA=moduleB`
  Déclare temporairement que `moduleA` lit `moduleB`.
  Cela revient à ajouter `requires moduleB;` dans le descripteur de `moduleA`.
  Ainsi, `moduleA` peut accéder aux packages exportés de `moduleB`.
- `--add-exports moduleB/com.modB.package1=moduleA`
  Exporte temporairement le package `com.modB.package1` du module `moduleB` vers `moduleA`.
  Cela équivaut à ajouter :
  `exports com.modB.package1 to moduleA;`
  dans le descripteur de `moduleB`.

Distinction importante :

- `--add-reads` établit la lisibilité au niveau du module.
- `--add-exports` accorde l'accès à des packages spécifiques.
- `--add-opens` (non illustré ci-dessus) fonctionne comme `--add-exports`, mais autorise également l'accès par réflexion profonde (deep reflection), souvent nécessaire pour certains frameworks.

Ces options ne modifient pas les métadonnées compilées du module ; elles ajustent simplement le graphe des modules pour l'invocation spécifique de `javac` ou `java`.

38.2.2 Options Applicables Uniquement à `javac`

Ces options s'appliquent uniquement à la phase de compilation :

- **`--module-source-path`**
  (aucun raccourci)
  Utilisée par `javac` pour localiser les définitions des modules source.
- **`-d`**
  Spécifie le répertoire de sortie dans lequel les fichiers `.class` seront générés après la compilation.

38.2.3 Options Applicables Uniquement à `java`

Ces options s'appliquent uniquement à la phase d'exécution :

- **`--list-modules`**
  (aucun raccourci)
  Liste tous les modules observables puis termine.
- **`--show-module-resolution`**
  (aucun raccourci)
  Affiche les détails de la résolution des modules lors du démarrage de l'application.
- **`--describe-module`** ou **`-d`**
  Décrit un module spécifié puis termine.

38.2.4 Distinctions Importantes

L'option `-d` a des significations différentes selon l'outil :

- Dans **`javac`**, `-d` définit le répertoire de sortie des fichiers de classes compilés.
- Dans **`java`**, `-d` est un raccourci pour `--describe-module`.

De plus, `-d` ne doit pas être confondue avec **`-D`** (D majuscule).

- **`-D`** est utilisée lors de l'exécution d'un programme Java pour définir des propriétés système sous forme de paires nom-valeur sur la ligne de commande.

```
java -Dconfig.file=app.properties com.example.Main
```

Dans cet exemple, `-Dconfig.file=app.properties` définit une propriété système accessible à l'exécution via `System.getProperty("config.file")`.

38.3 Compiler un Module Unique

Pour compiler un module, vous devez spécifier le chemin des sources du module et le répertoire de destination.

```
javac -d out \
src/com.example.hello/module-info.java \
src/com.example.hello/com/example/hello/Main.java
```

Une approche plus évolutive utilise `--module-source-path`.

```
javac --module-source-path src \
      -d out \
      $(find src -name "*.java")
```

> **Note**
>
> `--module-source-path` indique à javac où trouver plusieurs modules à la fois.

38.4 Compiler des Modules Multiples Interdépendants

Lorsque des modules dépendent les uns des autres, leurs dépendances doivent être résolubles à la compilation.

`--module-path` **mods** (répertoire d'exemple contenant des modules interdépendants) doit contenir des JAR modulaires déjà compilés ou des répertoires de modules compilés (chacun avec son propre module-info.class).

```
javac -d out \
--module-source-path src \
--module-path mods \
$(find src -name "*.java")
```

Ici :

- `--module-source-path` localise les arbres de sources des modules
- `--module-path` fournit des modules déjà compilés

38.5 Empaqueter un Module dans un JAR Modulaire

Après la compilation, les modules sont généralement empaquetés sous forme de fichiers JAR.

Un JAR modulaire contient un `module-info.class` à sa racine.

Si `module-info.class` est présent, le JAR devient automatiquement un `module nommé` et son `nom` est pris depuis le descripteur (et non depuis le nom du fichier).

```
jar --create \
--file mods/com.example.hello.jar \
--main-class com.example.hello.Main \
-C out/com.example.hello .
```

> **Note**
>
> Un JAR avec `module-info.class` est un `module nommé, pas un module automatique`. Lorsqu'un JAR contient un `module-info.class`, son nom de module est pris depuis ce fichier et n'est pas déduit du nom du fichier.

38.6 Exécuter une Application Modulaire

Pour exécuter une application modulaire, vous utilisez le `module path` et spécifiez le `nom du module`.

```
java --module-path mods \
--module com.example.hello/com.example.hello.Main
```

Vous pouvez raccourcir cela en utilisant les options `-p` et `-m`.

```
java -p mods -m com.example.hello/com.example.hello.Main
```

> **Note**
>
> Lors de l'utilisation de modules nommés, le classpath est ignoré pour la résolution des dépendances entre modules.

38.7 Explication des Directives de Module

Le fichier `module-info.java` contient des directives qui décrivent les dépendances, la visibilité et les services.

Chaque directive a une signification précise.

38.7.1 `requires`

La directive `requires` déclare une dépendance vers un autre module.

Sans elle, les types du module dépendant ne peuvent pas être utilisés.

```
module com.example.app {
    requires com.example.lib;
}
```

Effets de requires :

- La dépendance doit être présente à la compilation et à l'exécution
- Les packages exportés du module requis deviennent accessibles

38.7.2 `requires transitive`

`requires transitive` expose une dépendance aux modules en aval.

Il propage la lisibilité.

```
module com.example.lib {
    requires transitive com.example.util;
    exports com.example.lib.api;
}
```

Signification :

- **Tout module qui requiert com.example.lib lit automatiquement com.example.util**
- **Les appelants n'ont pas besoin de déclarer requires com.example.util explicitement**

> **Note**
>
> Cela est similaire aux « dépendances publiques » dans d'autres systèmes de modules.
>
> Lisible ≠ exporté : une dépendance transitive n'exporte pas automatiquement vos packages.

38.7.3 `exports`

`exports` rend un package accessible aux autres modules.

Seuls les packages exportés sont visibles à l'extérieur du module.

```
module com.example.lib {
    exports com.example.lib.api;
}
```

Les packages non exportés restent fortement encapsulés.

38.7.4 `exports ... to` (Exports Qualifiés)

Un export qualifié restreint l'accès à des modules spécifiques.

```
module com.example.lib {
    exports com.example.internal to com.example.friend;
}
```

Seuls les modules listés peuvent accéder au package exporté.

38.7.5 `opens`

`opens` permet un accès réflexif profond à un package.

Il est principalement utilisé par des frameworks utilisant la réflexion.

```
module com.example.app {
    opens com.example.app.model;
}
```

> **Note**
>
> opens NE rend PAS un package accessible à la compilation. Il affecte uniquement la réflexion à l'exécution.

38.7.6 `opens ... to` (Opens Qualifiés)

Vous pouvez restreindre l'accès réflexif à des modules spécifiques.

```
module com.example.app {
    opens com.example.app.model to com.fasterxml.jackson.databind;
}
```

> **Note**
>
> `opens` affecte la réflexion ; `exports` affecte la compilation et la visibilité des types.

38.7.7 Table des Directives Principales

Directive	But
`requires`	Déclarer une dépendance
`requires transitive`	Propager une dépendance
`exports`	Exposer un package
`exports ... to`	Exposer à des modules spécifiques
`opens`	Autoriser la réflexion à l'exécution
`opens ... to`	Restreindre l'accès réflexif

38.7.8 Exports vs Opens — Accès à la Compilation vs à lExécution

Visibilité	Compilation ?	Réflexion à l'exécution ?
`exports`	Oui	Non
`opens`	Non	Oui
`exports ... to`	Oui (modules limités)	Non
`opens ... to`	Non	Oui (modules limités)

> **Important**
>
> `JPMS` ajoute un `module path`, mais le `classpath` existe toujours. Ils peuvent coexister, mais les modules nommés ont la priorité.

38.8 Modules Nommés, Automatiques et Unnamed

`JPMS` supporte différents types de modules afin de permettre une migration progressive depuis le classpath.

JPMS doit interopérer avec du code legacy.

Pour supporter l'adoption progressive, la JVM reconnaît trois catégories différentes de modules.

38.8.1 Modules Nommés

Un `module nommé` possède un `module-info.class` et une identité stable.

- Encapsulation forte
- Dépendances explicites
- Support complet JPMS

38.8.2 Modules Automatiques

Un JAR sans `module-info` placé sur le `module path` devient un `module automatique`.

Son nom est dérivé du nom du fichier JAR.

- Lit tous les autres modules
- Exporte tous les packages
- Pas d'encapsulation forte

> **Note**
>
> Les modules automatiques existent pour faciliter la migration. Ils ne conviennent pas comme conception à long terme.

38.8.3 Module Unnamed

Le code sur le classpath appartient au `module unnamed`.

- Lit tous les modules nommés
- Tous les packages sont ouverts
- Ne peut pas être requis par des modules nommés

> **Note**
>
> Le `module unnamed` préserve le comportement legacy du classpath.

38.8.4 Résumé Comparatif

Type de module	module-info présent ?	Encapsulation	Lit
`Named`	Oui	Forte	Déclarés seulement
`Automatic`	Non	Faible	Tous les modules
`Unnamed`	Non	Aucune	Tous les modules

38.9 Approche Top-Down et Bottom-Up pour modulariser une application

Lors de la migration d'une application existante (non modulaire) vers le Java Platform Module System (JPMS), deux stratégies principales peuvent être adoptées : **top-down** et **bottom-up**.
Ces deux approches nécessitent une compréhension claire des interactions entre les **modules nommés**, les **modules automatiques** et le **module non nommé**.

38.9.1 Approche Top-Down

Dans une `approche top-down`, on commence par **modulariser le module principal de l'application**, puis on migre progressivement ses dépendances.

38.9.1.1 Règles fondamentales

1. **Un JAR placé sur le module path devient un module automatique.**
    - Son nom est déterminé soit :
        - À partir de l'entrée `Automatic-Module-Name` dans le manifeste,
        - Soit dérivé du nom du fichier JAR (les tirets sont remplacés par des points et les numéros de version sont ignorés).

Exemple :
`mysql-connector-java-8.0.11.jar` → `mysql.connector.java`

- Un `module automatique` :
  - Exporte tous ses packages.
  - Lit tous les autres modules.

2. **Un JAR placé sur le classpath appartient au module non nommé.**
   - Le `module non nommé` :
     - Exporte tous ses packages.
     - Peut lire tous les autres modules.
   - Cependant, il n'a pas de nom ; aucun module ne peut donc déclarer `requires` à son égard.
3. **Les modules nommés explicitement (avec un fichier `module-info.java`)**
   - Peuvent déclarer des dépendances à l'aide de :

     ```
     requires some.module;
     ```

   - Peuvent dépendre :
     - D'autres modules nommés
     - De modules automatiques
   - Ne peuvent pas dépendre du module non nommé (puisqu'il n'a pas de nom).

Conséquence importante :

> Un `module nommé` peut lire un `module automatique`, mais il ne peut pas lire le `module non nommé`.

38.9.1.2 Implications pratiques

Supposons :

- JAR de l'application = `A`
- `A` dépend directement de `B`
- `B` dépend de `C`

Si vous modularisez `A` en premier :

- `A` doit déclarer `requires B;`
- Donc `B` doit être placé sur le module path (module nommé ou automatique)
- Si `B` devient un module nommé :
  - `C` doit également être placé sur le module path (nommé ou automatique)

Ainsi, dans une migration top-down :

- On commence par la couche application.
- On modularise progressivement les dépendances vers l'extérieur.
- Les modules automatiques sont souvent utilisés temporairement pendant la transition.

38.9.1.3 Résumé des règles d'accès

Type de module	Exporte	Peut lire
`Module nommé`	Exportations déclarées uniquement	Modules requis uniquement
`Module automatique`	Tous les packages	Tous les modules
`Module non nommé`	Tous les packages	Tous les modules

> **Important**
>
> - Les `modules automatiques et non nommés` sont **permissifs**.
> - Les `modules nommés` imposent des règles explicites de dépendance et d'export.

38.9.2 Approche Bottom-Up

Dans une `approche bottom-up`, on commence par modulariser les `bibliothèques de plus bas niveau`, puis on progresse vers les modules de niveau supérieur, `jusqu'à l'application principale`.

38.9.2.1 Stratégie principale

On convertit d'abord les bibliothèques fondamentales en modules nommés correctement définis avec un descripteur explicite `module-info.java`.

Ensuite :

- Les modules qui en dépendent sont modularisés.
- Enfin, l'application principale devient elle aussi un module nommé.

Cette approche met l'accent sur :

- Des relations `requires` explicites
- Des `exports` maîtrisés
- Une encapsulation forte dès le départ

38.9.2.2 Avantages architecturaux

Comparés aux modules automatiques :

- Les modules nommés n'exportent que ce qui est explicitement déclaré.
- Ils ne lisent pas implicitement tous les autres modules.
- Les frontières d'encapsulation sont clairement définies.

La modularisation bottom-up conduit généralement à :

- Un graphe de dépendances plus propre
- Une meilleure maintenabilité
- Des frontières de modules plus solides

38.9.3 Comparaison conceptuelle

Top-Down

- On commence par l'application principale.
- Les dépendances sont modularisées selon les besoins.
- On s'appuie souvent temporairement sur des modules automatiques.
- Migration initiale plus rapide.

Bottom-Up

- On commence par les bibliothèques cœur.
- Les descripteurs de modules sont définis strictement dès le départ.
- La migration progresse vers le haut.
- Architecture modulaire plus disciplinée et robuste.

38.9.4 Perspective de migration

En pratique, les projets réels combinent souvent les deux stratégies :

- Une migration top-down permet d'activer rapidement l'exécution modulaire.
- Une phase de raffinement bottom-up remplace ensuite les modules automatiques par des modules nommés correctement définis.

Cette approche hybride permet une adoption progressive du JPMS tout en renforçant progressivement l'encapsulation et la clarté architecturale.

38.10 Inspection des Modules et des Dépendances

38.10.1 Décrire les Modules avec java

```
java --describe-module java.sql
```

Cela affiche `exports`, `requires` et `services` d'un module.

38.10.2 Décrire les JAR Modulaires

```
jar --describe-module --file mylib.jar
```

38.10.3 Analyser les Dépendances avec `jdeps`

`jdeps` analyse statiquement les dépendances de classes et de modules.

```
jdeps myapp.jar
```

```
jdeps --module-path mods --check my.module
```

Pour détecter l'utilisation d'API internes du JDK :

```
jdeps --jdk-internals myapp.jar
```

38.11 Créer des Images Runtime Personnalisées avec `jlink`

`jlink` construit un runtime Java minimal contenant uniquement les modules requis par une application.

```
jlink
--module-path $JAVA_HOME/jmods:mods
--add-modules com.example.app
--output runtime-image
```

Avantages :

- runtime plus petit
- démarrage plus rapide
- aucun module JDK inutilisé

38.12 Créer des Applications Autonomes avec `jpackage`

`jpackage` construit des installateurs spécifiques à la plateforme ou des images applicatives.

```
jpackage
--name MyApp
--input mods
--main-module com.example.app/com.example.Main
```

`jpackage` peut produire :

- .exe / .msi (Windows)
- .pkg / .dmg (macOS)
- .deb / .rpm (Linux)

38.13 Résumé Final JPMS en Pratique

- `JPMS` introduit une `encapsulation forte` et des dépendances fiables
- Les `modules` remplacent les conventions fragiles du classpath
- Les `services` permettent des architectures découplées
- Les `modules automatiques` et le `module unnamed` supportent la migration
- `jlink` et `jpackage` permettent des modèles modernes de déploiement

◀ 37. Java Platform Module System (JPMS) | ▲ Index | 39. Services dans JPMS (Le Modèle ServiceLoader) ▶

39. Services dans JPMS (Le Modèle ServiceLoader)

Table des matières

`JPMS` inclut un mécanisme de service intégré qui permet aux `modules` de découvrir et dutiliser des implémentations à l'exécution sans coder en dur des dépendances entre `fournisseurs` et `consommateurs`.

Ce mécanisme est basé sur l'`API ServiceLoader` existante, mais les modules le rendent fiable, explicite et sûr.

39.1 Le Problème que les Services Résolvent

Parfois un module doit utiliser une capacité, mais ne devrait pas dépendre dune implémentation spécifique.

Des exemples typiques incluent : - frameworks de journalisation - pilotes de base de données - systèmes de plugins - fournisseurs de service sélectionnés à l'exécution

Sans services, le consommateur devrait dépendre directement dune implémentation concrète.

Cela crée un couplage fort et réduit la flexibilité.

39.1.1 Rôles dans le Modèle de Service

Le `modèle de service JPMS` implique quatre rôles distincts.

Rôle	Description
`Interface de service`	Définit le contrat
`Fournisseur de service`	Implémente le service
`Consommateur de service`	Utilise le service
`Service loader`	Découvre les implémentations à l'exécution

39.1.2 Module Interface de Service

L'`interface de service` définit l'API dont les `consommateurs` dépendent.

Elle doit être exportée afin que les autres modules puissent la voir.

```
package com.example.service;

public interface GreetingService {
    String greet(String name);
}
```

```
module com.example.service {
    exports com.example.service;
}
```

> **Note**
>
> Le module d'interface de service ne devrait contenir aucune implémentation.

39.1.3 Module Fournisseur de Service

Un `module fournisseur` implémente l'interface de service et déclare quil fournit le service.

```
package com.example.service.impl;

import com.example.service.GreetingService;

public class EnglishGreeting implements GreetingService {
    public String greet(String name) {
        return "Hello " + name;
    }
}
```

```
module com.example.provider.english {
    requires com.example.service;
    provides com.example.service.GreetingService with com.example.service.impl.EnglishGreeting
}
```

Points clés : - Le `fournisseur` dépend de l'`interface de service` - La classe d'implémentation n'a pas besoin d'être exportée - La directive `provides` enregistre l'implémentation

39.1.4 Module Consommateur de Service

Le `module consommateur` déclare quil utilise un service, mais ne nomme aucune implémentation.

```
module com.example.consumer {
    requires com.example.service;
    uses com.example.service.GreetingService;
}
```

> **Note**
>
> `uses` déclare l'intention de découvrir des implémentations à l'exécution.
>
> Un module qui déclare `uses` mais ne possède aucun fournisseur correspondant sur le module path compile normalement, mais `ServiceLoader` retourne un résultat vide à l'exécution.

39.1.5 Chargement des Services à lExécution

L'`API ServiceLoader` effectue la découverte de service.

Elle trouve tous les fournisseurs visibles pour le graphe des modules.

```
ServiceLoader<GreetingService> loader =
    ServiceLoader.load(GreetingService.class);

for (GreetingService service : loader) {
    System.out.println(service.greet("World"));
}
```

`JPMS` garantit que seuls les fournisseurs déclarés sont découverts.

La découverte "accidentelle" basée sur le classpath est empêchée.

39.1.6 Règles de Résolution des Services

Pour qu'un service soit découvrable par `ServiceLoader`, plusieurs conditions doivent être satisfaites :

Règle	Signification
Le module fournisseur doit être lisible	Résolu par le graphe `requires`
L'interface de service doit être exportée	Les consommateurs doivent la voir
Le consommateur doit déclarer `uses`	Sinon ServiceLoader échoue
Le fournisseur doit déclarer `provides`	La découverte implicite est interdite

39.1.7 Couche Service Locator

Il est possible dintroduire une couche supplémentaire appelée `Service Locator`.

Dans cette architecture :

- Le `consommateur` n'utilise pas directement `ServiceLoader`
- Le `Service Locator` est le seul composant qui déclare `uses`
- Le `consommateur` dépend du `Service Locator`

Structure architecturale :

```
Consommateur → Service Locator → ServiceLoader → Fournisseur
```

Module du Service Locator :

```
module com.example.locator {
    requires com.example.service;
    uses com.example.service.GreetingService;
}
```

Classe Service Locator :

```
package com.example.locator;

import java.util.ServiceLoader;
import com.example.service.GreetingService;

public class GreetingLocator {

    public static GreetingService getService() {
        return ServiceLoader
                .load(GreetingService.class)
                .findFirst()
                .orElseThrow();
    }
}
```

Module Consommateur :

```
module com.example.consumer {
    requires com.example.locator;
}
```

Le consommateur ne déclare pas `uses` parce quil ninvoque pas directement `ServiceLoader`.

39.1.8 Schéma Séquentiel dInvocation

Séquence dexécution :

1. Le `Consommateur` invoque `GreetingLocator.getService()`
2. Le `Service Locator` invoque `ServiceLoader.load(...)`
3. Le `ServiceLoader` consulte le graphe des modules
4. Le système identifie les modules qui déclarent `provides`
5. L'implémentation du `Fournisseur` est instanciée
6. L'instance est retournée au `Consommateur`

Schéma séquentiel :

```
Consommateur
   |
   | 1. getService()
   ▼
Service Locator
   |
   | 2. ServiceLoader.load()
   ▼
ServiceLoader
   |
   | 3. Résolution du fournisseur
   ▼
Implémentation Fournisseur
   |
   | 4. Instance retournée
   ▼
Consommateur
```

39.1.9 Tableau Récapitulatif des Composants

Composant	Rôle	exports	requires	uses	provides
SPI	Définit le contrat	✅	❌	❌	❌
Fournisseur	Implémente le service	❌	✅	❌	✅
Service Locator	Effectue la découverte	(optionnel)	✅	✅	❌
Consommateur	Utilise le service	❌	✅	❌	❌
